Reasoning About Digital Systems†

A.S. Wojcik

Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027

Abstract

The objective of this paper is to explore the uses of automated reasoning in the process of the design and analysis of digital systems. A discussion of reasoning and automated reasoning is presented. Several examples of successful application of this technology are described, and current research is discussed.

I. Introduction

1991 IEEE

The following paper is reprinted from the Proceedings of the 19th International Symposium on Multiple-Valued Logic held in Guangzhou, People's Republic of China in May, 1989. Due to the political unrest at that time, Dr. Wojcik was unable to attend the symposium and present his invited address.

Each of us has been involved in reasoning about digital systems. Whether it be hardware, software, or the integration of the two areas, we assume that we have applied our logical powers to solving problems about digital system design and evaluation. As designers we have also had to address the complexity of design, and have sought ways to assist us in our particular design task. In all of this, we usually take the notion of reasoning for granted.

The objective of this paper is to explore the use of automated reasoning in the design process, where we include both the synthesis and analysis of hardware and software in the definition of design. To do so, we need to have a common understanding of what is meant by automated reasoning and what are its capabilities and limitations.

To understand automated reasoning, we need to define what we mean by reasoning. For the purposes of this paper, our approach is that of Woet, et al. [1] in which reasoning is characterized as the process of drawing conclusions from facts. In order to be certain that the reasoning process is sound, we insist that the conclusions reached must follow from the facts from which they are drawn. Unlike some of the reasoning processes that we might be involved with when we consider system design, we limit the definition of reasoning to be purely logical reasoning, in the classic sense of logical. Common-sense, intuition, fuzzy, or probabilistic reasoning is not included. Clearly, these other types of reasoning are open to further research and can be included in other approaches to automated reasoning.

We view the design process to be one that is based on a clearly defined set of rules, called axioms. These rules reflect the reality of a specific design world that is of interest, and they manipulate the facts that describe the world of interest. If the world can be well-defined and adequately represented in this way, then we can hope to be able to reason about this world. It should be noted that the process of design establishes a model of the world of interest. The effectiveness of automated reasoning or any other approach to Computer Aided Design (CAD) is to a large extent dependent on the accuracy of the model.

Based on this definition of reasoning and this notion of the design process, we consider the objective of automated reasoning to be a way to write and to use programs that assist in the design process. Assistance implies that the programs can help the designer to solve problems and to answer questions that require reasoning (in the way that we have defined it). It does not imply that the programs perform the design process in an independent, self-contained manner.

Given this definition of automated reasoning, it must be asked why should such a process be considered for use in the system design?

As has been described elsewhere [2], with the increasing complexity of systems, it has become imperative to approach design in a structured, hierarchical fashion. In software parlance, we use (to a great extent) a top-down design process. The highest level in the process would be a system level description, such as "design a 32-bit CPU using CMOS gate arrays," or "design an arbiter to handle requests for resources from five independent sources." While the latter specification may seem to be more concrete, each describes a system-design objective. Such high-level descriptions are subsequently refined at lower levels of the hierarchy until at the lowest level, we have, for example, the masks for the silicon chips. The design process requires appropriate tools at each level of the hierarchy.

We can focus on moving from one level i in the hierarchy to the next lower level i + 1. The synthesis problem requires the design to be more fully developed from the level of abstraction represented at i. For example, proceeding from a register transfer description of a system to a gate level design is a common step in the design process. But each of these two levels contains a description which is an abstraction of the intended design objective. Analysis is the problem of verifying that the implementation at level i + 1 satisfies precisely the same design goals that have been implemented at level i. If a CAD tool allows the designer to intervene in the synthesis process from i to i + 1, then the level i + 1 implementation may contain features that could not be derived from the CAD tools themselves. Further, the intervention might have had a global effect that is not apparent.

Automated reasoning can be helpful in synthesis at some steps in the design hierarchy. It may help to better represent the design constraints, to decide among many possible options, to represent timing requirements and synchronization issues. In analysis, the current predominant tool is simulation. However, automated reasoning may become absolutely essential, since simulation is not a completely satisfactory technique. To paraphrase Dijkstra, simulation can show the presence of errors but can never completely demonstrate their complete absence.

II. Automated Reasoning

Given this definition of reasoning and the basic concepts underlying automated reasoning, we must consider the way in which automated reasoning programs accomplish their task of reasoning.

It is assumed that the reasoning programs will incorporate inference rules. These rules assist in deriving new facts about the world of interest from given facts. The inference rules used are those that most closely mimic logical reasoning.

We have used the systems AURA (AUtomated Reasoning Assistant) and ITP (Interactive Theorem Prover) [3] developed at Argonne National Laboratory.  

†This work was supported in part by Argonne National Laboratory.
National Laboratory. In particular, ITP is a system written in Pascal that has been ported to a wide variety of computing systems. It incorporates a broad spectrum of inference rules that allow the researcher to experiment with a variety of approaches to analysis and synthesis. In [1], a detailed explanation of automated reasoning is given along with further insights into the use of ITP.

One important point about ITP is that it has a language, clausal form, associated with it. This language allows the manipulation of the basic constructs, clauses, with which the axioms and facts are represented. Further, the inference rules that are designed into the ITP system operate on the clauses.

Using such a system for a particular problem world requires considerable time to describe a model of the world, to establish strategies to deduce facts, and to prove assertions about the world. It is hoped that a reasoning approach will result in a more inclusive consideration of the world being modeled and will eliminate unproductive analyses of the world more rapidly than other approaches. However, a major limitation to using such a system is the size and complexity of the world being analyzed. It is essential that a hierarchical approach be used in examining the world model which will reduce complexity by partitioning the problem into computationally feasible tasks. The process of partitioning is quite important, and the burden is on the user to be sure that the partitioned model continues to accurately reflect the world of interest.

It should be noted and emphasized that automated reasoning is an assistant that can be used only in appropriate parts of certain problem domains. It is not a panacea for solving all problems or even all problems in one problem domain [4]. Further, by adopting an automated reasoning approach, we do not preclude that a reasoning approach will result in the use of non-reasoning-based approaches. It is expected that a reasoning approach may be replaced by algorithms or other procedures implemented in other languages which may use totally different paradigms for problem solution.

III. Synthesis

The basic philosophy for synthesis using automated reasoning is to begin by representing the elements used in the design by a set of axioms and a collection of facts. Then the goal or objective, i.e., design a module based on the given facts and axioms, is stated as a realizability theorem. The proof generated by the system of the realizability theorem yields the synthesized module.

The work described in [5,6] on the synthesis of multi-valued logic functions is seminal in this area. This work established the feasibility of performing synthesis at the circuit, gate and module levels using automated reasoning. What this work demonstrated was the importance of stating the synthesis problem as a theorem. It also carefully delineated the limitations of this approach to synthesis. In principle, automated reasoning will generate a proof (i.e., a design) given enough time and memory provided that it is dealing with a theorem. It cannot tell us that something either is or is not a theorem (since we cannot solve the Halting Problem). Further, even if the system is dealing with a theorem, there is the possibility that a proof is not generated. For instance, the set of facts may be incomplete, or the inference rules used may be insufficient to actually produce the desired result.

Another important consideration is that the proof generated may be an answer (i.e., design) to a question that is the wrong question! That is, if the specification of the module that was to be synthesized was not itself accurate, the design that is produced will correctly implement an unwanted module. The reasoning system cannot determine that it was provided with a logically correct, but unneeded, specification.

The work on synthesis lead to the consideration of the importance of having a language that could adequately represent the world of interest. The clause language was satisfactory from the technical perspective of providing a means to deduce facts. However, it became apparent that to facilitate synthesis and analysis, a representation language that simplified the modeling of the specification of system behavior was needed.

IV. Analysis

Analysis is commonly thought of as the inverse to synthesis. Usually, analysis is also called verification. The goal of verification is to demonstrate the equivalence between a design and a specification. As previously indicated, while simulation is the de facto standard for verification, it is an imperfect standard that needs assistance from other possible approaches.

Formal verification has the goal of establishing a mathematical proof that a given property or behavior is realized by an implementation. In terms of the design hierarchy, the problem is to deduce that a desired behavior at level \( i + 1 \) is the same as that which was specified at level \( i \). It can be seen that in verification, a language for representation is very important.

The area of formal verification has received a considerable amount of attention [7-9]. In some early work [8,9], we considered the use of a simple register file transfer language to facilitate the representation of register and gate-level hardware designs. While useful, such a language was limited in its applicability to the design hierarchy, and it was not very useful for representing software. Several languages have been reported in the literature. We determined that a modified form of Petri nets [10], called flow nets, provided a robust technique to represent hardware, software, and their interaction at all levels of the design hierarchy. In addition, the flow nets mapped very simply into the clausal form language of ITP.

Flow Nets

Theoretically, a Petri net is an abstract machine that can be made equivalent in computational power to Turing Machines. A Petri net is a directed, bipartite graph. There are two types of nodes in these graphs: Places and Transitions, with Places, drawn as circles, representing conditions, and Transitions, drawn as bars, representing events (see Figure 1). Tokens, represented as black dots within Places, symbolize the state of the net; their number and position change as a result of net execution. The execution of a net entails the firing of Transitions. If all input places to a transition contain tokens, the transition is said to be enabled. Once enabled, an event occurs, namely, the transition fires. This implies that tokens are removed from the input places, and a token is deposited in each of the output places.

To this basic structure, we have introduced several extensions to define Flow nets [4,11]. These extensions do not increase the computational power of Petri nets, but rather make them more usable in the required representation of the world of hardware and software. Symbolic tokens are used. Hence, a token has a symbolic value associated with it. For example, the symbolic value could be a ternary logic value, it might be a Boolean expression, or a list of assumptions that must be true for certain behavior to be valid. Each Transition and Place is given a user-defined type. Tokens are transformed during the execution of a net by the transitions, with the transformation defined by the type associated with the transition. In Figure 1, the transition type is Transfer. Transitions describe functional activity at an arbitrary level of abstraction dependent on the level of the design hierarchy that is being considered. Further examples of transition types include T-gate, ALU, if-then-else, and bubble-sort. A third extension to Petri nets is allowing places to have the semantics of a read-only-memory. This implies that the resident token can be accessed but not removed during net execution. Finally, we require a control signal to be defined for every transition (the dotted line in Figure 1). The control signals define an interval of time during which an enabled transition may fire while they also activate a particular function associated with a multifunctional transition.

Flow nets provide a means to model a system of interest, and also allow a simple mapping into clausal form representation. Figure 3 shows a gate level model of a typical 2-bit ALU (74LS181). The Flow net representation is also given. In this figure, transition and place
types represent logic gates and wires. Also shown is part of the clausal
The templates for Transition and Place are self-explanatory. Conn_In
and Conn_Out describe connections between places and transitions.
The label sport_name specifies an input or output port of a transition.
The Function template allows the specification of a transition type.

The notation \( P(i) \) specifies Port i of a transition with \( i \) being
the lowest numbered port. In Figure 3, the clause \( \text{Function}(\text{A}, \text{P}_{3}, \text{I}, \text{AND}(\text{P}(1), \text{P}(2))) \); indicates that the Transition labelled A4 when given
the control signal value 1 will produce at the Place labelled P3 the
symbolic token \( \text{AND}(\text{P}(1), \text{P}(2)) \), with the tokens at ports \( \text{P}(1) \) and \( \text{P}(2) \)
substituted into the string.

**Reasoning**

For this simple example, the reasoning program will contain a
description of the ALU along with axioms about Boolean algebra.
Other axioms will provide for the firing of the Flow net. Symbolic
tokens that are generated will be manipulated by the axioms to both
simplify and canonicalize them. Verification asks if the set of func-
tions performed by the modelled ALU are the same as those specified
by the designer of the ALU.

One can surmise that this level of verification requires Boolean
comparison between the functional behavior as represented by the
symbolic tokens generated by the execution of the Flow net and a
predetermined description of the desired behavior of the module.
Boolean comparison is a computationally complex problem. We have
developed an algorithm for comparison which is implemented in C
[12]. As an example of the performance of this algorithm, the
verification of all 16 functions of a 1-bit ALU required 1.9 seconds
on a Sun 3/280. Using this tool outside the domain of the ITP program is
an example of our philosophy to use appropriate tools for a given task.

It should be noted that the automated reasoning approach illus-
trated by this example shows that we can demonstrate that each step of
our process is provably correct. Each step of the net execution is the
result of applying a mathematically correct inference rule to a collec-
tion of facts. Hence, the symbolic token that is generated is a theorem.

V. Other Applications of Automated Reasoning

The complete generality of the Flow net structure coupled with
the modelling capability provided by the use of symbolic tokens and
the typing of transitions and places makes these nets useful in a variety of
applications. In [13], some preliminary work on the verification of
software is presented. In this application, tokens are used to represent
the program state and also to carry verification conditions. While the
previous section discussed verification of Boolean functionality, there
are other types of behavior that can be considered. In [11], the
problem of verifying the property of fault tolerance in the Draper Labora-
tories' Fault Tolerant Processor (FTP) is discussed. This application
shows that the modelling and analysis procedure that we have
developed can be successfully adapted to a very different kind of
behavior. In this case, fault tolerance verification requires a formal
proof that the Draper FTP will correctly execute a set of data distribu-
tion instructions in the presence of a single failure anywhere in the
FTP system. A failure could be anything from a gate malfunctioning
to an entire CPU crashing. The FTP system is used to reason about the
nature of the interconnection of various hardware units that comprise
the FTP and to deduce conditions which must be true in order for a
data distribution instruction to properly execute in a faulted en-
vironment.

Most recently, we are adopting temporal logic [14] into our
verification approach. The objective is to be able to reason about timing
considerations in a system. Included in such an analysis will be the
verification that a hardware device does not have any hazard or race
conditions. In this application, tokens carry information about timing
conditions. These conditions are represented by logic expressions that
contain both Boolean and temporal operators. Axioms have been
developed that allow the manipulation of these hybrid expressions. It
is anticipated that the algorithm that we have developed for Boolean
comparison will be extended to allow the comparison of temporal
expressions.

VI. Conclusions

We have attempted to define reasoning and, in particular,
automated reasoning. It is important to understand that other
definitions of reasoning are possible, and that other approaches to
automated reasoning can be tried. Other notions of reasoning have not
been included since the current status of ITP, and its successor, do not
provide for their inclusion.

The use of automated reasoning in the process of design and
analysis has been sketched. Again, we must emphasize that automated
reasoning is not a panacea for all problems. There are suitable niches
in the design hierarchy where the reasoning approach is effective.
Finally, the approach to verification using reasoning has been
described. While other approaches to verification are possible, the
Flow net modelling capability coupled with the use of reasoning makes
this approach very viable for a number of hardware and software
problems. Based on the successful results that have been obtained and on
current work, it is felt that automated reasoning can prove to be a valu-
able addition to computer aided design.

References


ITP*, Mathematics and Computer Science Division Report, Argonne

tation, Department of Computer Science, Illinois Institute of Tech-

multiple-valued logic circuits by automated theorem-proving tech-
September 1983.

rial logic using theorem-proving techniques," *IEEE Transactions on

correctness: introduction and survey of current research," *IEEE Com-


system based on an automated reasoning system," *Proceedings of the 21st Design Automation Conference*, Albuquerque, New Mexico,


of fault tolerance using theorem-proving techniques," *IEEE Transac-


---

**Fig. 1. Transfer Transition Type with Control Signal Place**

```
In Data

Control

Transfer

Out
```

**Fig. 2. Clause Templates for the ITP System**

```
Transition(transfer_name, xtype, xcontrol);
Place(xplace_name, xtoken, xtype);
Conn_In(transfer_name, sport_name, xplace_name, scontrol);
Conn_Out(transfer_name, sport_name, scontrol, xplace_name);
Function(transfer_name, soutput_place, scontrol, xfunction);
```
Place(A1, Arb, Wire);
Place(B1, Arb, Wire);
Place(P1, Arb, Wire);
Place(P2, Arb, Wire);

Function(A1, P1, 0, 0);
Function(A1, P1, 1, NOT(P1));
Function(A1, P2, 0, 0);
Function(A1, P2, 0, P(1));

Transition(N1, T1, cnlt(1));
Transition(A1, T1, cnlt(1));
Transition(A2, T1, cnlt(30));
Transition(A3, T1, cnlt(31));
Transition(A4, T2, cnlt(32));

Conn_In(A1, A1, P(1), cnlt(1));
Conn_In(B1, N1, P(1), cnlt(1));

Conn_Out(E1, P(3), cnlt(1), O3);

Function(N1, P1, 0, 0);
Function(N1, P1, 1, NOT(P1));
Function(A1, P2, 0, 0);
Function(A1, P2, 0, P(1));

Function(A4, P3, 0, 0);
Function(A4, P3, 1, AND(P1, P(2)));
Function(A5, P6, 0, 0);
Function(A5, P6, 1, AND(P1, P(2)));

Function(E1, O3, 0, 0);
Function(E1, O3, 1, EOR(P1, P(2)));

Fig. 3. Low-level Flow Net Description of 2-bit ALU