Sādhanā, Vol. 21, Part 6, December 1996, pp. 719–739. © Printed in India.

An overview of MICKEY: A knowledge-based hardware–software codesign framework for microprocessor-based systems

RAJ S MITRA, PARTHA S ROOP and ANUPAM BASU

Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721 302, India
email: rsm@cadence.com; anupam@cse.iitkgp.ernet.in

MS received 18 March 1996

Abstract. We present the architecture of a second-generation expert system for the automated design of microprocessor-based systems. A novel feature is the integration of a device handbook knowledge base with a shallow expert system, to provide resilience and deep reasoning capability. The design tasks, knowledge sources, behaviour modelling scheme, behaviour mapping algorithms, and inter-layer communication are briefly described.

Keywords. Expert system; CAD; microprocessor-based systems; deep reasoning; device modelling; incomplete knowledge.

1. Introduction

Today, microprocessor-based systems are widely used in industrial applications because of their low cost, programmability, and ready availability of supporting peripherals. In contrast, complete hardware solutions (e.g. ASIC-based) have much higher costs and design periods, and complete software solutions are often not capable of supporting the real-time constraints of application.

Computer-aided design of microprocessor-based systems require the design of the hardware configuration along with the design of application-specific software and the device drivers. Design automation research in the last decade has primarily concentrated on hardware design *or* software design, but has neglected the key area of *cooperative* design of hardware *and* software. But the increasing demand from the industry for lower design periods has forced the international research community to explore methodologies for automating the *codesign* of mixed (hardware–software) systems (Srivastava & Brodersen 1991; Kalavade & Lee 1993; Kumar *et al* 1993; Smailagic & Siewiorek 1993; Chou *et al* 1994; Hu *et al* 1994) during the last three years.

For designing microprocessor-based systems, it is essential to incorporate the capability of adequate semantic requirement interpretation of problem specification as well as that of the external environment with which the target system is supposed to interact. Only semantic interpretation, for example as in boolean synthesis, is not sufficient for this domain. Hence, a knowledge-based approach is the appropriate paradigm for designing such systems. This approach also facilitates the encoding of design heuristics in the form of situation-action rules. Such heuristics have been traditionally used by human experts. An explicit recourse to all this domain knowledge for solving each and every design problem turns out to be very costly and impractical. In this perspective, the expert system approach provides a flexible solution to this problem.

However, it is well known that the power of a knowledge-based system is very much dependent on the extent of the knowledge encoded in the system. Although the shallow knowledge-based systems perform reasonably well when the solution lies within the boundary of the encoded knowledge, they fail to generate a solution if the association between the relevant problem subgoal and the desired solution has not been encoded in the system. In contrast, *second generation expert systems* (Keravnou & Washbrook 1989) resort to *deep* (causal or behavioural) knowledge when shallow knowledge fails to find a solution. Such an integration of deep knowledge with shallow knowledge is necessary to make a robust knowledge-based design system.

The usage of second generation expert systems is of special relevance in the domain of microprocessor-based systems, where the repertoire of devices is growing at a regular pace. When a human designer fails to achieve an elegant solution with the devices known to his experience, he/she consults device manuals and application notes to find new devices which can solve the desired design subgoal. The design expert system should be able to mimic the human designer's capability of analysing the behaviour of the desired subgoal and the available devices and to generate the necessary interface to the device.

The present article is a report on the first Indian effort in this direction. We have developed a knowledge-based CAD framework for automating the design of microprocessor-based systems. In this paper we propose a knowledge-based system having a novel two-layer architecture, the first layer being a *shallow* layer which synthesizes the target systems using precompiled heuristic transformation rules and associated procedures. Whenever such precompiled knowledge turns out to be insufficient to solve a design subgoal, the second layer – which is the *deep* layer – is invoked. The deep layer is endowed with the behavioural knowledge about the microprocessor peripheral devices and the different design functions. This layer performs behavioural (model-based) reasoning to find a solution for the "failed" subgoal, and returns the result to the shallow layer, which then continues with the design tasks. Thus, two-layer architecture provides a resilient environment for design.

In this paper, our emphasis is on presenting the integrated environment, and on the communication and cooperation between the two layers. For completeness, we also present, briefly, the salient features of the individual layers. Details of the shallow layer can be found in Mitra *et al* (1993, 1994), and the description of the behavioural mapping algorithms can be found in Mitra *et al* (1996).

2. Related works

Automated hardware software codesign has been used in a variety of applications: DSP (Kalavade & Lee 1993), wearable computers (Smailagic & Siewiorek 1993), automobile control (Hu *et al* 1994), robot control (Srivastava & Brodersen 1991) etc. Two key issues in hardware–software codesign are hardware–software partitioning (Kumar *et al* 1993) and hardware–software interface synthesis (Chou *et al* 1994). A brief survey of the issues and approaches in hardware–software codesign is available in Micheli (1994). Hardware–software codesign is essentially a system-level design problem, and hence draws on past research efforts in high level synthesis (McFarland *et al* 1990) (especially on the issues of partitioning, allocation, scheduling and hardware synthesis), and on those in automated synthesis of domain-specific software (Barstow 1985; Jullig 1993).

AI techniques have been effectively used to solve the complex problem of system design. Chippe (Brewer & Gajski 1990, 1991) is a hybrid expert system for design, that encodes analysis knowledge in the form of rules and implementation knowledge in the form of procedures, resulting in fast executions. Other noteworthy CAD systems for the design of computer configurations and circuits are MICON (Tseng & Siewiorek 1986), R1 (McDermott 1982) and VEXED (Mitchell *et al* 1985).

A number of researchers have investigated the issue of integrating the deep level knowledge about the device, in design and diagnosis problems (Keller *et al* 1990; Keuneke 1991). Most of these approaches use a *structure function behaviour* model of the device as a form of deep level knowledge. The *structural model* is based on the physical organization of the components. The *function* of the device is its intended purpose. The functional specification describes the device's goals at an abstract level. Functions are achieved by *behaviours*. In other words, function is what is expected and behaviour is how this expected result is achieved (Keuneke 1991). Behaviour is often represented as a causal sequence of transition of partial states.

3. System overview

As mentioned earlier, we have developed a two-layer architecture for a knowledge-based design system, in order to allow the design process to have the advantages of shallow as well as deep reasoning. On the occurrence of a failure in the shallow layer, the deep layer is invoked to resolve the failure. If that succeeds, the shallow layer continues with its processing, using the data returned by the deep layer. If, however, the deep layer fails to generate the required solution, the shallow layer backtracks and tries to find another problem decomposition – one that does not contain the failed subgoal.

The two-layer architecture of our synthesis system is shown in figure 1. The *S-layer* is the expert system (christened MICKEY) that performs the synthesis tasks with the help of *shallow* design knowledge compiled by the human expert. This layer uses precompiled shallow knowledge about implementation of specific design functions in order to translate the problem specifications into the target system's hardware and software. This layer is a hybrid expert system (Brewer & Gajski 1990; Bailey *et al* 1991; Kambhampati *et al* 1993), in which several knowledge sources, rule-based as well as procedural, interact and contribute towards generating the solution.

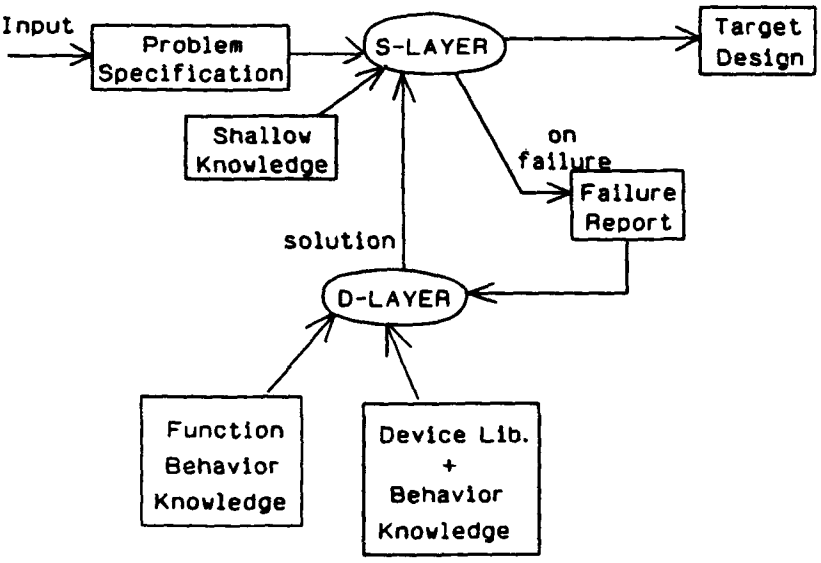


Figure 1. Overall system architecture.

However, knowledge is seldom complete. When the S-layer detects that it does not have any knowledge about implementing some design subgoal, it invokes the *D-layer* (called MINNIE) by passing the details of the "failed" subgoal. The D-layer contains the behavioural models of available devices, organized in the form of a database. This layer also contains procedures for retrieving relevant device information from the database, and for mapping the behaviour of the failed subgoal to the behaviours of the retrieved devices. Thus, the database of the D-layer coupled with the mapping algorithms serve as a deep knowledge-based system, which can find solutions from first principles. It may be noted that the purpose of the D-layer is essentially to mimic a human designer who, when running short of experiential knowledge, consults device handbooks to arrive at a solution. On successfully finding a device that can implement the failed subgoal, the D-layer passes the specifications of the device's interface to the S-layer, which then continues with the synthesis taks.

The S-layer and the D-layer together form a two-layer knowledge-based architecture for the synthesis of microprocessor-based systems. The integration of the D-layer with the S-layer aims at making the design system resilient.

4. The S-layer

The S-layer translates the problem specifications into the target system's hardware and software, by using precompiled design knowledge. A schematic overview of the S-layer is shown in figure 2, which depicts the sequence of design tasks, the different knowledge sources used, the supporting subtasks, and the blackboard-like shared data structure for storing partial design results.

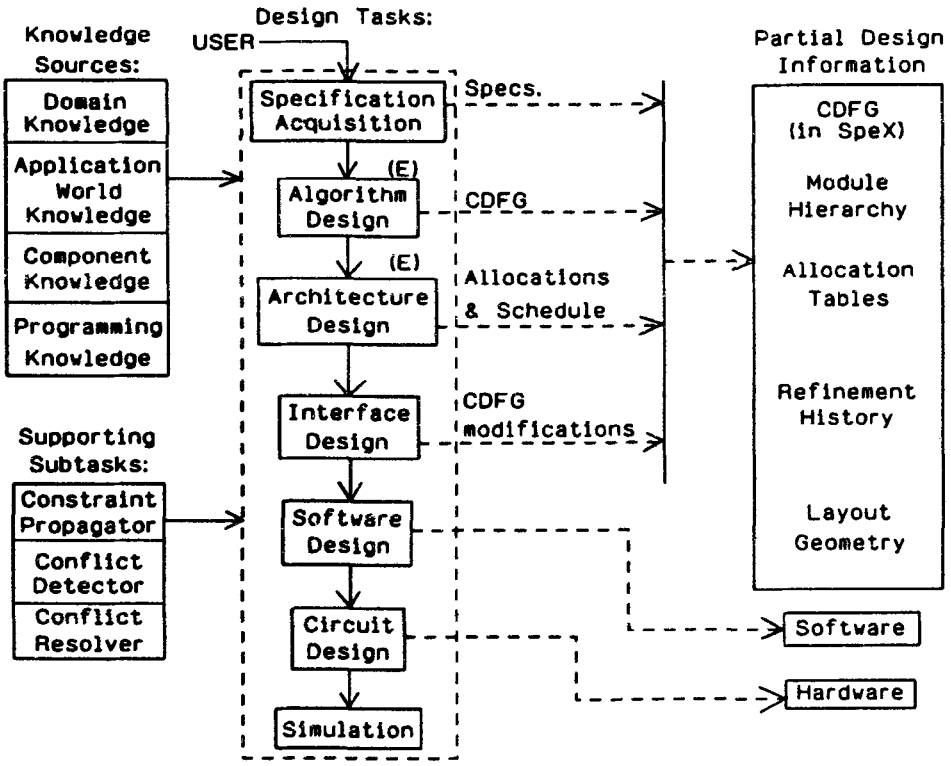


Figure 2. Architecture of S-layer.

4.1 *The design tasks*

In order to synthesize the target system, several tasks have to be performed, such as the acquisition of the specifications from the user, functional decomposition, hardware–software partitioning, interface synthesis, software synthesis, circuit synthesis and simulation.

These tasks use several categories of knowledge to achieve their objectives. The different representations of knowledge have been discussed in a later section. The different categories of knowledge used for the purpose are the Application World Knowledge, Design Refinement Knowledge, and Device Knowledge.

4.1a *Specification acquisition:* The specification of the target system is represented in SpeX, which is a visual language based on the statechart language (Harel 1987). Statecharts have constructs for the modular specification of the control flow and concurrency of complex systems. In addition to these constructs, SpeX also has features for specifying data flows, data constraints, real-time constraints and implementation preferences for the functional elements (FEs).

These specifications are acquired from the user with the help of a graphical user interface, and then converted to an equivalent textual form. The specifications are validated by checking for syntactic correctness.

4.1b *Algorithm design:* The specification may contain nodes for abstract functionalities, which have to be decomposed into a lower level of detail. This decomposition is performed by the present task, by a top-down refinement process. Every decomposition step introduces new FEs, control flows, data flows and constraints, and also maps the inputs and outputs of the parent function to those of the children. The design refinements are performed by design refinement rules, which have been encoded in a specialized shell, APS, designed for this purpose.

The selection of the appropriate refinement to be made to the partial design, consists of two decision steps.

(1) Selection of the function or subgoal to be refined, and

(2) selection of the refinement to be made on the selected function.

These decisions are made by the following strategy:

- select the function that has the minimum (nonzero) number of applicable refinements.
- for the chosen function, select a refinement that satisfies the maximum number of constraints.

This strategy has been formulated with an analogy to the Consistent Labelling Problem (CLP) (Haralick & Shapiro 1979), where a variable is selected that has the minimum domain set of candidate values and for the chosen variable, a value is selected that satisfies the maximum number of constraints.

The refinement process goes hand in hand with constraint propagation and conflict resolution. Constraints are propagated (Steinberg 1987) from one part of the design to another in order to (i) influence the proper choice of future refinement steps, and (ii) obtain conflict-free designs. Depending on the nature of the constraints, the conflicts are handled in different ways:

- Conflicts in interval constraints, that are imposed on parameters, are resolved by the algorithm proposed by Hyvonen (1992). In this method, such constraints are resolved by taking the intersection of the conflicting intervals.
- Mismatches in data type constraints are resolved by introducing a patch-up FE to transform one constraint to the other. For example, an analog-to-digital-converter is introduced to convert an analog signal (say, current) to its digital equivalent, so that necessary computations may be performed by the microprocessor.
- When communicating processes execute at different rates, buffers are set up to store the communication data.
- If a storage location is simultaneously accessed by more than one concurrent processes, a conflict in the address bus is generated. This type of conflict is resolved by introducing an arbitration mechanism for the access, such that a slower process has precedence over a faster process.
- If a conflict on performance constraints is detected, a preliminary hardware–software partitioning is done. Conflicts on performance constraints are due to (i) processor requirement conflicts, which arise due to conflicting concurrent processes, or (ii) data access time conflicts.

The resultant partial design is a control and data flow graph (CDFG) where each FE has a known implementation, hardware and/or software. These FEs are called primitive functions (PFs).

4.1c *Architecture design:* A key issue in hardware–software codesign is hardware–software partitioning. From among the available implementations for each PF of the CDFG, an appropriate implementation has to be selected such that the real-time constraints of the problem are satisfied. In order to determine whether such constraints are satisfied, a feasible schedule of the allocated implementations also has to be formed. When considering single-microprocessor target systems without multi-tasking, the scheduling should ensure a single-thread software. Thus, hardware–software partitioning consists of allocation of specific implementations for each PF, and scheduling these implementations. This is performed by the present task.

The allocation and scheduling steps are formulated as an integrated CLP, where a PF is analogous to a variable and an available implementation is analogous to a value. The solution to this CLP, subjected to two sets of constraints (the timing constraints to be satisfied and the area cost constraints to be optimized), would lead to two sets of partitions, one consisting of the hardware implementations and the other consisting of the software implementations. But for this application, a few extensions to the conventional CLP is required. They are:

- The set of variables is *dynamic*, since extra PFs may be added to the partial design to resolve conflicts between implementations of interacting PFs.
- The cost of the target system does *not* increase with every labelling, since reuse of already allocated implementations does not add to the cost of the design.
- *Two* levels of backtracking have to be considered, for the allocation and scheduling steps respectively.

A forward checking algorithm is used to find a solution to the CLP, and the user is allowed to specify a time bound for the search. The branch-and-bound search technique results in a monotonic decrease in the solution cost. Search heuristics are used to quickly converge to the optimal cost.

4.1d *Interface design:* After the hardware–software partition is formed, the synthesis of the interface between the software partition and the hardware partition is the next key issue that has to be addressed in a hardware–software codesign framework. The present task synthesizes the interface by (i) allocating nonconflicting addresses to the devices to be placed on the system bus, (ii) converting event-based transitions of the CDFG into interrupt service routines (ISRs), and (iii) synthesizing the device drivers.

The addresses are allocated by a heuristic algorithm that attempts to reduce the address decoding logic. The ISRs are created by a top-down refinement process which replaces the event-based transitions by the respective interrupt service actions, thus fragmenting the CDFG. In the modified CDFG, the events are captured by the respective interrupt lines, and the relevant ISRs handle the transitions.

The device drivers are created, again by a refinement process, by synthesizing the interface that allows a device to exhibit the behaviour of the function that has to be implemented by it. This interface, consisting of software as well as hardware modules, programs the programmable devices and performs data transfers (along with the requisite data transformations) to and from the device. A rule for synthesizing device drivers is described in § 4.2.

4.1e *Software design:* At this stage of the design, the CDFG of the partial design has several characteristics: (i) it does not contain any partial orders of FEs, (ii) there are no event-based transitions, (iii) explicit partitions have been formed for software and hardware implementations, and (iv) the implementations of all interfaces between software and hardware implementable modules have been determined, and these have also been partitioned as software or hardware implementations. From the software partition of the CDFG, the target system's software is generated by macro-substitution of C-program templates for each FE, and subsequent compilation to machine code.

4.1f *Circuit design:* This task synthesizes the address decoding circuitry and establishes the pin connectivity of the devices allocated during hardware–software partitioning.

4.1g *Simulation:* In order to verify that the synthesized target system behaves correctly, the software and the hardware are cosimulated. For this, the software is treated as initialization data for the system ROM, and the circuit is simulated. Event-driven simulation is performed, and the behaviours of the devices are represented in SpeX. Faults, if any, are traced manually to their cause, and after correcting the error, the design tasks are executed again.

4.2 *Design knowledge representation*

The processing and knowledge requirements of each task of the S-layer are summarized below. Along with each knowledge item, the type of representation used for that item is mentioned. These representation schemes will be explained subsequently.

(1) *Specification acquisition task:*

- (a) Acquisition algorithm [Procedure]
- (b) The characteristics of design functions [Database]

(2) *Algorithm design task:*

- (a) Refinement steps [Rules]
- (b) Constraint propagation algorithm [Rules]
- (c) World constraints [Rules]
- (d) Function constraints [Rules]
- (e) Conflict detection and resolution strategies [Rules]
- (f) PF-Flags [Facts]

(3) *Architecture design task:*

(a) Hardware–software partitioning algorithm [Procedure]
(b) Knowledge about candidate implementations [Database]

(4) *Interface design task:*

(a) Address constraints of devices [Database]
(b) Address allocation algorithm [Procedure]
(c) Refinement steps [Rules]

(5) *Software design task:*

(a) Program templates [Database]
(b) Macro-substitution and refinements [Procedure]

(6) *Circuit design task:*

(a) Refinement steps [Rules]
(b) Constraint propagation algorithm [Rules]
(c) Device constraints [Rules]
(d) Function constraints [Rules and Facts]
(e) Conflict detection and resolution strategies [Rules]

(7) *Simulation:*

(a) Simulation algorithm [Procedure]
(b) Code for function behaviour [Database]

(8) *Task management:*

(a) Task hierarchy and sequence [Rules]
(b) Failure handling [Rules]

Mainly two types of representation schemes have been used to encode the above requirements: rule-based and procedure-based. The input for the rules is encoded as facts (i.e. the working memory elements of the expert system shell that are not modified by any rule), and the input to the procedures are data items, sorted and indexed on key values. The latter can be conceptualized as a relational database; hence the use of the term *Database* in the above enumerations. These two representation schemes are described below.

4.2a *Rule-based representation:* Rules are mainly used for: i) partial design refinement, ii) constraint propagation and analysis, and iii) task sequencing and failure handling. These rules have been implemented in a production system language APS (Mitra 1995).

The rules for partial design refinement perform functional decomposition of the FEs of the partial designs. Hence, the inputs and outputs of these rules are the different features of the partial design – the FEs, data flows, control flows, concurrency definitions etc. Besides deleting the FE that is being refined (referred to as the *parent*) and creating the new FEs

(referred to as the *children*), these rules also map the data and control flows of the parent to those of its children.

A rule for synthesizing the device driver for an Intel 8253 timer is shown in figure 3a. This rule refines the partial design to implement an up-counter by this device. Given the address allocation of the device and the maximum value of the counter, this rule computes the programming commands for the device and the transformation of the output data. Line 11 of the rule refers to a device constraint – in this case, the address offset for the selected timer. The device constraints are represented as *facts* in the working memory.

Rules are also used for constraint propagation, conflict detection and conflict resolution. Rather than representing the constraints of the application world (e.g. the characteristics of ECG signals) and the devices (e.g. I/O constraints) as semantic nets and traversing these nets in order to collect and propagate these constraints, these constraints are clubbed together with the specific rules that propagate them. The rule for propagating constraints of the ECG domain into the partial design is shown in figure 3b. It defines the minimum sampling rate of an ECG-signal as 200 Hz and the frequency of the periodic ECG-signal as 1.2 Hz.

The propagation of constraints may result in design conflicts. The detection of a conflict, and the strategy for resolving it, are combined into a single rule.

Task sequencing, as well as inter-layer communication, is also performed by rules. The strategy followed here is the same as that used by R1 (McDermott 1982), by using the principle of maximum specificity to initiate and terminate tasks.

4.2b *Procedural representation:* This kind of processing has been used for (i) specification acquisition, (ii) hardware–software partitioning, (iii) software generation, and (iv) simulation. The procedures have been encoded as C programs, and their input knowledge is accessed from their relevant input files.

5. The D-layer

The architecture of the D-layer is shown in figure 4. Input to D-layer is in the form of a failure report from S-layer. This report contains information about the design function for which S-layer did not have implementation knowledge. The output of D-layer is the list of devices and their interfaces which can implement the desired function.

The behaviours of design functions and available devices are stored in two databases, the Function Database (FDB) and the Device Database (DDB). Based on the contents of the failure report, D-layer retrieves the behaviour of the function and the behaviours of *candidate* devices that *may* implement the function. Subsequently, the behaviour of the function is compared with the behaviours of each of these candidate devices, and the functional specifications of the necessary interface is generated by the behaviour-mapping algorithms. Subsequently, an implementation of the interface is synthesized by using the knowledge about the signal constraints and programming modes of the selected device.

The primary task of D-layer is to compare two behaviours, that of a design function and an available device, in order to determine whether the former can be implemented by the latter. Achieving this objective, requires that the device behaviours and the function

```
(a) 1. (prog-dev-8253-upcounter-init
    2.          ;; initialize counter-by-8253
    3.    (context
    4.    (task * 0 :fn int-devdrv)
    5.    ?s = (fe :fn start :attrs ?s0)
    6.    (fe * ?s0 :fn up-counter :attr ?a1)
    7.    (impl :fn ?s0 :id ?id :dev 8253 :part ?prt)
    8.    )
    9.    ?mc = (getval ?a1 :maxcnt)
   10.    (gt ?mc 255)
   11.    (devinfo :dev 8253 :part ?prt :addr ?ct)
   12.    (addr-space :id ?id :mapping ?m :from ?base)
   13.    ==>
   14.    ?basec = (plus ?base ?ct)      ;; counter address
   15.    ?ctwd = (plus (mult ?ct 64) 48)
   16.    ?cnt-h =(trunc (fldiv ?mc 256))
   17.    ?cnt-l =(minus ?mc (mult ?cnt-h 256))
   18.    ?s1 = (make fe :type state :fn write :attrs (setof
   19.                   (item :addr (plus ?base 3))
   20.                   (item :data ?ctwd) (item :map ?m)))
   21.    ?s2 = (make fe :type state :fn write :attrs (setof
   22.                   (item :addr ?basec)
   23.                   (item :data ?cnt-l) (item :map ?m)))
   24.    ?s3 = (make fe :type state :fn write :attrs (setof
   25.                   (item :addr ?basec)
   26.                   (item :data ?cnt-h) (item :map ?m)))
   27.    (make tr :source ?s1 :destn ?s2)
   28.    (make tr :source ?s2 :destn ?s3)
   29.    (make mapc :type exit :oldst ?s :newst ?s3)
   30.    (make mapc :type entry :oldst ?s :newst ?s1)
   31.    (del ?s)
   32.    )

(b)1. (constr-prop-6            ;; transfer ECG constraints
    2.    (context
    3.    (task * 0 :fn algo-constr-prop)
    4.    ?df = (datafl :constr ?c)
    5.    (eq (getval ?c :type) ECG_signal)
    6.    (undef (getval ?c :min_sample_rate))
    7.    )
    8.    ==>
    9.    (mod ?df :constr (join ?c (setof (item :min_sample_rate 200)
   10.          (item :freq 1.2))))   ;; 1.2 == 72/60
   11.    )
```

Figure 3. (**a**) Rule for synthesizing device driver for Intel 8253 to be operated as an up-counter. (**b**) Rule for propagating constraints of ECG-signals.

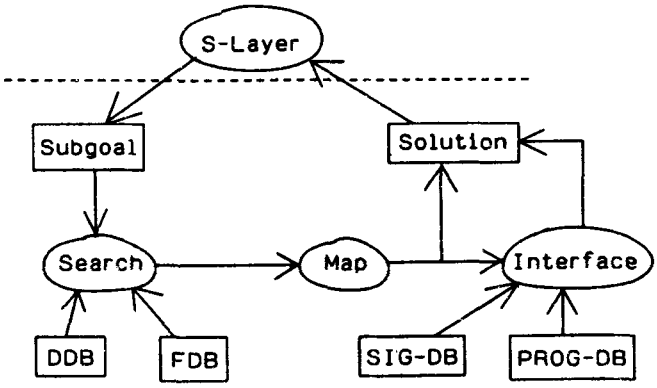


Figure 4. Architecture of D-layer.

behaviours be represented in a proper form, which facilitates the mapping between the two. The formalism of Finite State Machines (FSMs) is commonly used for representing behaviours of a large class of devices and circuits. However, this formalism is not suitable for the purpose of behavioural mapping because of the following reason.

Most devices (e.g. microprocessor peripherals) use memory. Representation of the behaviour of such a device as a single FSM, requires an exponential number of states. This is because the combination of the states of the memory elements is explicitly encoded as states of the machine itself. For example, an FSM representation of an n-bit up-counter requires 2^n states. In such an encoding scheme, transitions between memory states are based on the relevant data input events.

In the present system, we have adopted a new representation formalism which has been used to represent the deep knowledge about the behaviour of the devices and functions. Next, we have developed a mapping approach, which is essentially a search algorithm to see whether a device can implement a given function. It also derives the specification of the interface required to achieve such a match.

5.1 *Representing behavioural knowledge*

One approach for reducing the number of states, is the EFSM modelling scheme (Devadas *et al* 1991). This scheme makes the internal registers explicit, and register update operations are encoded as actions of the transition arcs of the state transition graphs. However, behavioural mapping of two EFSMs requires the performance of reachability analyses of the two behaviours, and this can be very expensive for large problems. The statechart language (Harel 1987) provides another formalism for reducing such an explosion of states, by using compositions of communicating FSMs. In our modelling scheme, termed Composite Finite State Machines (CFSMs), we use a similar composition technique for reducing the number of states.

A schematic of the CFSM model is shown in figure 5. A CFSM consists of a number of constituent machines (which may be FSMs or combinational units), operating concurrently and communicating with each other. One of these constituent machines is termed the

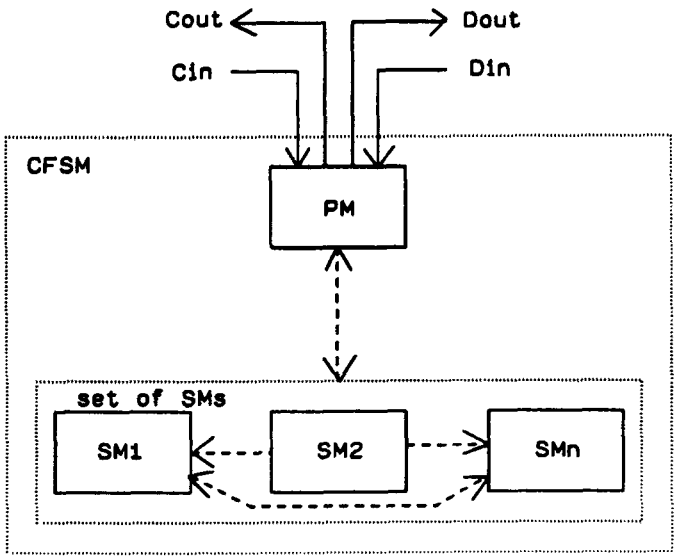


Figure 5. CFSM as a set of communicating machines.

primary machine (PM), which is essentially an FSM and represents the abstract functional behaviour of the overall CFSM. The other constituent machines are called *subsidiary* machines (SMs), and handle the data and memory operations only.

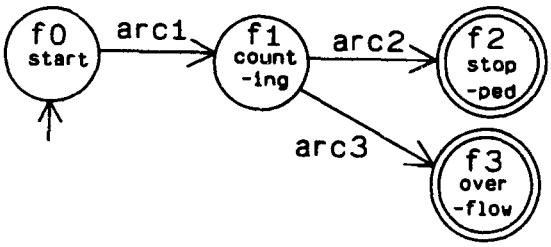
The functioning of an SM is controlled by the PM, or by another SM, via *internal* data and control signals. These internal communications may (i) initiate, stop, suspend or resume the functioning of an SM, (ii) access the results of computations performed by the SM, or (iii) be control signals generated by an SM. All *external outputs* are performed by the PM. As for the *external inputs*, the control inputs are accepted by the PM, and the data inputs are handled by the relevant SMs. The operational semantics of every constituent machine of a CFSM is similar to that of statecharts (Harel 1987).

Every state transition arc in the PM is labelled by a triplet $e[c]/a$, where: (i) e is an event that activates the transition, (ii) c is a set of guard conditions that enable the actual firing of the transition, and (iii) a is a set of actions that are executed by the PM before entering the next state.

The PM of an up-counter is shown in figure 6. In a conventional FSM representation, it would have required $MAXCNT+1$ states. In the PM, f_0 is the start state, f_1 is a nonterminal state, and f_2 and f_3 are terminal states corresponding to whether an overflow has occurred or not before the counter has been stopped.

This PM, $\mathcal{F}$, interacts with two SMs, S_1 and S_2. S_1 embodies the function *no-of(CLK+)*, i.e. it keeps track of the number of occurrences of the rising edges of the CLK input. S_2 represents a function for monitoring the output of S_1. This function generates a control signal when the output of S_1 exceeds a given bound. $\mathcal{F}$ collects data from S_1 when required, and the output of S_2 is used to trigger the *overflow* transition in $\mathcal{F}$.

The reduction in the number of states in this PM is due to the presence of the abstract nonterminal state f_1, which denotes the process of *counting*. When the PM is in this state,



Control Inputs: START, STOP
Data Inputs: CLK(0:1)
Control Outputs: OVFL(0:1)
Data Outputs: DATA(0:MAXCNT−1)
SMs: S1: no-of(CLK+); S2: S1 ≥ MAXCNT

Edge Labelings:
arc1: START/init(S1),init(S2)
arc2: STOP/DATA=S1,OVFL=0
arc3: S2/OVFL=1

Figure 6. PM of an up-counter.

the SM S_1 continues with its task of counting, and the SM S_2 continuously monitors the output of S_1.

5.2 *Behavioural mapping*

The objective of behavioural mapping is to determine the implementability of a function F, by a given device D. A device D implements a function F, if and only if there exists an interface $\mathcal{I}$ such that any input sequence, I, that is accepted by F is also accepted by $D.\mathcal{I}$ (which is the device with the interface attached to it), and the output sequence of F for that input is the same as the output of $D.\mathcal{I}$.

The device's interface may transform the inputs of F before sending them to D. Similarly, the outputs of D may also have to be transformed in order to make them equivalent to those of F. These transformations are determined by the behaviour-mapping process. A library of available transformation operators is maintained, to achieve the required transformation functions.

It is assumed that the behaviours of F and D are represented in the CFSM formalism, as described in the previous section. Although the PM of a CFSM does not *use* the data events directly, the functions that are to be performed on the data are encoded implicitly, by name, within the PM. Moreover, all outputs are generated by the PM alone. Thus, the PM captures the overall functional behaviour of the CFSM. Hence, for the purpose of behavioural mapping, it is sufficient to consider the behaviours as described by the respective PMs alone.

Let the symbol $\mathcal{F}$ be used to denote the PM of the desired function F, and the symbol $\mathcal{D}$ be used to represent the PM of the available device D. Then, $\mathcal{F}$ is *implementable* by $\mathcal{D}$ if and only if there exist signal transformation functions, t_i and t_o, such that for every path

in $\mathcal{F}$ from the start state to a final state which accepts the input sequence I, thus generating the corresponding output sequence O, there exists a path in $\mathcal{D}$ from a start state to a terminal state, which accepts the input sequence $t_i(I)$, and produces the output sequence O' where $t_o(O') = O$. Hence, a process of search is required to map $\mathcal{F}$ (which is a graph) onto $\mathcal{D}$. Along with the mapping of the states of $\mathcal{F}$, the signals of F also get mapped to the signals of D, with the requisite transformations. These mappings are generated by a variation of the FSM equivalence algorithm of Hopcroft & Ullman (1979). The mapping is not necessarily surjective because, in general, a device may be capable of exhibiting a wide range of functionalities, only a subset of which is sufficient to meet the requirements of F. However, the mapping should be injective.

Based on the results of the mapping, the implementation of the required interface is then determined by using the signal and timing constraints of the device. In addition, if the device is programmable, the programming modes are considered as well.

6. Task management

The D-layer and the different tasks of the S-layer, mentioned in the preceding sections, have to communicate with each other in order to form a coordinated problem-solving system. The details of such inter-task communication are described in this section.

6.1 *Communication of design information*

The inter-task communication has been implemented with the help of a shared blackboard-like data structure. Each design task outputs a partial design on which the relevant constraints are defined. This information can be conceptualized as having four fragments, all of which are stored in the blackboard.

(1) The control and data flow graph (CDFG), that depicts the control and data flow among the functional elements (FEs) of the design, along with the description of the concurrency among the FEs, the constraints imposed on the data flows, and the implementations that have been allocated to each FE.

(2) The constraint networks consisting of the timing constraints and the relations among the various parameters of the design.

(3) The geometrical layout of the elements of the CDFG, in order to facilitate viewing of the partial design and its interactive editing.

(4) The design history, i.e. the search tree that has been explored so far.

These fragments are connected to each other very closely. For example, the leaves of the design history tree are the nodes of the CDFG; timing constraints are defined on sections of the CDFG; the parameters of the nodes of the CDFG are related by constraints, and the layout information is associated with each node and arc of the CDFG.

Each task takes its input data from this shared blackboard, and on completion of the processing stores its outputs in the same place. This partial design information is represented as working memory elements of APS. Some of the tasks of the S-layer are implemented as procedures, and require a different encoding of the data. These procedures convert the

```
If
 task = Startup
Then
 Create subgoal for initiating Simulation
 Create subgoal for initiating Circuit-design
 Create subgoal for initiating Software-design
 Create subgoal for initiating Interface-design
 Create subgoal for initiating Architecture-design
 Create subgoal for initiating Algorithm-design
 Delete task
```

Figure 7. Rule for starting the processing of S-layer.

data available in the blackboard into the required format, and on completion of the task store the results in the blackboard after performing the reverse conversion.

6.2 *Inter-task control flow*

The S-layer is a hybrid expert system, in which rule-based and procedural tasks interact with each other. The sequence among these tasks is managed by a set of task management rules. For example, the first rule to be fired by the system, shown in figure 7, creates the subgoals corresponding to the different tasks of the topmost hierarchy of the task structure. These subgoals are created in the reverse order, so that in the goal stack the last goal created becomes the first goal to be initiated.

Corresponding to each of these task-goals, there exists a rule for creating subgoals for the respective subtasks, or for initiating the relevant procedure. For example, the rule for initiating the search procedure for hardware–software partitioning, shown in figure 8, calls the appropriate procedure, and after its successful termination, deletes the subtask-goal from the goal-stack.

S-layer backtracks to recover from a failure. Failure occurs when a procedure returns a failure, or when there does not exist any rule to solve a subgoal. Unlike OPS5 (Brownston *et al* 1986), the feature of backtracking is inbuilt into the shell (APS) that is used for inferencing. The backtracking mechanism of the shell is used to handle the different situations of failure.

(1) Failure to solve a task, in which case the S-layer backtracks to the previous task.

(2) Failure to solve a subtask, in which case the S-layer backtracks to the previous subtask.

```
If
 task = Architecture-design
 subtask = Hardware-software-partitioning
Then
 Call hardware-software-partitioning-algorithm
 Delete subtask
```

Figure 8. Rule for initiating hardware–software partitioning.

(3) Failure of D-layer to find candidate implementations for a design function, in which case the S-layer backtracks to find an alternative problem decomposition.

In all these cases of backtracking, the user determines the source of the failure. Subsequently, the S-layer backtracks to that point in the design history, and continues its processing from that point onwards, after revising the relevant design decision.

6.3 *Inter-layer communication*

As mentioned earlier, a failure report is created by the S-layer before it invokes the D-layer. The constituents of the failure report are:

(1) Subgoal name, which is the name of the function that could not be solved by S-layer.

(2) Constraint list, which is a list of constraints on the failed function, that have to be satisfied by any device that will solve the subgoal.

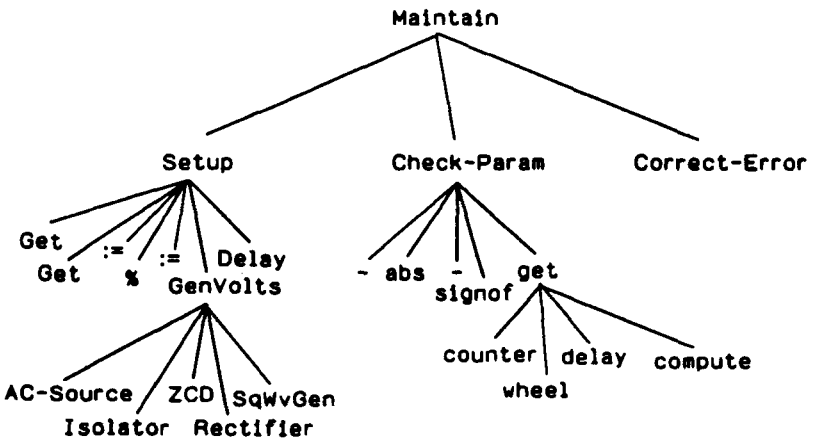
This report is generated by collecting together all control flows and data flows related to the failed subgoal. These control and data flows, along with the function's attributes, constitute the constraints imposed on the function. In addition to the above mentioned control and data flows, the global constraints of the design are also included in the report.

The D-layer is invoked from S-layer by the task management rules, after generating the failure report. Using the name of the failed subgoal as a key, the device database (DDB) is searched for candidate devices that *may* implement the subgoal. The output of the database search is either a single candidate device, or a list of candidate devices, depending on a parameter supplied by the user. Behavioural matching is then performed on the candidates extracted from the DDB.

The output of the D-layer is a list of devices and their interfaces, each of which will implement the failed subgoal. In addition, the D-layer also returns a value, which indicates whether it has *succeeded* or *failed* in its task. If the list of devices is empty (i.e. if no device is found that can implement the failed subgoal) then the D-layer returns *NIL*; else it returns *TRUE*. If the returned value is *NIL*, the S-layer backtracks to find an alternate problem decomposition; else it continues with the present design. On the success of the D-layer, a flag is created in the S-layer's working memory to indicate that the failed subgoal is a PF.

The D-layer is invoked by the S-layer either from the Algorithm Design Task or from the Architecture Design Task. The PF information, created on the D-layer's success, is used by the algorithm design task to continue with its processing. For the purpose of the architecture design task, the information passed is that for device allocation. In addition to these two tasks, the interface design task also requires information about the new implementation, in order to integrate it into the design. In cases where the implementation is classified as a hardware implementation, the information for the interface design task specifies the address constraints, the device initialization process, the device stopping process, and the mapping between the dataflows of the function and of the device. In cases where the implementation is classified as a software implementation, the information for the interface design task specifies the address constraints, the device driving process, and the mapping between the dataflows of the function and of the device.

(a)



(b)

Signal Bindings		
Binding Type	$\mathcal{F}$	$\mathcal{D}$
Input	START	LOAD
	STOP	CHMOD
	not(CLK÷2)	PULSE
Output	WAVE	OUT
Constr	initial cond	GATE = 1
	WIDTH/2	mem(INVAL)@LOAD

State Bindings	
$\mathcal{F}$	$\mathcal{D}$
state f_0	state d_0
state f_1	state d_1
state f_2	state d_2
state f_3	state d_4

(c)

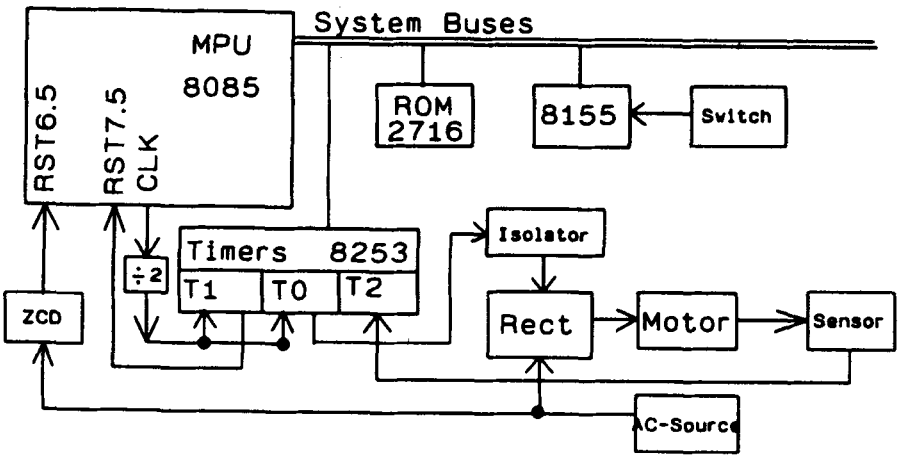


Figure 9. Results for a speed controller problem. (**a**): partial decomposition tree; (**b**): state and signal bindings generated by mapping algorithm; (**c**) target system's circuit.

7. Results

The proposed CAD framework has been implemented on an HP-350 workstation, and applied to design a variety of microprocessor-based systems: (i) speed controller of a DC motor, (ii) different versions of over-current protectors (by polling, by interrupt, multiple-protectors), and (iii) ECG monitoring system. The D-layer has been used to find implementations for the following subgoals of the S-layer: (i) an up-counter by Intel 8253 timer, (ii) square wave generator by Intel 8253 timer, (iii) handshake data input protocol by Intel 8255 port, (iv) multiple interrupt handling by Intel 8259 interrupt controller.

For the speed controller problem, the user specifies the characteristics of the motor, the way it has to be operated (by the phase control method), and the desired speed. The partial functional decomposition tree for this problem is shown in figure 9a. One of the subgoals of the problem is a *Square Wave Generator* (used for operating the motor), for which no implementation is known to the S-layer. The D-layer is invoked for this subgoal, and the state and signal bindings produced by the mapping algorithm, for implementing the function by an Intel 8253 timer operating in mode 3, are shown in figure 9b. The specifications of the device's interface are passed to the S-layer, which continues with the other design tasks. Finally, the target system's circuit (shown schematically in figure 9c) and the software are generated and cosimulated.

In this way, hardware–software codesign of microprocessor-based systems is performed by the S-layer, and the D-layer imparts resilience to the S-layer.

8. Conclusion

In this paper, we have described a new two-layer architecture for computer-aided design. The incorporation of the behaviour modelling feature in the *deep*-layer provides resilience to the overall system, by allowing the *shallow*-layer to fall back on such deep reasoning whenever the shallow knowledge is found to be incomplete. The necessity of deep reasoning has been emphasized by expert system researchers for quite some time. This paper convincingly demonstrates the applicability of such reasoning in CAD systems, and successfully employs the proposed CAD framework to solve real-life industrial problems.

References

Bailey G D, Raghavan S, Gupta N, Lambird B, Lavine D 1991 InFuse – an integrated expert neural network for intelligent sensor fusion. In *Proc. IEEE/ACM Int. Conf. on Developing and Managing Expert System Programs.* pp 196–201

Barstow D 1985 Domain-specific automatic programming. *IEEE Trans. Software Eng.* 11: 1321–1336

Brewer F, Gajski D D 1990 Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. Comput. Aided Design* 9: 681–695

Brewer F D, Gajski D D 1991 An expert system paradigm for design. In *Proc. 23rd DAC* (New York: IEEE Press)

Brownston L, Farrell R, Kant E, Martin N 1986 *Programming expert systems in OPS5* (Reading, MA: Addison-Wesley)

Chou P, Walkup E A, Borriello G 1994 Scheduling for reactive real-time systems. *IEEE Micro* 14: 37–47

Devadas S, Keutzer K, Krishnakumar A S 1991 Design verification and reachability analysis using algebraic manipulation. In *Proc. ICCD-91* (New York: IEEE Press) pp 250–258

Haralick R M, Shapiro L G 1979 The consistent labelling problem, part 1. *IEEE Trans. Pattern Anal. Machine Intell.* 1: 173–184

Harel D 1987 Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8: 231–274

Hopcroft J E, Ullman J D 1979 *Introduction to automata theory, languages and computation* (Reading, MA: Addison-Wesley)

Hu X, D'Ambrosio J G, Murray B T, Tang D 1994 Codesign of architectures for automotive powertrain modules. *IEEE Micro* 14: 17–25

Hyvonen E 1992 Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artif. Intell.* 58: 71–112

Jullig R K 1993 Applying formal software synthesis. *IEEE Software* 10: 11–22

Kalavade A, Lee E A 1993 A hardware software codesign methodology for DSP applications, *IEEE Design Test* (Sept.): 16–28

Kambhampati S, Cutkosky M R, Tenenbaum J M, Lee S H 1993 Integrating general purpose planners and specialized reasoners: Case study of a hybrid planning architecture. *IEEE Trans. Syst. Man Cybern.* 23: 1503–1518

Keller R, Baudin C, Iwasaki Y, Nayak P, Tanaka K 1990 Compiling redesign plans and diagnostic rules from a structure/behaviour device model. Tech. Rep. FIA-90-07-01

Keravnou E T, Washbrook J 1989 What is a deep expert system? An analysis of the architectural requirements of second generation. *Knowledge Eng. Rev.* 4: 3

Keuneke A 1991 Device representation: The significance of functional knowledge. *IEEE Expert* (April)

Kumar S, Aylor J H, Johnson B J, Wulf W A 1993 A framework for hardware software codesign. *IEEE Computer* (Dec.): 39–45

McDermott J 1982 R1: A rule-based configurer of computer systems. *Artif. Intell.* 19: 39–88

McFarland M C, Parker A C, Camposano R 1990 The high level synthesis of digital systems. *Proc. IEEE* 78: 301–318

Micheli G D 1994 Computer aided hardware software codesign. *IEEE Micro* 14: 10–16

Mitchell T M, Steinberg L I, Shulman J S 1985 A knowledge-based approach to design. *IEEE Trans. Pattern Anal. Machine Intell.* 7: 502–510

Mitra R S 1995 *Hardware software codesign of microprocessor based systems: A knowledge based framework.* PhD thesis, Indian Institute of Technology, Kharagpur

Mitra R S, Guha B, Basu A 1993 Rapid prototyping of microprocessor-based systems. In *Proc. Int. Conf. on Comput. Aided Design (ICCAD-93)* (New York: IEEE Press) pp 600–603

Mitra R S, Kumar M, Basu A 1994 Design of microprocessor-based systems: A knowledge-based approach. *IEEE Trans. Ind. Electron.* 41: 352–360

Mitra R S, Roop P S, Basu A 1996 A new algorithm for implementation of design functions by available devices. *IEEE Trans. Very Large Scale Integrated Syst.* 4: 170–180

Smailagic A, Siewiorek D P 1993 The VuMan 2 wearable computer. *IEEE Design Test* (Sept.): 56–67

Srivastava M B, Brodersen R W 1991 Rapid-prototyping of hardware and software in a unified framework. In *Proc. Int. Conf. on CAD (ICCAD-91)* (New York: IEEE Press) pp 152–155

Steinberg L I 1987 Design = top down refinement plus constraint propagation plus what? In *Proc. IEEE Systems Man and Cybernetics Conf.* (New York: IEEE Press)

Tseng C, Siewiorek D P 1986 Automated synthesis of data paths in digital systems. *IEEE Trans. Comput. Aided Design* 5: 379–395