

# Automatic Synthesis of SARA Design Models From System Requirements

Kar-Wing Edward Lor and Daniel M. Berry

**Abstract**—This paper describes research which has introduced partial automation to a requirement-driven design process. System design is a creative activity which requires a lot of human judgment. Yet it is possible for the machine to assist in the process. An interactive tool is built to assist, but not replace, the human designer in building a system based on the requirements. System ARchitect's Apprentice (SARA) is an environment-supported method for designing hardware and software systems. Starting from the requirements, a design is made in the form of the system's structural and behavioral models. So far, in the design environment that supports this method, only informal requirements have been involved and automation is rarely employed in the design process. The human designer is completely responsible for all decisions, details, and the correctness of the products. An automatic design synthesizer has been built to bridge this gap between the requirements and the design. Two models are employed to represent two facets of system requirements. The operational semantics of all constructs in the requirement models are defined. Automatic synthesis of design structures in the SARA domain is based on these operational definitions. The goal of the tool is to ease the task of system design within the SARA method.

**Index Terms**—Computer-aided software engineering, design automation, design synthesis, modeling of concurrent systems, system design methodology, requirement specifications.

## I. INTRODUCTION

A common thread running through most development methods and life-cycle models for real time systems is the importance of stating requirements and producing a design long before implementation begins. The requirements are obtained in discussions with the client, and a design is produced to demonstrate the architecture of a system to meet the requirements. The actual implementation of the system is done as a refinement of the design.

The importance of getting the requirements and the design on paper is recognized. Doing so forces more careful consideration and permits review by others with the aim of elimination of errors and verification that what is desired is expressed.

There are many languages, textual or graphical, for stating requirements; some of them are more formal than others. These languages include the Problem Statement Language (PSL) [25], the Requirement Statement Language (RSL) [2], Data Flow diagrams (DFD) [7], System Verification Diagrams

(SVD) [12], and even plain English, as well as its structured subsets [1]. There are many established methods for producing such requirements, including the PSL/PSA (Problem Statement Analyzer) Technique, System Requirement Engineering Methodology (SREM) for RSL, Structured Analysis (SA) for DFD's, System Specification Verification Methodology (SSVM) for SVD's [4], etc. Supporting these methods, there are tools aiding the preparation, analysis, and validation of such requirements, such as the Problem Statement Analyzer for the PSL/PSA technique, Requirement Engineering Validation System for SREM, Computer Science Corporation's in-house tools for SSVM, and numerous commercial packages for Structured Analysis.

At the next stage of the system life cycle, there are also many methods to produce a design. Established methods include the System ARchitect's Apprentice (SARA) [10], the Distributed Computing Design System (DCDS) [3], an extension of SREM, the Advanced Design AutoMation System (ADAM) [13], etc. Each of these methods has its own language or model to express the design, such as SARA's Structural Model and Graph Model of Behavior, DCDS's Distributed Design Language, or ADAM's Design Data Structures. Furthermore, there are also languages used specifically to express software or hardware, like various Program Design Languages [5], [6] for software design, and a hierarchy of hardware description languages for VLSI design, as used in CMU's Design Automation (CMU/DA) environment [9]. Among all these design methods, there are tools supporting one or more of the following activities: editing, analysis, correctness verification, simulation, performance measurement, prototyping, and implementation synthesis. These activities generally distinguish the design phase from the requirement phase in the development life cycle.

The subject of the research reported in this paper is the process of obtaining designs from already given requirements. In other words, given a requirement document with the requirements stated in some appropriate languages, how does one obtain a document expressing a useful design for a system? This design has to meet the requirements and permit other related activities in the design phase. Heretofore this process has been left largely to a human being—the programmer or system designer. While some requirements handling tools assist in the preparation of the requirements, none of them addresses the problem of producing a design from the requirements. PSA, for example, goes no further than assisting in the analysis and validation of the PSL-expressed requirements. While many design tools do provide

Manuscript received September 23, 1988; revised July 24, 1991. Recommended by S. S. Yau. This work was performed at the University of California, Los Angeles, and supported by Grants from Unisys, NCR, and the State of California MICRO program.

K.-W. E. Lor is with AT&T Bell Laboratories, Middletown, NJ 07748.

D. M. Berry is with the Department of Computer Science, Technion, Haifa, Israel 32000.

IEEE Log Number 9104157.

|                                                                                                                                                                                                                                                |                                                                                                          |                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>STIMULI —</b><br/>One or more external system or sub-system entities from any source, manual or automated, that invoke this component, e.g. an operator request or command, a system message, an external file, a system state, etc.</p> | <p><b>LABEL —</b><br/>Unique numerical identifier assigned to each decomposition element in the SVD.</p> | <p><b>RESPONSES —</b><br/>One or more entities produced by this component, e.g. a switch of system state, an action, a message, an invoice, etc.</p> |
| <p><b>NAME —</b><br/>A descriptive name for the decomposition element. This block should identify the process; it should not describe those process components that modify or constrain the process.</p>                                       |                                                                                                          |                                                                                                                                                      |

Fig. 1. Layout of a decomposition element.

methodological assistance to the design process, none do so for requirements stated in an established but yet unrelated requirement language. For example, the SARA environment has a formal way of stating design and purports to support requirement-driven design, but in fact has no formal way of stating the requirements and of bridging the gap between the requirements and design.

A knowledge-based system, called the Design Assistant, was built to help the system designer to transform requirements stated in one particular collection of requirements language into a design stated in one particular collection of design languages. As a front-end to the SARA method, the requirements prepared are expressed in Data Flow Diagrams and System Verification Diagrams. The design languages are SARA's Structural Model and Graph Model of Behavior. This particular choice of design languages was dictated by the fact that the tool was built on top of the SARA platform. The Design Assistant synthesizes the Structural Model and the Behavioral Data Model from the Data Flow Diagrams, and the Behavioral Control Model from the System Verification Diagrams. Due to space limitations, this paper addresses only the synthesis of the Behavioral Control Model, the focus of this research.

This paper first reviews the SVD requirement specification features, and then the SARA design models. While there are certainly many other such requirement and design languages, space limitations prevent talking about any more than those that are used in the Design Assistant and in the example of the paper. Then the paper describes a knowledge-based tool for synthesizing a particular domain of SARA design from the requirements, and an example is given to illustrate this synthesis process. This example shows the rules used and how they are applied. The paper concludes with an evaluation of the approach.

## II. SYSTEM VERIFICATION DIAGRAMS AS SYSTEM REQUIREMENTS

In this research in design automation, two views are employed as the requirements of a system—namely, the *functional requirements* and the *operations concept*. A requirement analyst uses Data Flow Diagrams and System Verification Diagrams to represent the functional requirements and the operations concept, respectively, as suggested by a multiple-

view requirement validation method [8]. Since the facet of the design synthesis addressed in this paper is based on the System Verification Diagrams, this section focuses on this particular requirement model.

The operations concept of a system is expressed in the System Verification Diagrams (SVD), based on an underlying graph-based model, called the Stimulus-and-Response Model. Its primary purpose is to demonstrate a static event-dependency relationship among systems and subsystems. In addition, requirements in the form of an SVD are also indicative and structured enough to derive skeletons of the design models.

An SVD is simply a directed graph in which each node corresponds to a system/subsystem requirement specification. A node, called a decomposition element (DE), is considered a functional black box which takes external stimuli and produces responses. The layout of a DE, plus the description of each of its entities, is shown in Fig. 1.

The actual function of a DE is defined elsewhere. Each DE is associated with a primitive process in the Data Flow Model, the other requirement model used in this research. Each primitive data-flow process contains text revealing its actual input-to-output transformations. However, such internal functionality is out of the scope of the design domains addressed in this paper.

In our work, we categorize the possible stimuli and responses in the model for the sake of formal definitions. Each response produced will eventually become part of one, one, or more conditional or unconditional stimuli of other DE's. Each stimulus, in turn, may come from none, part of one, one, or several responses produced somewhere else. They are described in Table I.

Since the SVD is a directed graph connecting all DE's, this directed graph indicates the logical relations among the DE's or system/subsystem requirements. A logical relation indicates sequencing, competition, and/or sharing upon arrival of a stimulus; namely, a *common stimulus*. There are five permissible logical relations in the model, as presented in Table II.

Fig. 2 illustrates a sample SVD for a recording module of an aircraft monitoring system. This is an example of the SEQUENTIAL-EXCLUSIVE-OR relation. The common stimulus is the state *recording in progress*, or its complement,

TABLE I  
STIMULUS AND RESPONSE CLASSIFICATIONS

| Stimuli or Responses | Descriptions                                                                                                                            |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Physical stimulus    | An object such as a message, an initiation signal, a set of data, etc.                                                                  |
| Stimulus condition   | A particular condition imposed on a physical stimulus                                                                                   |
| Synchronous stimulus | An initiation signal at a specified time interval                                                                                       |
| State stimulus       | The continuous truth of a condition on the system, or one or more of its subsystems                                                     |
| Disjunctive stimulus | The logical disjunction of two or more stimuli                                                                                          |
| Stimulus sequence    | An ordered sequence of two or more stimuli                                                                                              |
| Physical response    | An object such as a message string, an initiation signal, a set of data, etc.                                                           |
| State response       | A change of the state of the system or its subsystems                                                                                   |
| Action response      | An initiation of an action that produces a physical response, or changes a system state                                                 |
| Alternative response | Consists of a condition and two responses; if the condition is satisfied, it produces the first response, otherwise the second response |
| Response sequence    | An ordered sequence of two or more responses                                                                                            |

*recording not in progress*. The truth or falsity of this condition is used to decide which of the three DE's to invoke. This example also consists of a physical stimulus, a state setting/clearing response, and an action response.

### III. THE SARA DESIGN METHOD

The SARA design method is a requirement-driven, top-down design method; there exists a graphics-oriented design environment in which such design is carried out. At the topmost level are the requirements. Each subsequent level is a design, each being a refinement of the requirements or design of the level above. The bottommost level is an ultimate realization of the requirements. If the requirements have in fact been dealt with properly level-by-level, then the realization correctly meets the requirements. At each design level the system at that level is described using the two main formal models of SARA, the Structural Model (SM) and the Graph Model of Behavior (GMB).

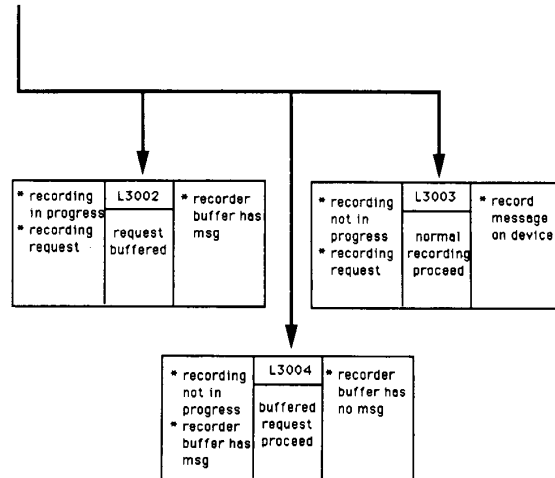


Fig. 2. A sample multidestination relation with a state common stimulus.

Only those aspects of the models needed to carry out the example synthesis are discussed herein. Complete details on the SARA method and environment are presented in [10].

#### A. The Structural Model (SM)

The SM represents the structure of a system. The model has three primitives: *module*, *socket*, and *interconnection*. A module represents a system component. Hierarchical decomposition is achieved by refining a module into submodules. A module is connected to another module by an interconnection bridging two sockets, the modules' communication ports.

In a typical design, there is a top-level module called **universe** with no socket. The initial decomposition is to refine it into two submodules, the desired **system** and its external **environment**. These two submodules communicate via one or more interconnections. This decomposition process repeats until the system and its environment are divided into modules small enough, with the behavior of each one precise enough to be modeled by a single GMB.

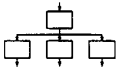
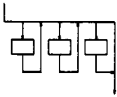
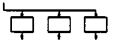
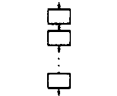
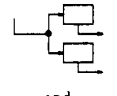
#### B. The Graph Model of Behavior (GMB)

The GMB [22] models three different but related aspects of the system: control, data, and interpretation. These three domains are defined independently, but the primitives within them must be consistent with each other.

The GMB control domain describes concurrency, synchronization, and precedence relations in a graph using an underlying theoretical model similar to that of Petri Nets [20]. The control graph is a directed hypergraph; i.e., a graph in which the edges may have one or more sources and one or more destinations. A *control node* represents an event, and a *control arc* represents precedence constraints among two or more events using the token passing mechanism.

The GMB data domain reveals the system's data storage units, their values at various system states, and their access rights by the data processing units. The data graph is a bipartite

TABLE II  
LOGICAL RELATIONS IN STIMULUS AND RESPONSE MODEL

| Relations                                                                                                        | Descriptions                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <p>exclusive-or</p>            | This relation is single-predecessor, multi-successor. Only one of the successor events will occur.                                                                                                                                                                                                                                                                                            |
|  <p>sequential-inclusive-or</p> | This relation is multi-successor. The stimulus conditions of the first successor DE are checked. If they are satisfied, then functional processing of the DE is performed. Regardless of the result, it still checks the stimulus conditions of the remaining successor DEs, and carries out the functions of the ones satisfied. As a result, any combination of successor events may occur. |
|  <p>sequential-exclusive-or</p> | This relation is multi-successor. The stimulus conditions of the first successor DE are checked. If they are satisfied, the event will occur. Otherwise the second successor's stimulus conditions are checked, and so on. In other words, only the first satisfied successor will occur.                                                                                                     |
|  <p>sequence</p>                | This relation is single-predecessor, single-successor. The predecessor must occur before the successor.                                                                                                                                                                                                                                                                                       |
|  <p>and</p>                     | This relation is multi-successor. The successors represent the start of parallel processes, all of which will occur, provided all of their stimuli are there.                                                                                                                                                                                                                                 |

directed graph—i.e., a graph with two kinds of nodes, *datasets* and *data processors*—and each arc, called a *data arc*, connects a data processor to an accessible dataset. Each data processor is mapped to at least one control node in the control domain.

The Interpretation Domain defines simulation timing control, data transformations of the data processor, and control decisions of the associated control node. Any programming language may be used for this domain. The current SARA implementation, being coded in a statically scoped LISP called T [23], uses T as the interpretation language as well. Attached to a data processor, the interpretation code is executed whenever the processor is invoked.

All the control and data domain primitives, as well as how the token machine operates on them, are described in more detail in Table III.

#### IV. RULE-BASED DESIGN SYNTHESIS

This section addresses various issues in automatic design synthesis. Previous work on intelligent design tool includes Kowalski's Design Automation Assistant (DAA) [15], built on top of the CMU/DA environment, and Knapp's Design Planning Engine (DPE) [14], built on top of the ADAM environment. DAA addresses automatic synthesis only in VLSI, while DPE is an intelligent tool aiding the evolution of a system represented in different phases of the Design Data Structures. In our research, we build a tool that synthesizes general hardware/software design from requirements prepared from methods which are well-known but unrelated to the design method itself.

Given two views of system requirements, the design synthesizer helps to produce three views of the system design: structural model, behavioral control model, and behavioral data model of the system. This section describes the general synthesis approach, as well as the synthesis of the behavioral control domain in detail. Since the process is not fully automatic, human input is needed in various stages of a design session. This section also addresses the role of the human designer.

##### A. The Approach of Synthesis

The design synthesis process is regulated by a collection of design rules, representing the knowledge of the SARA design method. A rule in this system is in the form of

$$\langle \textit{antecedent} \rangle \Rightarrow \langle \textit{consequence} \rangle$$

where the  $\langle \textit{antecedent} \rangle$  checks whether a requirement construct satisfies certain conditions, and the  $\langle \textit{consequence} \rangle$  represents the design actions to take place in that case.

The human designer picks a portion of the requirements to start the synthesis. The portion selected constitutes a primary goal, in the form of

**synthesize design from  $\langle \textit{selected requirements} \rangle$**

to be fed to a rule interpreter. The interpreter, employing a forward-chaining scheme, then tries the rules on the primary goal. Upon satisfaction of the antecedent, the consequence taken is either:

- a sequence of actions which create primitives in the SARA domain,

- breaking up of the primary goal into subgoals, each of which is responsible for synthesizing design objects from a subcomponent of the originally selected component, or
- a combination of both.

The rule-based approach was selected for the synthesis because of two reasons. First, the current set of synthesis rules is by no means complete. It would thus be dangerous to lock the current set of rules into a traditional imperative program with each rule expressed explicitly as a sequence of tests. There is always room to enhance the design knowledge by including additional rules about domain specific knowledge, design alternatives, and optimization decisions. A rule-based system is more extendable in that new rules may be added at will, *without having to change the rule interpretation engine*. Second, given just a synthesized product, the human designer may question how a particular design decision is derived, or specifically, why certain design constructs are generated. With a rule-based system, the sequence of rule firings serves as a natural explanation of the decisions during the automated design process.

### B. Control Domain Synthesis

This section addresses the primary facet of the synthesis process, the behavioral control domain. In particular, it describes the knowledge of synthesizing control node sequences according to selected constructs in the stimulus-and-response model. In case the control domain alone is not sufficient to model a construct, the data and interpretation domains are used to supplement the modeling.

Building the control domain according to the System Verification Diagrams requires in-depth knowledge of the GMB control domain and semantics of the constructs in the stimulus-and-response model, as well as the connection between the two models. The very first rule applied in a session is one that takes an SVD and creates subgoals for its components:

#### SVD.1

*Antecedent:* any SVD

*Consequence:* Subgoal: synthesize control domain objects for initial relations of SVD

Beginning with an SVD, the two essential tasks in deriving the control graphs are:

- transform the event dependency information, i.e., the relationships among the DE's, into control node sequences, and
- generate control node sequences from various stimuli and responses of a DE, and connect them to the sequences generated above.

When building the synthesizer, the most crucial step is to establish the formal definitions, represented by the SARA models, for all requirement constructs. These definitions are considered operational, in the sense that an underlying token machine governs the actual semantics. In addition, a design process requires certain bookkeeping and optimization decisions, as well as considering design alternatives. The following

$((state.signal \text{ or } notstate.signal) * (state \text{ or } notstate)) \text{ and } A1 > A1$

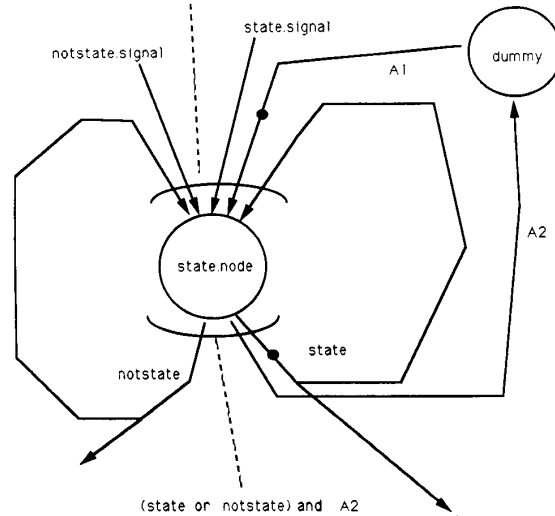


Fig. 3. GMB primitives to model state stimulus.

subsections present the synthesis rules of several sample requirement constructs based on these operational definitions and bookkeeping.

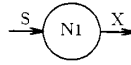
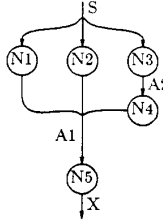
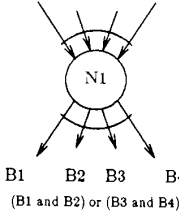
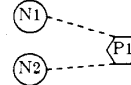

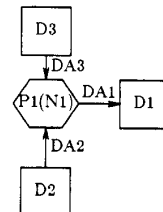
1) *Synthesis Rules for Stimuli:* In the System Verification Diagram, each stimulus only represents an informal concept—what invokes the decomposition element. Each conceptual entity must have its semantics formally defined before any GMB synthesis can be carried out. The formal definitions of two sample stimuli are presented in this section.

a) *State Stimulus:* A state stimulus, representing the continuous truth or falsity of a system condition, is modeled by a control node sequence  $S$ . If a DE has the state as its stimulus, a control arc from  $S$  should be ready to invoke the control node representing the DE all of the time. In other words, an event-invoking-arc corresponding to the state should always have one token on it. With less than one token, it cannot invoke the event when it should. With more than one token, it may invoke the event when it is not supposed to. The node sequence  $S$  should also be able to switch from state to its complement state upon arrival of a state-switching signal. Fig. 3 shows an illustration of  $S$ . Here, the node **dummy** controls the synchronization. With its associated interpretation code specifying a delay of 0 time unit, it invokes **state.node**, the main node of  $S$ , at every simulation time unit to ensure steady deposit of one token on the arc **state** or **notstate**. The arcs **state** and **notstate** head toward **state.node** to ensure that the token on either of them is always lifted, preventing the tokens from *piling up*.

A case analysis of the interpretation of **state.node** is given as follows:

- in the case state-switching arc, **state.signal** is among the triggering arcs, a token is deposited on **state**, and the status **state** is recorded in the interpretation domain
- in the case state-switching-arc, **notstate.signal** is among

TABLE III  
DESCRIPTIONS OF GMB PRIMITIVES

| TYPE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | GRAPHICAL                                                                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>A named <i>control node</i> represents a step in a process being modeled. A controlled data processor (see below) may be associated with a node to provide interpretation of the process.</p> <p><u>Example:</u> A node N1 has a single entry arc S and a single exit arc X.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                    |
| <p>A named directed <i>control arc</i> represents non-volatile precedence relations between sets of nodes. If there are more than one source or destination nodes the arc is called <i>complex</i>; otherwise it is called <i>simple</i>. An enabling token is placed on an arc either as a starting state or upon termination of any of its source nodes. When a node is initiated, its enabling tokens are absorbed.</p> <p><u>Example:</u> A2 and X are simple control arcs. A1 is a complex control arc whose source set consists of nodes N1, N2 and N4, and whose sole destination is N5. S is an incoming complex control arc whose destination set consists of N1, N2 and N3. If there were an initial token on S, the token machine would non-deterministically invoke N1, N2 or N3.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                    |
| <p style="text-align: center;"><i>Input Control Logic</i></p> <p>A logical relation among the input arcs to a node specifies a precedence condition on the token states for the node to be initiated. An <i>OR</i>, <i>&gt;</i>, or <i>+</i> in the input logic means that a token on any of the operand arcs may initiate the node; an <i>AND</i> input logic means that all operand arcs must have a token to initiate the node. One or more tokens from the triggering arcs which satisfy the input logic are absorbed for the node initiation. For the <i>OR</i> logic, a token is absorbed non-deterministically from one of the triggering arcs; for the <i>&gt;</i> logic, a token is absorbed from the first triggering arc in the logic; for the <i>+</i> logic, one token is absorbed from each of the triggering arcs; and for the <i>AND</i> logic, one token is absorbed from each arc in the logic.</p> <p><u>Example:</u> If A1, A2, or both A3 and A4 have tokens, then N1 can be initiated. Tokens are lifted according to this precedence — A1 and A2, either A1 or A2, A3 and A4.</p> <p style="text-align: center;"><i>Output Control Logic</i></p> <p>A logical relation among the output arcs specifies which arcs have tokens placed upon them when a control node is terminated. When an <i>OR</i> output relation holds, a data processor interpretation must decide which one or more arcs receive tokens. When an <i>AND</i> relation holds, all output arcs receive tokens.</p> <p><u>Example:</u> When N1 is terminated, its associated controlled data processor will decide whether tokens are to be placed on B1 and B2, or B3 and B4.</p> | <p style="text-align: center;"><math>(A1 + A2) &gt; (A3 \text{ and } A4)</math></p> <p style="text-align: center;">A1    A2    A3    A4</p>  <p style="text-align: center;">B1    B2    B3    B4<br/>(B1 and B2) or (B3 and B4)</p> |
| <p>A named <i>controlled data processor</i> represents a data transformation object which is activated when an associated control node is initiated. An interpretation of the data transformation and other parameters such as time delay or resource requirements can be associated with the data processor.</p> <p><u>Example:</u> Processor P1 is initiated whenever either N1 or N2 is initiated. The control graph carries the burden of guaranteeing that N1 and N2 are enabled in a desired sequence. Otherwise they will be activated in a non-deterministic order, and the simulator will show possible contention.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                  |
| <p>A named <i>dataset</i> represents a passive collection of data.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                  |
| <p>A named <i>data arc</i> statically binds one or more data processors to a dataset. The arc indicates the access rights and mechanism to the data. The access mechanisms include non-destructive read (<i>R</i>), simple write (<i>W</i>), destructive read (<i>DR</i>), first-come-first-serve read (<i>FCFSR</i>), first-come-first-serve write (<i>FCFSW</i>), last-come-first-serve read (<i>LCFSR</i>), and last-come-first-serve write (<i>LCFSW</i>).</p> <p><u>Example:</u> Processor P1 is initiated by control node N1. P1 reads data from datasets D2 and D3, via arcs DA2 and DA3, respectively, and writes the result into dataset D1 through arc DA1.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                  |

- stim.12  
*Antecedent:* **stim** is a state stimulus  
 $\wedge$ no control node sequence has been synthesized for **stim** yet  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **stim** belongs to  
 Create a control node sequence as in Fig. 3;  
 Connect arc STATE to PARENTNODE;  
 Query the human designer for the initial state, place a token on arc STATE or NOTSTATE, and record the initial state into the interpretation code;  
 Assign pseudo interpretation code for node STATE.NODE —  
 (cond  
 ((**Strigger**) include STATE.SIGNAL  
 (\$output\_arc (and STATE A2))  
 record status 'state')  
 ((**Strigger**) include NOTSTATE.SIGNAL  
 (\$output\_arc (and NOTSTATE A2))  
 record status '~state')  
 (else  
 (let ((Current.Status from status recorded))  
 (cond (Current.Status = 'state'  
 (\$output\_arc (and STATE A2))  
 (Current.Status = '~state'  
 (\$output\_arc (and NOTSTATE A2))))))  
 Assign interpretation code for node DUMMY — (\$delay 0)
- stim.14  
*Antecedent:* **stim** is a state stimulus  
 $\wedge$ a control node sequence, as in Fig. 3, has already been synthesized for **stim** or its complement  
 $\wedge$ the arc corresponding to **stim** is not pointing to a node synthesized from a DE  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **stim** belongs to, and the arc STATE in Fig. 3 be the arc corresponding to **stim**  
 Fork arc STATE to PARENTNODE, make it an arc with multiple heads — {STATE.NODE, PARENTNODE}.
- stim.2  
*Antecedent:* **stim** is a physical stimulus  
 $\wedge$ control domain primitives have already been synthesized for **stim**  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **stim** belongs to, PHY.ARC be the control arc synthesized for  
**stim**  
 If PARENTNODE  $\notin$  headset of PHY.ARC already  
 Then  
 Duplicate PHY.ARC and instead of its own headset, make it point to PARENTNODE.  
 endif
- stim.5  
*Antecedent:* **stim** is a physical stimulus  
 $\wedge$ nothing has been synthesized for **stim** yet  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **stim** belongs to  
 Create a no-tail control arc heading into PARENTNODE.

the triggering arcs, which indicates a switch to  $\neg$ state in the system, a token is deposited on **notstate**, and the status  $\neg$ state is recorded in the interpretation domain

- otherwise, a token is deposited on **state** or **notstate** according to the state status previously recorded.

This particular model only works for a binary state stimulus, but it is trivial to enhance the node sequence to model a stimulus of more than two states. For each additional state, simply create an additional multihead arc, like **state**, originating from the **state.node**. The synthesis rule for a state stimulus is given in stim.12.

Before synthesizing any control node sequence for a stimulus or response, the synthesizer also has to consider bookkeeping; i.e., whether or not a node sequence already exists for a certain requirement primitive. If it does, there is no reason to create a replica. It requires only some minor adjustments on the existing node sequence as well as an additional connection between it and the synthesized node of the DE. With that in

mind, synthesis rules for stimuli and responses are based on their operational definitions, as well as their current synthesis statuses. For rule stim.12, there is an associated rule for bookkeeping (stim.14).

*b) Physical Stimulus:* A physical stimulus may be cumulative or noncumulative. A stimulus such as a toggle signal or a message for recording is considered cumulative, since all instances produced by the producer have to be consumed eventually. A single control arc in the GMB control domain and a dataset with properties of a queue in the data domain are adequate to model this stimulus. Two synthesis rules for physical stimulus are presented in stim.2 and stim.5.

2) *Synthesis Rules for Responses:* Corresponding to synthesis rules for stimuli, there is a set of design synthesis rules for responses. This section presents the formal definitions of two responses, as well as their corresponding synthesis rules.

resp.6  
*Antecedent:* **resp** is a state response  
 $\wedge$ a node sequence, as in Fig. 3, has already been synthesized for the state, or the complement of state  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **resp** belongs to, and RESP.ARC be the state-changing stimulus arc heading into node STATE  
 Add PARENTNODE to the tailset of RESP.ARC

resp.14  
*Antecedent:* **resp** is an action which changes a state  
 $\wedge$ a node sequence, as in Fig. 3, has already been synthesized for state or  $\neg$ state  
*Consequence:* Let PARENTNODE be the node synthesized from the DE where **resp** belongs to, and Fig. 3 be the node sequence already synthesized  
 Construct a node sequence as in Fig. 4, with ACTION.INIT added to the tailset of NOTSTATE.SIGNAL, and ACTION.DONE added to the tailset of STATE.SIGNAL.

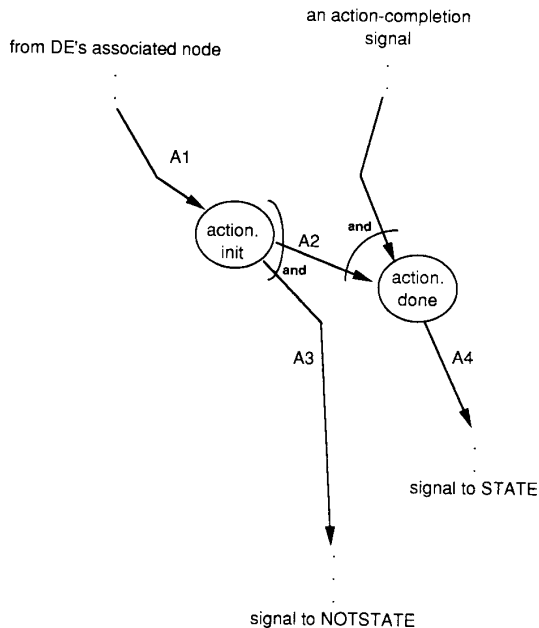


Fig. 4. Control node sequence to model state-switching action.

a) *State response:* A single control arc is enough to model a state switching response. For instance, the response that changes a system state is simply represented by the arc **state.signal** or **notstate.signal** in Fig. 3. Suppose a node sequence has already been constructed for the state; a synthesis rule for this response is presented in resp.6.

b) *Action response:* An action which eventually changes a system state may have an unspecified duration time. If the completion of an action leads to the truth of a *system state*, it means that in the duration of the action, *state* is still false, or  $\neg$ *state* is true. To model this scenario, two control arcs are used to represent responses *state* and  $\neg$ *state*. A node sequence is then built on top of them. Fig. 4 shows how a state-switching action is modeled. The node **action.init** will deposit a token on **A3**, causing a system state to be false. After the action is completed, the node **action.done** will put a token on **A4** and switch the *system state* to true. Suppose the node sequence for state is already constructed; the synthesis rule for this action response is presented in resp.14.

3) *Modeling of Event Dependency:* In the stimulus-and-response model, each decomposition element represents a system or subsystem, invoked by its stimuli to produce responses. When synthesizing control domain primitives from a decomposition element, normally one control node is sufficient for a DE. However, if logical relations are taken into account, it is possible to use one control node to represent multiple DE's, or multiple control nodes to represent one DE. It depends on the relation and the stimulus involved.

The modeling is straightforward if it is a SEQUENCE relation, say, originating from DE **DE<sub>1</sub>** and heading to DE **DE<sub>2</sub>**. A control node corresponding to **DE<sub>2</sub>** is created, connected to the node corresponding to **DE<sub>1</sub>** by a control arc.

For multidestination relations—AND, EXCLUSIVE-OR, SEQUENTIAL-EXCLUSIVE-OR, or SEQUENTIAL-INCLUSIVE-OR—more possibilities arise. It depends on the number of stimuli in the destination DE's, and the appearance of the common stimulus within the DE's.

A multidestination relation means that, conceptually, upon arrival of the common stimulus, one or more DE's within the group will be activated. The relation itself and the nature of the stimulus determine the ones actually invoked. To model this scenario with the control domain, the basic idea is to construct a control node **DE<sub>i</sub>.node** for each decomposition element **DE<sub>i</sub>**, each of which consists of some response-producing actions. On top of the set of control nodes generated, there is one or more decision nodes examining the stimulus and determining which **DE<sub>i</sub>.node** to activate.

In a SEQUENTIAL-EXCLUSIVE-OR relation with the common stimulus being a state, the truth or falsity of the state is going to invoke one of the DE's. The control node sequence to define the semantics of such a relation is illustrated in Fig. 5, in which a bold arc represents a collection of control arcs synthesized from all the other stimuli or responses of a DE in the said relation. The decision-making node, **DEC.NODE**, deterministically decides which DE-associated node to invoke.

The synthesis rules to create control domain objects for this multiple-destination relation are given in Rel.2 and GroupDE.4.

4) *A Sample Synthesis:* This subsection presents a small-scaled sample synthesis of GMB objects from the requirements. The sample requirement is a recording subsystem. The recording activities, as specified in the requirements, depend on two parameters—whether or not the recorder itself is idle,



**Rel.2**

**Antecedent:** Rel is a multidestination relation — AND, EXCLUSIVE-OR, SEQUENTIAL-EXCLUSIVE-OR, or SEQUENTIAL-INCLUSIVE-OR

**Consequence:** Subgoal: synthesize control domain objects for destination DE's of Rel

**GroupDE.4**

**Antecedent:** DEs are destination of a SEQUENTIAL-EXCLUSIVE-OR relation, which is associated with a state stimulus

**Consequence:** Create a control node DEC.NODE;

**Subgoal:** synthesize a node sequence for state stimulus if necessary, let STATE be the arc representing the stimulus;

Make arc STATE point to DEC.NODE;

For  $DE_i$  among  $DEs = \{DE_1, DE_2, \dots, DE_n\}$ ,

Create a node  $DE_i.node$  if necessary, and an arc  $A_i$  connecting DEC.NODE and  $DE_i.node$ ;

**Subgoal:** synthesize control arcs heading to DEC.NODE, for  $DE_i$ 's remaining stimuli;

Assign pseudo interpretation code for  $DE_i.node$  —

*response-producing code* for  $DE_i$ ;

Assign input and output logic to DEC.NODE;

Assign pseudo interpretation code for node DEC.NODE —

`(cond (($trigger) include DE1.stim`

`($output_arc A1)`

`($trigger) include DE2.stim`

`($output_arc A2)`

`...`

`($trigger) include DEn.stim`

`($output_arc An))`

For  $DE_i$  among DEs,

**Subgoal:** synthesize control arcs originating from node  $DE_i.node$  for  $DE_i$ 's responses;

**Subgoal:** synthesize control domain primitives for  $DE_i$ 's output relations;

Result control node sequence for the current group is shown in Fig. 5.

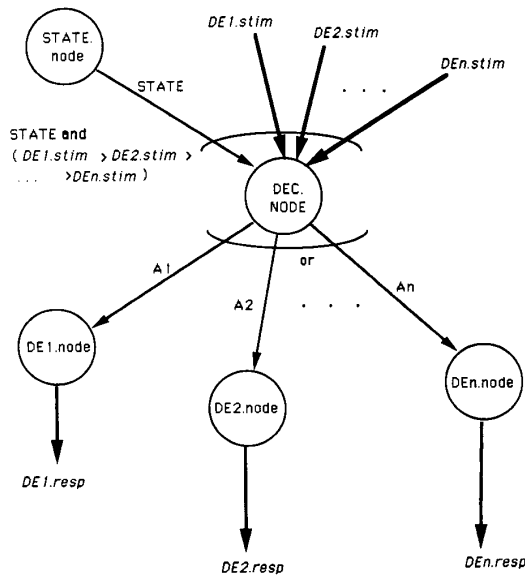
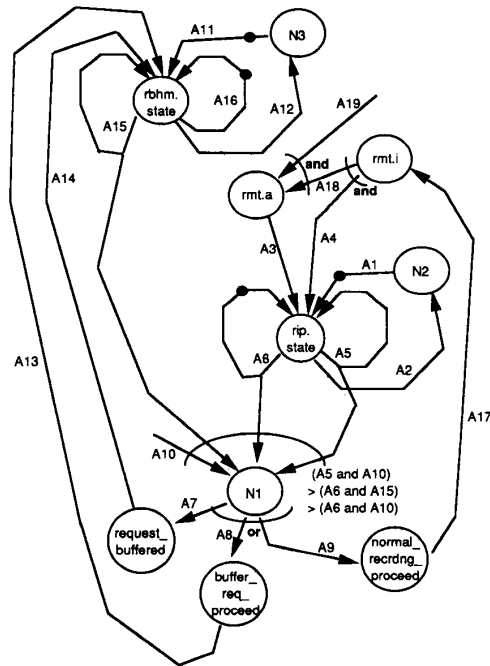


Fig. 5. Modeling of SEQ-XOR relation with state common stimulus.

and whether or not the recorder buffer is empty. As a result, node sequences for system states *recording in progress or not* and *recorder buffer has messages or not* are essential in the recorder module. In sum, this example illustrates control domain synthesis for a multidestination relation, two state stimuli and responses, an action response, and a physical stimulus.

Input to the synthesizer is the System Verification Diagram in Fig. 2. After the firing of 13 rules, a control graph skeleton is constructed, as shown in Fig. 6.



**Note:**

- rbhm.state** - represents the state 'recorder buffer has message'
- rip.state** - represents the state 'recording in progress'
- rmt.l** - represents the initiation of action 'record message on device'
- rmt.a** - represents the completion of 'record message on device'
- N1** - driver node of the multi-destination relation

Fig. 6. Control graph skeleton synthesized for recorder.

- 1) rule SVD.1 for the SVD for *recorder*.
- 2) rule Rel.2 for the sequential-XOR relation leading to decomposition elements *REQUEST BUFFERED*,

*BUFFERED REQUEST PROCEED*, and *NORMAL RECORDING PROCEED*.

- 3) rule *GroupDE.4* for the decomposition elements *REQUEST BUFFERED*, *BUFFERED REQUEST PROCEED*, and *NORMAL RECORDING PROCEED*.
- 4) rule *stim.12* for the stimulus *recording in progress*.
- 5) rule *stim.14* for the stimulus *recording not in progress*.
- 6) rule *stim.5* for the stimulus *recording request* in *REQUEST BUFFERED*.
- 7) rule *stim.12* for the stimulus *recorder buffer has message* in *BUFFERED REQUEST PROCEED*.
- 8) rule *stim.2* for the stimulus *recording request* in *NORMAL RECORDING PROCEED*.
- 9) rule *resp.6* for the response *recorder buffer has message* in *REQUEST BUFFERED*.
- 10) rule *x.resp.3* for the response *recorder buffer has message* in *REQUEST BUFFERED*.
- 11) rule *resp.6* for the response *recorder buffer has no message* in *BUFFERED REQUEST PROCEED*.
- 12) rule *x.resp.3* for the response *recorder buffer has no message* in *BUFFERED REQUEST PROCEED*.
- 13) rule *resp.14* for the action response *record message on device* in *NORMAL RECORDING PROCEED*.

Among the rules fired, rule *x.resp.3* deals with response produced for external context; i.e., another SVD. However, since the system state *recorder buffer has message* and its complement are only referenced in the current context, the consequence of this rule is simply

Do not synthesize anything.

### C. Synthesis of Structural Model and Data Domain

Compared to the synthesis of the control domain, syntheses of the structural and data domains of a design are straightforward. Requirements expressed in the other requirement model, the Data Flow Model, provide the basis for structural and data domain synthesis. Since the Data Flow Diagrams are organized in a hierarchical manner, the synthesizer simply transforms the hierarchical representation of the Data Flow Model to the Structural Model, while taking care to conform to the model's syntax, restrictions, and semantics. On the other hand, the Data Flow Diagrams at the lowest level are the input for data domain synthesis. A majority of the data-flow primitives at the lowest level may be implicitly mapped to SARA's data domain primitives. This also leads to a fairly straightforward transformation.

### D. Completeness of the Rule Set

There are two completeness issues. The first is the syntactic completeness of the rule set, and the second is the degree of automation engendered by the rule set. The set of rules is syntactically complete, since it takes care of every possible requirement construct. Any requirement in the form of an SVD may be fed into the synthesizer to produce a SARA design skeleton. In the context of a particular requirement model as input and a particular design model as output, the model

transformation is complete. However, the design synthesizer may not be considered as a completely automatic designer replacing the role of a human. This incompleteness stems from one or more of the following reasons:

- The system currently does not handle automatic synthesis of the interpretation domain associated with each control node. Synthesis of such a domain, in **T** code, from some natural language specifications requires an automatic code-synthesizer, which is generally impossible and is outside of the scope of this research.
- In an actual design, the human designer often considers design alternatives based on criteria like resource trade-offs, modularity, component reusability, domain-specific knowledge, etc. The rule set is always subject to expansion because of existences of such alternatives.
- The classification of requirement entities is not primitive enough to provide a precise correspondence between the requirements and the design constructs. It is difficult for the synthesizer to understand the actual semantics of a requirement entity to generate the most concise, appropriate design entity, even though such semantics may be trivial to a human.

As a result, the synthesizer cannot be made completely automatic, at least not until much more expertise about system design and knowledge on all potential systems to be designed are codified into the rules and engine. In other words, the knowledge base is always subject to expansion until all universal knowledge about hardware and software system design is included. This is why the synthesizer can at best serve only as a design assistant.

### E. Role of Human Designer

Before, during, and after a synthesis session, certain human involvement is required. The design synthesizer is not to replace the human, but to design a skeleton of the system under the direction of a human. The human designer assumes three responsibilities in a design session. To start the synthesis, he or she selects the appropriate requirement diagram for the synthesizer to create a design. During the synthesis, he or she interactively provides information to the synthesizer, regarding the system being designed (e.g., the placement of a token on arc **A5** or **A6** in Fig. 6, indicating the initial state of *recording in progress*) and guides the synthesizer to design according to his or her preference (e.g., select one if design alternatives exist). Finally, after the synthesis is done, he or she patches up the unfinished parts of the design. The finishing touch includes connecting the tailless or headless control arcs (e.g., **A10** and **A19** in Fig. 6) to the appropriate nodes or sockets, and converting the pseudo interpretation code to actual **T** code.

## V. CONCLUSION

This paper addresses a design synthesizer which aids the human designer in a requirement-driven design method. We conclude this paper with the status of this research, as well as what it does and does not achieve.

A Design Assistant prototype was developed to support the claims in this research. The goal of this implementation is to illustrate this concept of automatic design synthesis as well as to test the synthesis rules. It was implemented on the SUN workstations at UCLA, on top of a prototypical SARA design environment [16]. Using the object-oriented programming paradigm, this prototype consists of tools that:

- create the System Verification Diagrams and Data Flow Diagrams, in the form of objects in the requirement models, and
- synthesize SARA's structural and behavioral models, in the form of objects in the SARA domain, from the two requirement models.

Like the original SARA design tools, the Design Assistant is coded in T. The requirement and design primitives are implemented as T objects. The definition of a requirement object carries both static attributes and dynamic synthesis statuses. Take a sample rule, *stim.5*; its T representation is given as follows:

```
(imply (stim)
  (and (cg.phy_stim? stim) (not_yet_syn?
    stim))
  (cg.syn_phy_stim stim))
```

where *stim* is the bound variable within the rule. Predicate *cg.phy\_stim?* and routine *cg.syn\_phy\_stim* are coded as T functions. Routine *cg.syn\_phy\_stim*, by calling a sequence of GMB object editing routines, simply creates a new control arc object and makes it point to the appropriate node. Predicate *not\_yet\_syn?* is an operation of the *stim* object.

The synthesis knowledge encoded is based on the formalization of the semantics of the requirement constructs, as well as certain trade-off, bookkeeping, and optimization decisions during a design session. According to this knowledge, 21 rules are derived for structural model synthesis, 59 for control domain synthesis, and 37 for data domain synthesis. This set of rules is geared toward the general, domain-independent design problems.

Using this prototype, a total of five syntheses were carried out. These five examples were the submodules of an aircraft monitor system. The requirements of these modules consisted of a wide range of constructs, including most of the stimuli—physical, synchronous, state, and disjunction—most of the responses—physical, state, action and response sequence—and three kinds of relations—SEQUENCE, EXCLUSIVE-OR, and SEQUENTIAL-EXCLUSIVE-OR with various forms of common stimulus. After more than 200 rule firings, the Structural Model, Control Graphs, and Data Graphs synthesized provided the skeleton of a SARA design model for the aircraft monitor. After the human designer patched up the unfinished parts of the three domains, the model was successfully simulated by the GMB simulator. Details of all the synthesis rules, the requirement examples, the sample synthesis, and the design model produced are available in [17].

This research provides a better understanding and a methodical approach of the SARA-based design process with respect

to the two requirement methods. However, its application is not limited to any particular design models employed. To accommodate another model, such as the Petri Nets or PDL, as the synthesis output, simply replace the current set of rule consequences by a new one knowing how to generate design in the desired model.

## REFERENCES

- [1] R. J. Abbott, "Program description by informal English description," *Commun. ACM*, Nov. 1983.
- [2] M. Alford, "A requirement engineering methodology for real time processing requirements," *IEEE Trans. Software Eng.* vol. SE-3, pp. 60–69, Jan. 1977.
- [3] M. Alford, "SREM at the age of eight: the distributed computing design system," *IEEE Computer*, pp. 36–46, Apr. 1985.
- [4] P. C. Belford and D. S. Taylor, "Specification verification—a key to improving software reliability," in *Proc. Symp. Computer Software Eng.*, Apr. 1976, pp. 83–96.
- [5] D. M. Berry, N. Yavne, and M. Yavne, "Application of program design language tools to Abbott's method of program design by informal natural language descriptions," *J. Syst. Software*, pp. 221–247, Sept. 1987.
- [6] S. H. Caine and E. K. Gordon, "PDL—a tool for software design," in *Proc. Nat. Computer Conf.*, 1975, pp. 271–276.
- [7] T. de Marco, *Structured Analysis and System Specification*. New York: Yourdon, 1979.
- [8] M. S. Deutsch, "A multiple view paradigm for modeling and validation of real-time software systems," in *Proc. Int. Conf. on Reliability and Robustness of Eng. Software*, Sept. 1987.
- [9] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A design methodology and computer aids for digital VLSI systems," *IEEE Trans. Circuits Syst.*, vol. CAS-28, July 1981.
- [10] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA (System ARCHitects' Apprentice): modeling, analysis, and simulation support for design of concurrent systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 293–311, Feb. 1986.
- [11] F. S. Etesami and G. S. Hura, "Rule-based design methodology for solving control problems," *IEEE Trans. Software Eng.*, vol. 17, pp. Mar. 1991.
- [12] K. F. Fischer and M. G. Walker, "Improved software reliability through requirements verification," *IEEE Trans. Rel.*, vol. R-28, pp. 233–240, Aug. 1979.
- [13] R. Jain, K. Kucukcakar, M. J. Mlinar, and A. C. Parker, "Experiences with the ADAM synthesis system," in *Proc. 26th ACM/IEEE Design Auto. Conf. (Las Vegas, NV)*, 1989.
- [14] D. W. Knapp and A. C. Parker, "A design utility manager: the ADAM planning engine," in *Proc. 23rd ACM/IEEE Design Auto. Conf.*, June 1986, pp. 48–54.
- [15] T. J. Kowalski, *An AI Approach to VLSI Design*. Boston: Kluwer, 1986.
- [16] E. Krell and E. Lor, "Current state of the SARA/IDEAS design environment," in *Proc. Softfair II*. New York: IEEE, Dec. 1985, pp. 218–230.
- [17] K. E. Lor, "An assistant for requirement-driven system design," Ph.D. diss., Computer Sci. Dept., Univ. California, Los Angeles, 1988.
- [18] M. D. Lubas and M. T. Harandi, "Knowledge-based software design using design schemas," in *Proc. 9th Int. Conf. on Software Eng.*, Mar. 1987, pp. 253–262.
- [19] D. Partridge, *Artificial Intelligence Applications in the Future of Software Engineering*. Chichester, UK: Ellis Horwood, 1986.
- [20] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [21] R. Razouk, M. Vernon, and G. Estrin, "Evaluation methods in SARA—the graph model simulator," in *Proc. Conf. on Simulation, Measures and Modeling of Computer Syst.*, 1979, pp. 189–206.
- [22] R. Razouk and G. Estrin, "The graph model of behavior," in *Proc. Symp. on Design Automation and Microprocessors*. New York: IEEE, Dec. 1980, pp. 67–76.
- [23] S. Slade, *The T Programming Language: A Dialect of Lisp*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [24] G. M. Swinkels and L. Hafer, "Schematic generation with an expert system," *IEEE Trans. Computer-Aided Des.*, vol. 9, Dec. 1990.
- [25] D. Teichroew and E. A. Hershey III, "PSL/PSA: a computer-aided technique for structure documentation and analysis of information processing system," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, Jan. 1977.



**Kar-Wing Edward Lor** received the B.S. degree from the University of Maryland, College Park, and the M.S. and Ph.D. degrees from the University of California, Los Angeles, all in computer science.

As a student, he was involved in the high-level language computer architecture project at the University of Maryland, and the SARA/IDEAS design methodology project at UCLA. In 1988 he joined the Data Communication Research Department of AT&T Bell Laboratories as a member of the technical staff. His research interests include design

automation, requirement specifications, expert systems, and network management.

Dr. Lor is a member of the ACM and the IEEE Computer Society.



**Daniel M. Berry** received the B.S. degree in mathematics from the Rensselaer Polytechnic Institute, Troy, NY, in 1969, and the Ph.D. degree in applied mathematics and computer science from Brown University in 1974.

From 1972–1987 he was on the faculty of the Computer Science Department at the University of California, Los Angeles. Currently, he is a Professor in the Faculty of Computer Science at the Technion, and is a member of the technical staff of the Software Engineering Institute. His research interests are

in software engineering and electronic publishing.

Dr. Berry is a member of the ACM and the IEEE Computer Society.