FUNCTIONAL SPECIFICATION AND TESTING
OF LOGIC CIRCUITS

M. S. ABADIR
Department of Electrical Engineering, University of Southern California, CA, U.S.A

and

H. K. REGHBATI
Department of Computing Science, Simon Fraser University, Burnaby, B.C., V5A IS6, Canada

(Received 28 May 1985)

Communicated by Nick Cerone

Abstract—The very rapid growth in the complexity of testing logic circuits presents a description problem of increasing severity. In this paper two approaches to this problem are discussed and compared. In the first approach path-tracing techniques are used to generate the experiment description of logic circuits from their binary decision diagram representation. In the second approach symbolic execution is used to provide a link between the functional description of a logic circuit, written in a procedural language, and its nonprocedural assertion description. A commercial logic circuit is used throughout the paper as a running example.

1 INTRODUCTION

The very rapidly increasing complexity of digital devices and systems has placed increased emphasis on the problem of describing these devices with a concise description that could be used in analyzing, implementing, and testing the devices. The densities of digital devices have been doubling each year since the early 1960s. Description techniques which specify the actual implementation of the device are becoming practically unfeasible because such implementations are either unknown or too large and complicated to be efficiently handled. A possible solution is to describe these devices in terms of their input/output relationships using an implementation-free functional description.

In this paper we are mainly concerned with the use of functional description tools in the area of hardware testing, and in particular functional testing. Functional testing can be defined as the process of verifying that a given device does what it is supposed to do[1]. For example, if the device is a cell in the memory, the process of functional testing must check that the read operation reads the cell content, the write operation writes the desired value in the cell, the cell can hold its information, etc. Thus, we can reformulate the definition of functional testing as the process of checking the performance of a device in its various modes of operation. Clearly, functional testing requires a description of the device’s performance in each mode of operation.

Unfortunately, classical functional description methods—such as Karnaugh maps, truth tables, and state tables—have the property of growing exponentially with the number of variables involved. Thus, they become completely unmanageable for many MSI circuits, not to mention LSI and VLSI. On the other hand, many hardware description languages have been developed to provide a concise functional description of the device or system under consideration. These languages can be classified into two main categories: procedural and nonprocedural languages[2]. Procedural description languages are similar to high-level programming languages; hence they are easy to write, modify, and understand. However, the main disadvantage of this class of languages is their inability to provide the user with an explicit description of the device’s modes of operation. Such information is essential to any test generation system, and they could be obtained from nonprocedural descriptions of the device. However, nonprocedural descriptions are very hard to generate.

In this paper we will show how we can bridge this description gap using two different approaches. The first approach involves the use of a special type of graph called a binary decision diagram to define the digital function under consideration[3]. These types of graphs are amenable to storage and manipulation on digital computers. The diagrams provide a straight
forward means of automatically generating a complete set of experiments describing all the different modes of operation\cite{1}. In section 2 we will describe binary decision diagrams and how they may be derived for various digital devices. Techniques will then be described for using the diagram to generate the device set of experiments and how these descriptions can simplify many test generation problems. In the second approach, we will use the technique introduced by Oakley\cite{4} to transform a procedural description of a device into a nonprocedural assertion description and we will show how such assertions can be used as input to automatic test generation systems. The main aspects of this technique will be discussed in section 3. A comparative study between the two approaches will be given in section 4. Section 5 will summarize and conclude this paper.

2. TESTING WITH BINARY DECISION DIAGRAMS

Akers\cite{3} has introduced a very attractive way to specify the precise logical behavior of a digital device by means of a binary decision diagram\cite{5-7}. These diagrams not only permit concise description of large digital functions but they also easily accommodate the many operational constraints typically associated with such functions, which must be considered in the test generation process. These diagrams are especially amenable to storage and manipulation on a digital computer with the result that generation procedures become correspondingly simpler. In this section we shall examine the use of binary decision diagrams in the area of functional testing.

A binary decision diagram is nothing more than a concise means for completely defining the logical operation of one or more digital functions in an implementation-free form. The information usually found in an IC catalog is sufficient to derive the set of binary decision diagrams describing the functions performed by the different modules in that device. These diagrams (such as truth tables and state tables) are amenable to extensive logical analysis. However, they do not have the unpleasant property of growing exponentially with the number of variables involved, as in the case of truth tables and state tables. Moreover, the diagrams can be stored and processed very easily in a digital computer. For more details on binary decision diagrams, the reader should consult the original papers by Akers\cite{1,3,8}. The following example illustrates how binary decision diagrams are used to describe functional modules.

Example 1. Consider the SN54194 4-bit bidirectional shift register shown in Fig. 1(a). In normal operation this register can perform five operations: clear, hold, parallel load, shift right, and shift left. The functional behavior of this device is determined by the values of four control signals \((Z, Cl, G_1, \text{ and } G_0)\) as specified by the table shown in Fig. 1(b). The shift register has 10 external inputs and 4 output lines. The next state of the \(i\)th bit of the register is denoted by \(Q_i\), while the current state value of the same bit is denoted by \(Q_i\). Figure 2 shows the binary decision diagram which completely defines the operation of the \(i\)th bit of the register. Each node in the diagram is associated with a binary variable and there are two branches coming out from each node: the right branch is the one branch, while the left branch is the zero branch. According to the value of the node variable one of the two branches will be selected when

![Binary Decision Diagram](image)

<table>
<thead>
<tr>
<th>Z</th>
<th>Cl</th>
<th>G_1</th>
<th>G_0</th>
<th>Functional Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Clear every bit of the register.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Hold the register contents.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Hold the register contents.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Load the values of ((A_1, A_2, A_3, A_4)) in parallel.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Shift the register content one bit to the left; (Q_i) is fed into the register from the right.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Shift the register content one bit to the right; (Q_i) is fed into the register from the left.</td>
</tr>
</tbody>
</table>

Fig. 1. (a) A 4 bit bidirectional shift register; (b) its functional behavior.

\(^\dagger\)There are procedures to generate the binary decision diagram of a function from its truth table or from its boolean expression. Similar procedures can easily be constructed for functions described by state tables, or by a hardware description language.
processing the diagram. By entering the diagram at the node indicated by the arrow labelled with $Q^*_i$, and then proceeding through the diagram following the appropriate branches until a terminal value is reached, the value of $Q^*_i$ is determined.

As mentioned before, the problem of testing a given device can be described as the problem of validating the device performance in its different modes of operation. We will refer to the specification of one mode of a module’s performance as an experiment. Note that the testing problem is partitioned into small subproblems: each one involves validating one mode of operation described by an experiment. Of course, the relatively simple nature of individual experiments makes the problem much easier to handle [9, 10]. In the next example we will illustrate how the module’s complete set of experiments can be generated from its binary decision diagram.

**Example 2.** Consider the diagram of Figure 2. Every possible path starting at $Q^*_i$ represents one mode of operation, or equivalently specifies an experiment. It follows that if we trace all the possible paths from $Q^*_i$ to an exit value or to an exit variable, we will automatically obtain a complete and disjoint set of experiments for $Q^*_i$. Each experiment is formulated by simply recording the branch values of the variables involved in the path. The experiment’s output is obtained by recording the exit value or the value of the exit variable. All the variables which are not involved in a certain path are assigned x’s (an x means a don’t care value). Hence, the complete set of experiments describing the $i$th bit of the shift register is shown in Fig. 3.

Every individual experiment, in itself, is not a test. Rather it is a way of describing how the device will behave when some of the variables are fixed. An input combination is said to be contained by an experiment if the fixed variables of the experiment match those in the input pattern. It is important to note that every input combination is contained by one and only one experiment.

In the next subsection we will discuss how binary decision diagrams, or more precisely, the experiments generated from the diagram can simplify many of the problems associated with test generation.

![Fig. 2. Binary decision diagram for the shift register.](image)

![Fig. 3. The complete set of experiments for the $i$th bit of the shift register.](image)
2.1 **Functional testing with binary decision diagrams**

So far we have shown how we can extract the experiments of a device from its binary decision diagram. In the next example the effectiveness of these experiments in testing the internal lines of the device will be investigated.

**Example 3.** Consider the shift register of examples 1 and 2. Figure 4 shows a natural way of implementing 1 bit of the register using a D flip-flop.

Now suppose we use the experiments of Fig. 3 as a test pattern. The effectiveness of such a test in covering single stuck-at faults affecting the internal lines of the circuit (labelled a, b, c, d, e, f, and g) is of interest. Note that the external input and output lines of the circuit are directly available to the test generation system and faults affecting them can be dealt with explicitly. However, internal lines are unknown to the test generation system and we will rely on the experiments to exercise faults affecting these lines.

It can be easily shown that the experiments, when used as a test, will detect all single stuck-at faults affecting the internal lines except two faults: a stuck-at-1 and b stuck-at-1, where there is a 50% chance that they will be detected depending on the values of the unspecified variables in the experiments. For example, if $A_i = 1$ during the execution of the tenth experiment (see Fig. 3), then the fault a stuck-at-1 will be detected.

The above example indicates that the experiments can be used as a very useful test pattern, and they do cover many of the single stuck-at faults affecting internal lines of the circuit under consideration. It should also be noted that the only way we can get 100% single stuck-at faults coverage is by using an exhaustive test (too lengthy) or by starting from a gate level description of the circuit (which contradicts our initial assumption of approaching the problem at the functional level).

One of the testing techniques that may use the experiments of a device is the random testing approach. In this technique the test patterns are chosen at random[9]. The main drawback of this testing technique is that it does not guarantee the testing of all the modes of operation of the device. In other words, some of the modes of operation are checked over and over, while other modes are hardly checked at all. This problem is very serious in sequential devices. For example, consider our shift register: 50% of the random patterns will check whether the clear signal works or not. The device experiments can help in solving this problem, by guiding the random testing process through the different modes of operation. We will randomly test each mode of operation individually by fixing the fixed variables in the experiment, and then select the values of the unspecified variables at random. Clearly this technique will ensure that all the device modes of operation have been exercised during the testing process.

Akers[8] also introduced a new probabilistic technique for test generation from functional descriptions given by binary decision diagrams. The diagrams can be used to generate a set of input and output probabilities that could be used to guide the excitation and sensitization tasks.

![Fig. 4. Gate level realization of the ith bit of the shift register.](image-url)
during the test-generation process. We will not discuss this technique in detail, and the interested reader can refer to [8] for more details.

We have used binary decision diagrams as the main description vehicle in developing a new functional test-generation system. Space limitations prevent us from discussing the technique in any detail, and the interested reader can refer to [10]. The usefulness of binary decision diagrams in testing large-scale integrated circuits was demonstrated by this work.

3. SYMBOLIC EXECUTION OF HARDWARE DESCRIPTION LANGUAGES

This section describes how symbolic execution can be used to convert the procedural description of a digital device into a nonprocedural assertion description, suitable as an input to systems that require knowledge of the device modes of operation.

3.1 Hardware description languages

Hardware description languages have been considered a very efficient tool for describing digital hardware at different levels of abstraction. These languages can be classified into procedural and nonprocedural languages. The key difference is in the way of handling sequencing of activities[2]. Nonprocedural descriptions express the semantics of a hardware function more directly than the procedural descriptions. On the other hand, procedural descriptions are easier to write, understand, and verify than nonprocedural descriptions.

The process of testing a large digital circuit requires a concise description of the function of that circuit. More precisely, we have to describe all the possible disjoint modes of operation of that circuit by giving the input conditions and observing the effects of each mode. Thus, by exercising each mode of operation we can verify the correctness of the circuit. This information is called the assertion description.

The nonprocedural description of a digital circuit function is, by definition, very similar to assertion descriptions. Each statement in the assertion description states the effect of a certain activity and specifies the condition under which this activity can take place. The key difference between assertion description and nonprocedural description is that the former describes the disjoint modes of operation of the device, while the nonprocedural description describes the different device activities, with the possibility that more than one activity may be activated concurrently (i.e. the activities may not be disjoint). Thus a number of these activities may represent one mode of operation. However, from the testing point of view, the differences between assertion description and nonprocedural description are not very serious. That is why we will use the two names interchangeably, in the rest of our discussion, to address the same class of descriptions.

Unfortunately, writing a nonprocedural description for today’s complex LSI and VLSI circuits is very difficult, if it could be done at all. On the other hand, procedural descriptions are much easier to write and understand. However, it does not express the semantics of the digital circuit explicitly. Hence, it can not be used directly to generate tests for digital circuits. Symbolic execution can be used to bridge the gap between the two types of descriptions, as will be discussed in the next subsection.

3.2 Symbolic execution of procedural hardware descriptions

The use of the symbolic execution technique[11] to aid in testing and verifying computer hardware has received great attention in the last few years. IBM is developing a system to symbolically execute a hardware specification written in a high-level language. The results are used to verify the correctness of a given implementation[12]. Carter, Joyner, and Brand[13] described symbolic simulation, a method similar to symbolic execution, and its use in proving the correctness of machine architectures implemented in microcode. They developed a special machine description language called LSS (Language for Symbolic Simulation), based on APL, to help interactively verify the correctness of microcode.

The most attractive research in this area was pursued by Oakley[4]. The ultimate goal of Oakley’s work was to show the feasibility of converting a machine procedural description into a nonprocedural assertion description via symbolic execution. As stated before, such an assertion description will express the semantics of the machine, or more generally any hardware function,
more directly than a procedural description. In other words, we will obtain a list of all the modes of operation of that digital function. This list can be used, by any test generation procedure, as a checklist to generate the patterns required to exercise each mode of operation.

In example 4, we will illustrate the technique using our shift register. We will start with a procedural description and show how the symbolic execution process works and how its output can be used for test generation purposes. Example 5 shows the application of symbolic execution to the problem of testing a computing machine.

Example 4. Consider the shift register of section 2. The ISPS procedural description[14,15] of the shift register is given in Fig. 5. Note that the variable lengths in the description is limited to 1 bit only.

Symbolic execution can proceed as in normal execution except when the symbolic inputs are encountered. This can occur in two ways: computation of an expression involving symbolic inputs and conditional branching that is dependent on the symbolic inputs.

The basic computational operators, such as addition and multiplication, are extended to accept and return symbolic values. Variables on the left-hand side, called output ussrrtions, are set to the returned symbolic value. These symbolic values are called output assertions.

When a conditional statement is executed and the predicate involves symbolic values, most probably it is not possible to determine which branch will be executed during the symbolic execution. Consequently, the only complete approach is to explore all the possible branches. Each branch has a symbolic path condition associated with it. This condition must be true for this particular branch to be executed. This symbolic path condition is called an input assertion since it represents a constraint on the input symbol values if this branch is to be taken.

Clearly, the symbolic execution of any high-level language procedure will fork at each unresolved conditional statement producing many execution paths. It is convenient to think of this process in terms of traversing an execution path tree. This tree can be formed by associating with each program statement a node and with each transition between two statements a directed arc. Nodes representing conditional statements will have more than one arc leaving it, each labelled by the corresponding path condition. The root of the tree is the starting point of execution, and the tree leaves represent the terminal statements. Each node will also have a set of output assertions describing all that is known about the program's variable at that point. To illustrate the above ideas, Fig. 6 shows the execution path tree of the shift register. Clearly, there are five possible terminal paths in the above tree numbered 1–5. The information associated with each path is called the context[4]. In fact, in the symbolic executor, a context actually defines a path. Basically, a context contains the input and output assertions generated to date on the path. A terminal context is the context associated with a terminal execution path. These
contexts are very important because they are directly used to generate the assertion description of the device. The terminal contexts for these five paths are listed in Fig. 7. It is clear that terminal contexts provide detailed information about the device's different modes of operation in the form of assertions description.

The careful reader can quickly discover the resemblance between the assertions of Fig. 7 and the experiments of Fig. 3. Clearly, one can use the same arguments as used in the last section to show the usefulness of the assertion description in test generation.

Example 5. Consider the ISPS description for a hypothetical machine called MINI[4] shown in Fig. 8.

MINI has a fleeting resemblance to a PDP-8 computer. However, there are only a few instructions in MINI, to keep the example manageable. There are six sections in MINI's description. The functions of these sections are clear from their names. Variable declaration is done in the first four sections, while the last two sections describe how the machine operates. An instruction cycle is represented by traversing the loop once in routine START. The instruction register (IR) is loaded from the memory location currently referenced by the program counter (PC); the PC is incremented, and the routine EXEC is called to execute the instruction. MINI has the following instructions:

- **TAD**, addition to the accumulator.
- **DCA**, deposit and clear accumulator.
- **ISZ**, increment memory location and skip if zero.
- **JMP**, unconditional jump.
- **UCL**, a pseudomnemonic which represents a group of instructions defined by how the instruction flags F0, F1, and F2 are set:
  - If F0 is set, AC is incremented;
  - If F1 is set, a skip occurs if AC is currently zero;
  - If F2 is set, the contents of AC and AX are interchanged.

Any combination of flags is legal, resulting in different instructions for each combination. Note that the operation code (OP) has a number of unused values (4–6).

The structure of any computer ISPS description is similar to conventional high-level language structures. The declaration sections define the memory components (registers) used in storing or transmitting information by giving the name of the register (or array of registers) and its dimensions. In addition, multiple names can be given to different parts of the same register—a property called memory overlay[15].

Using high level programming language terminology, the instruction interpretation process represents the main procedure in any computer description, while other processes—such as...
address calculation processes, instruction execution processes, etc.—represent either subprocedures or functions that may be called while executing the main procedure.

The symbolic execution of the ISPS description begins at the instruction interpretation process and follows all possible paths through the description. As explained earlier, the symbolic execution process is performed by a traversal of the execution path tree. This traversal is implemented as a depth-first search.

The execution path tree of the MINI computer is shown in Fig. 9. Each path is defined to terminate when the interpretation loop is about to be restarted, indicating that a new instruction will be fetched and executed. Each complete path from the root to a leaf is called a terminal execution path, or just an execution path. In the MINI's execution path tree, shown in Fig. 9, there are 17 terminal contexts in the path trees numbered 1–17. Each node in the tree represents an ISPS conditional statement (IF or DECODE). Note that several nodes may represent the same statement in the description arrived at by different routes. The input and output assertions generated on a particular path are merged together to formulate the path context. The complete list of terminal contexts for the MINI computer is shown in Fig. 10.

Examining the paths in Fig. 9 or their contexts in Fig. 10 points out an interesting fact: some of the execution paths represent a complete instruction (e.g. paths #1 and #5 represent the TAD and JMP instructions), while other paths represent only part of an instruction (e.g.

**MINI**

**BEGIN**

**Memory State**

M[0:4095] = 11:0

**Processor State**

PC = 11:0,
AC = 11:0,
AX = 11:0

**Internal State**

TMP = 11:0

**Instruction Format**

IR = 11:0,
DP = 2:0,
A = 8:0,
FQ = 2:0,
FI = 2:0

**Execution Processes**

**EXEC**

**BEGIN**

DECODE OP = \{0 **TAD**, 1 **OCA**, 2 **JZ**, 3 **JMP**, 7 **UCL**\}

0 **TAD** := (AC = AC + W(A)),
1 **OCA** := (W(A) = AC NEXT AC = 0),
2 **JZ** := (W(A) = A \& AC = X EXT)
3 **JMP** := (PC = A),
7 **UCL** := (IF FQ = \& PC = PC + 1 NEXT)

**END**

**Interpretation Process**

**START**

**BEGIN**

REPEAT (IR = [PC] NEXT PC = PC + 1 NEXT EXEC)

**END**

Fig. 8. ISPS description for the MINI computer.
paths #3 and #4 together represent the ISZ instruction). The reason behind this fact is the data-dependent branching that is an integral part of an instruction.

From the test generation point of view all the paths are equally important, since each path represents a unique mode of operation for the machine. The terminal contexts list provides the assertion description of all the disjoint modes of operation for the machine under consideration. These assertion descriptions are all that are needed by any test-generation procedure.
To generate a test sequence that will verify the functionality of one of the machine's modes of operation, we have to force the machine into this mode of operation. This is simply done by applying the patterns that will satisfy the input assertions associated with that mode of operation execution path. This information is available in the terminal contexts list already obtained. The second step is to verify that the machine is functioning correctly. This can be done by monitoring any change in the machine state while executing the test patterns and by proving that these changes match the output assertions of that particular mode of operation. Detecting all the changes in the machine state is not an easy task because we are dealing with complicated machines with many internal memory elements that are not easily observable. Also, satisfying some input assertions cannot be done directly, especially for data-dependent assertions, because most of the internal machine state variables are not directly controllable from the outside.

As a result of the above-mentioned problems, different instructions have to be executed to initialize the machine internal memory elements and to satisfy the input assertions. Another set of instructions is also needed to check the contents of all the internal memory elements and prove that the output assertions are achieved. Clearly, testing one mode of operation requires using other modes of operation, either to initialize variables or to read them. Thus, the test-generation process must start testing simple modes of operation, and then use them to test other complicated ones.

It must be noted that all the problems stated above have nothing to do with the technique used in describing the unit under test (UUT), and they are purely test administration problems. It is no longer necessary to automatically generate the test patterns.

It must be noted that the assertion description, obtained by symbolically executing the ISPS procedural description of a machine, has many other applications in addition to its use in automatic test generation. Recent examples of works that do (or could) use some form of assertion description as input are the automatic generation of assemblers, and the automatic generation of compilers[4,15].

4. COMPARATIVE DISCUSSION

In the last two sections we have presented two different solutions to the problem of describing a logic circuit at an implementation-free functional level. These descriptions express the semantics of the device behavior in a direct way that simplifies the test-generation problem.

Symbolic execution provided a link between procedural and nonprocedural descriptions of a digital device, while path-tracing techniques extracted the experiments of a digital device described by a binary decision diagram. Both nonprocedural descriptions (or more precisely assertion descriptions) and the device experiments contain explicitly all the necessary information about the modes of operation of the device under consideration. This information typically contains the input conditions of a certain mode of operation as well as the expected device performance during that mode of operation.

It must be noted that our comparative study between the two approaches does not aim to prefer one over the other. The two approaches are equally important because they address different types of digital devices.

Binary decision diagrams are suitable for describing SSI and MSI classes of circuits where the number of input variables and output functions is small, and the number of experiments is manageable. These diagrams handle only 1 bit variables. Thus, they model the function of the device at the lowest possible level.

On the other hand, procedural description languages can describe almost all the classes of digital circuits. However, they are most suitable for describing computer processors, starting from microprocessors on up to large-scale computers. The basic units of information in these descriptions are vectors of binary digits (registers). Clearly, this level of description is higher than the binary decision diagram's level, and it is called the register transfer (RT) level[2]. Of course, one can use a procedural description language to describe a digital device at the same level of binary decision diagrams by restricting the register lengths to 1 bit only. In this case the execution path tree of such a description will contain exactly the same information as the
binary decision diagrams representing the device. Thus, the symbolic execution of the procedural description approach is more general than the binary decision diagram experiment approach, but it has a lot of overhead expenses (the language interpreter, the symbolic executor, etc.). For this reason binary decision diagrams are more suitable for medium- and small-scale applications, while the symbolic execution technique can only pay off when considering large-scale problems.

5. SUMMARY AND CONCLUSION

The very rapid growth in the complexity of digital functions and systems presents a description problem of increasing severity[16]. In this paper we have described two new approaches to solving this problem.

In the first approach a special type of graph called binary decision diagrams are used to describe digital functions. Using path-tracing techniques we have shown how to generate what is called the diagram set of experiments. In the second approach we described how symbolic execution can be used to provide a link between the functional description of a device, written in a procedural language, and its nonprocedural assertion description.

Both the assertion description and the experiment description provide enough information about the digital device’s different modes of operation. In this paper we have given special attention to the use of this information in simplifying the process of generating tests for the device under consideration. We discussed how such information can be used either directly as test patterns or to aid the test-generation process. Also shown, via an example, was how much fault coverage one can get by relying on these functional descriptions, and the results were promising.

A conclusion that must be extracted from the preceding discussion is that both approaches are successful in providing a complete, concise, implementation-free, detailed description of the different modes of operation of the digital device involved. The symbolic execution approach is best suited for large-scale digital circuits, and especially computer processors. On the other hand, the binary decision diagrams approach is more devoted to smaller digital circuits which have limited functions. Thus, the two approaches can be shown to complement each other.

REFERENCES