A Design Utility Manager[1]

Technical Report CRI-85-25
November 18, 1985

David W. Knapp and Alice C. Parker

[1]This research was supported by the National Science Foundation under computer engineering grant #ECS-8310774

A Design Utility Manager[1]

Abstract

In this paper we present a software package which manages the digital design process. The design process is modelled using a planning paradigm. Under this paradigm design is seen as a process in which abstract models of operators are applied to abstract models of design states in a simulated or planning space, until a sequence of operators has been constructed to some depth. The sequence, or *plan*, is then executed, or carried out, in an execution space. This execution is monitored for violation of expectations; if violations occur, control is returned to the planner. The planner forms part of the USC ADAM (Advanced Design AutoMation) system: at the time of writing it is capable of constructing plans and causing them to be executed. The knowledge base of the planner is populated with register transfer level (RTL) concepts, and it can be populated with other knowledge sets.

Categories: 8,16

---

[1]This research was supported by the National Science Foundation under computer engineering grant #ECS-8310774

Submitted to the 23rd IEEE/ACM Design Automation Conference (June 29, 1986)

1. Abstract

In this paper we present a software package which manages the digital design process. The design process is modelled using a planning paradigm. Under this paradigm design is seen as a process in which abstract models of operators are applied to abstract models of design states in a simulated or planning space, until a sequence of operators has been constructed to some depth. The sequence, or *plan*, is then executed, or carried out, in an execution space. This execution is monitored for violation of expectations; if violations occur, control is returned to the planner. The planner forms part of the USC ADAM (Advanced Design AutoMation) system: at the time of writing it is capable of constructing plans and causing them to be executed. The knowledge base of the planner is populated with register transfer level (RTL) concepts, and it can be populated with other knowledge sets.

2. Introduction

2.1. The ADAM System

The USC ADAM system is designed to unify a number of design automation (DA) programs into a single framework. The main areas of effort are a set of custom layout tools, an expert system for the design of testable circuits, a knowledge-based synthesis system, and a database which provides a common representational scheme for designs, programs, and constraints. Within ADAM, all design data, procedures, and rule sets are treated as objects, communicating with one another by means of messages. The overall structure of ADAM is described in [6].

The goals of the knowledge-based synthesis system, of which the design planner is a part, are to produce correct implementations representing a range of tradeoffs, to allow varying degrees of user interaction, to allow design to proceed incrementally starting from a partially-complete initial design, and to meet several kinds of constraints. The ADAM synthesis subsystem uses both hard-coded and knowledge-based techniques to achieve these goals.

The ADAM planner is used to manage design activities. First a plan is constructed,

which details the activities and utilities that will be used; then the plan is executed. If execution of the plan is successful, a design that is correct and which meets input constraints is produced; otherwise the planner attempts to construct a new plan. The planner is in control of the execution process as well as the planning process.

This paper proceeds as follows: first, some salient properties and problems of the digital design domain are reviewed, and some related research is discussed. Next, descriptions of the planning, estimation, and execution processes performed by the ADAM system are given. Following that, the structure of the planner is discussed, and examples from the planner's knowledge base and rule set are given. The current status of the planner is then given, with a description of the test cases so far run. Finally, a short summary of future plans for the planner and its place in ADAM are given.

2.2. The Digital Design Process

The digital design process can be characterized by complexities of two basic kinds. First, there are complexities of sheer magnitude, such as gate counts; and second, there are complexities introduced by qualitative choices, e.g. between technologies and implementation styles. These qualitative choices often interact strongly. Moreover, their interdependencies are often cyclic. This means that not only do the outcomes of design decisions depend on the particular set of specifications, objectives, and constraints, but the ordering of decisions can also be permuted, with different results.

Furthermore, activities that are crucial in implementing one design may be irrelevant to the completion of another. For example, in MOS monolithic logic a finite state machine with a small number of states can be implemented very cheaply using one-hot state encoding and dynamic storage; in such a situation, techniques of state minimization appropriate to TTL SSI are a waste of time and silicon. It is sometimes more important to produce a design quickly, almost without regard to its efficiency, than it is to produce a highly efficient design. In the former situation, resort might be had to a silicon compiler and a simplified controller/datapath design style, whereas in the latter case a more complex multipath/multicontroller architecture and more expensive design techniques might be indicated. In another case, a program tuned only to a specific class

of applications may be applicable, such as a digital filter synthesizer.

Thus a fixed line of reasoning or sequence of activities will often lead to inferior designs, because important alternatives may not be investigated, inappropriate techniques may be applied, or relevant information may not be examined.

Unified design systems that attempt to maintain flexibility in the face of different optimization criteria and target system requirements face stringent software engineering requirements. In a system of N modules, the number of possible interfaces is $O[N^2]$. Worse, the number of possible chains of N operators is $O[N!]$, and the choice between sequences tends to be data-dependent. ADAM incorporates two strategies to deal with these problems:

1. ADAM uses a unified representation of design data, which ensures that only $O[N]$ interfaces need be constructed. This representation and its database implementation have been described elsewhere [7], [1].

2. ADAM uses planning techniques borrowed from artificial intelligence research to construct operator sequences at design time.

2.3. Related Research

The original unified DA system is CMUDA [4]; other systems that attempt to span the design process are MIMOLA [8] and CADDY [11]. These, however, rely on a more or less fixed path through the design activities. A recent enhancement to CMUDA [5] comes closest to the ADAM planner by the employment of a knowledge base of scripts. The Rutgers group [14] uses planning in a different sense than ADAM, by concentrating on the propagation of constraints within a design, rather than as an activity scheduling formalism. A system based on descriptions of hardware modules is reported in [15]; it translates from DDL to layout using knowledge similar to parts of the ADAM knowledge base.

3. The ADAM Planner

The *design planner* is an expert system with its own set of planning rules. It builds a design plan by concatenating members of a set of possible analysis and synthesis tasks and tools, including clocking scheme synthesis, component selection, critical path location, and area estimation. The planner uses estimated statistical factors in order to guide its choices of tasks where more than one option is possible. The planner uses knowledge about the design process declaratively represented by a network of frames; these frames contain knowledge about the design of hardware including taxonomy and methodology. Plans and histories alike are represented as annotated tours through the knowledge graph.

The planner is designed to oversee a suite of knowledge-based and hard-coded utilities. Specifically, the problem addressed by the ADAM planning engine is that of finding a sequence of design activities that will, when executed, result in the transformation of a specification at one level to a completed design at a lower level within a reasonable amount of time. The activities may be computationally expensive and the data upon which they operate tend to be highly complex. Hence activities should not be invoked unless they have a high probability of being useful in a sequence which produces the desired result. The planner allows ADAM to take into account the dependencies generated by individual specifications and partial designs, and to sequence design activities accordingly. Hence ADAM can construct sequences of design activities which are not explicitly coded into the system, but are constructed in response to the needs of a particular set of specifications and constraints.

Planning [12], [13], [17] is an activity whereby the operation of a system is simulated in order to economize search or avoid irrecoverable errors. At a minimum there are two spaces that must be considered: a *planning* space and an *execution* space. The execution space is populated with *operators* (in the case of ADAM, design utilities such as hardware allocators, microprogram generators, and dataflow graph optimization tools) and *states* (graphs and collections of symbols that collectively describe the design itself, e.g. schematics, dataflow graphs, timing and control graphs, and physical layouts). The planning space is populated with abstract representations of the execution space

operators and states. States in the planning space are characterized by collections of assertions about more or less incomplete designs. An example of an assertion is the list (**exist PLA-Layout it**), which states that a PLA layout of some circuit **it** exists. Abstract representations of operators are characterized by *preconditions* and *postconditions*: the preconditions express the conditions that must pertain for the operator to be applicable to a given state, and the postconditions express the conditions that will pertain after the operation has been applied (i.e. on the sink state). Preconditions are represented by sets of assertions that must be matched for the operator to be considered applicable, and postconditions by the sets of assertions that are added and deleted by the application of the operator.

For example, suppose the task is to implement a combinational logic element. The specification is given as a set of Boolean equations; the ultimate result is to be a layout (physical description) consisting of layer, width, and path information. Two ways to achieve this result are classical two-level Boolean minimization followed by standard-cell layout, and PLA minimization and synthesis. The preconditions for both ways are that the logic equations exist; the postconditions of both are that the logic equations exist, and that the layout also exists. This is shown in Fig. 3-1.

In the ADAM planner, a plan is constructed by forward chaining from the initial state. Forward chaining is done by matching the precondition of an operator to the initial state, and creating a new state at the sink end of the operator arc. This new state's assertions are a copy of the initial state's assertions with the modifications specified by the operator's postcondition. If the second state is the same as the goal state, the process is finished; otherwise the process is repeated. Once a goal state has been reached the plan is complete. It can be executed at that time by applying the execution-space operators to the execution-space states. Backward chaining is a similar method, except that the goal state is the one from which operators are chained in an effort to reach the initial state. Backward chaining was not used for the ADAM system because of two considerations:

1. Incremental execution (i.e. at planning time) is inefficient in a backward-chained system because the first operation executed is the last one planned.

Where there is a significant probability of misplanning or operator failure the loss of efficiency becomes severe.

2. Most design problems have many acceptable solutions. It is entirely possible that many hypothetical designs could meet a given specification, but without knowing their details, choosing a single goal state from among them cannot be done intelligently.

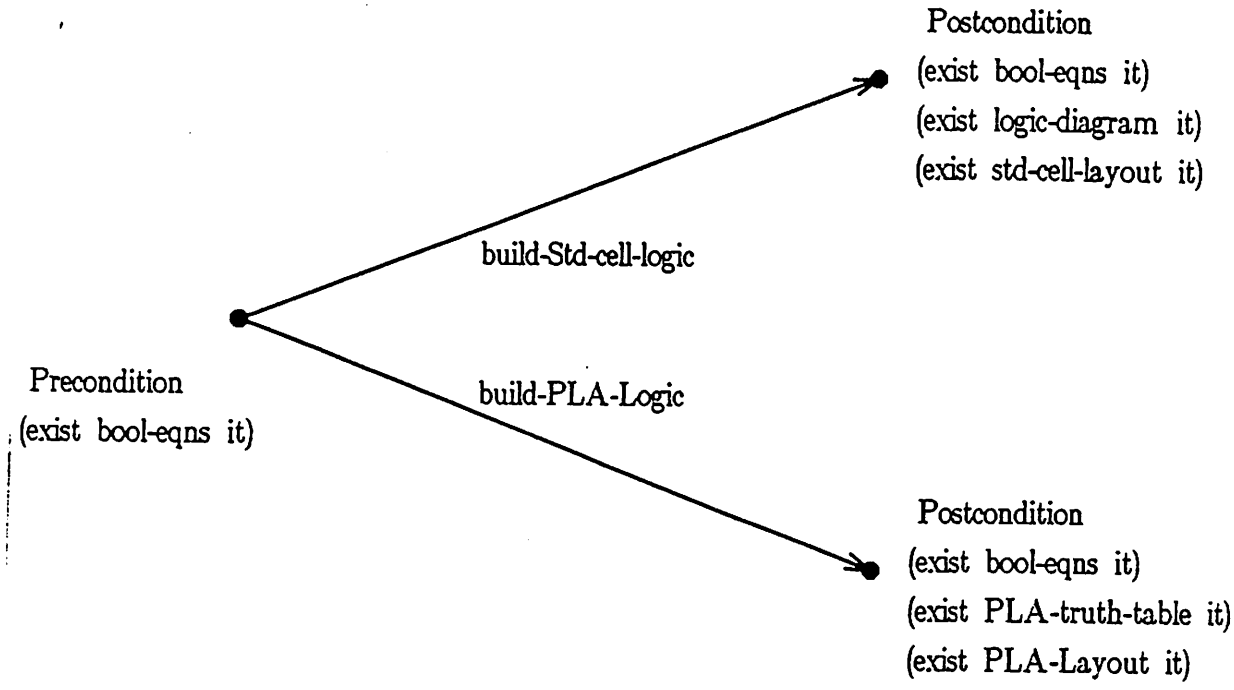

Figure 3-1: Illustration of Planning

3.1. Sequencing Operators

We will now describe the way in which the planner constructs a plan. Given a 'current state' of the design, represented as shown in Fig. 3-1 by a collection of assertions about the design,

1. Find the set of activities that can be applied to the state. Add them to the plan graph as arcs directed away from the current state.

2. The state of the design that results from applying each activity is constructed from the initial state, less any assertions in the 'delete set' (preconditions no longer true) of the activity, plus any assertions in the 'add set' (postconditions not previously true).

3. The most promising of the set of leaf states (i.e. states with no out-arcs) is chosen as the next current state.

In the ADAM design planner the selection of the 'most promising' state is guided by two considerations:

1. Operators that would add no new facts to the design state are not considered even though they might be applicable.

2. Numeric estimates of the advisability of the applicable operators given the design state and the planner's current goals are generated and compared.

The sink state of the most promising operator is then made the current state and chaining proceeds. The current search strategy restricts itself to leaf states generated in the last iteration unless those leaf states are all dead ends, at which time other leaves are considered.

3.2. Estimation and Goal Management

Estimation of the properties of the design state is an important part of the planning process, since so many planner decisions are design dependent. A planner that was only capable of matching symbols for the purpose of finding what activities were possible at a given point would have no way to prune the search space, and would tend to produce poor results because it would have no way to compare the merits of different implementations. In the ADAM design planning engine these property estimates fall into five basic classes:

1. A rough measure of the global advisability of applying an activity to the current state. This is fast and does not rely upon execution-space details; instead it only uses symbols found in the planning-space state.

2. Rough estimates of four numeric quantities: design time, area, speed, and power consumption. These estimates rely only upon symbols found in the current state and upon simple abstractions of its execution space counterpart, e.g. the number of nodes in a dataflow graph.

3. Estimates of the four quantities based on trial implementations. In such cases, a plan is constructed and carried out in the execution space as an effort to try to define the range of possible implementations.

4. Estimates of the four quantities based on analytic and statistical models. These vary in complexity and accuracy, and rely on procedures called in by the planner.

5. Estimates of the reliability of other estimates.

The first three classes of estimation are currently implemented. A statistical estimator for area exists but has not yet been integrated into ADAM.

The estimators used by ADAM's planning system are chosen so as to be as close to *monotonic* as possible. Monotonic estimators are not necessarily accurate; they merely satisfy the constraint that $\mathbf{EST_p}[\mathbf{A}(x)] > \mathbf{EST_p}[\mathbf{A}(y)]$ iff $\mathbf{P}(x) > \mathbf{P}(y)$, where x and y are two designs, the function $\mathbf{P}$ is the projection of those designs onto a property space (e.g. that of area), the function $\mathbf{EST_p}$ is the estimator function of the quantity $\mathbf{P}$, and the function $\mathbf{A}$ is the abstraction mapping from the execution space to an abstract planning space. Such an estimator is useful for comparing two or more design alternatives regardless of its absolute accuracy, because the choice of an alternative with a greater or lesser $\mathbf{P}$ would remain unchanged in the presence of more information than was available in the abstract space.

It can easily be shown that a combination of monotonic estimation and binary search results in an overall search complexity of $\mathbf{O}[n\log n]$, where n is the maximum depth of the plan and the outdegree of each plan node is bounded by some k. For this reason, monotonicity is theoretically important as a measure of estimator quality.

3.3. Mixed Planning and Execution

In the preceding discussion, plans have been described without reference to the execution space. However, the line between planning and execution is not distinct, and planning and execution become mixed. For example, suppose the purpose of the planner is to find the fastest possible implementation of a behavior within the constraint of a fixed target technology. The plan is constructed using class 1 estimators. The only

information available to such estimators at the time of the completion of that plan is the collection of assertions at the last node of the plan. Clearly that is not enough.

Suppose, for example, that a critical-path algorithm was scheduled somewhere in the plan. It would, when executed, mark all the nodes on the critical timing path of the design. This information is crucial to the timing estimator which is used to select a design style [16], and consequently to the choice of an appropriate synthesis package for that style, but it is not available until the planned **critical-path-marking** step of the plan is actually carried out. The only thing available before that happens is the assertion **(timing-graph critical-path-marked true)** but the details of the path and the actual marking do not exist until the algorithm has been executed.

Secondly, many activities are not atomic. That is, each activity can be decomposed into further sub-activities. Hierarchical planning is a recognized area of artificial intelligence research [13], [12]. When it is combined with actual plan execution, the detailing of a hierarchical plan begins to take on the attributes of execution; in fact the ADAM planning engine regards the detailed expansion of a hierarchical activity as equivalent to execution of a procedure. The difference is that expanding a plan which represents hierarchical activities only adds and deletes symbols (assertions) in an abstract space, whereas procedurally encoded activities affect the execution space and hence the concrete design itself.

Another reason planning and execution become mixed is plan failure. There are two ways in which a plan could fail. First, the plan could be executed to the end, but produce an inferior design, or one that did not meet constraints. Second, plans could fail because of inadequacies in the knowledge representation or application programs which become apparent at run time. Both of these kinds of plan failure are addressed by the ADAM planner, which at this time simply constructs a new plan and tries it when an execution dead-end is detected.

4. Structure of the Planner

The ADAM planning system structure is shown in Figure 4-1. There are two main sources of declaratively encoded knowledge in this system: the planner rules and the design knowledge base. Planner rules are if/then productions constituting knowledge about how design plans are constructed and the structure and meaning of design knowledge base constructs. The design knowledge base is organized as a collection of frames [9], [3] which collectively describe design activities, styles of hardware, and functional classes of hardware. The planning engine is a custom coded forward chained rule interpreter.

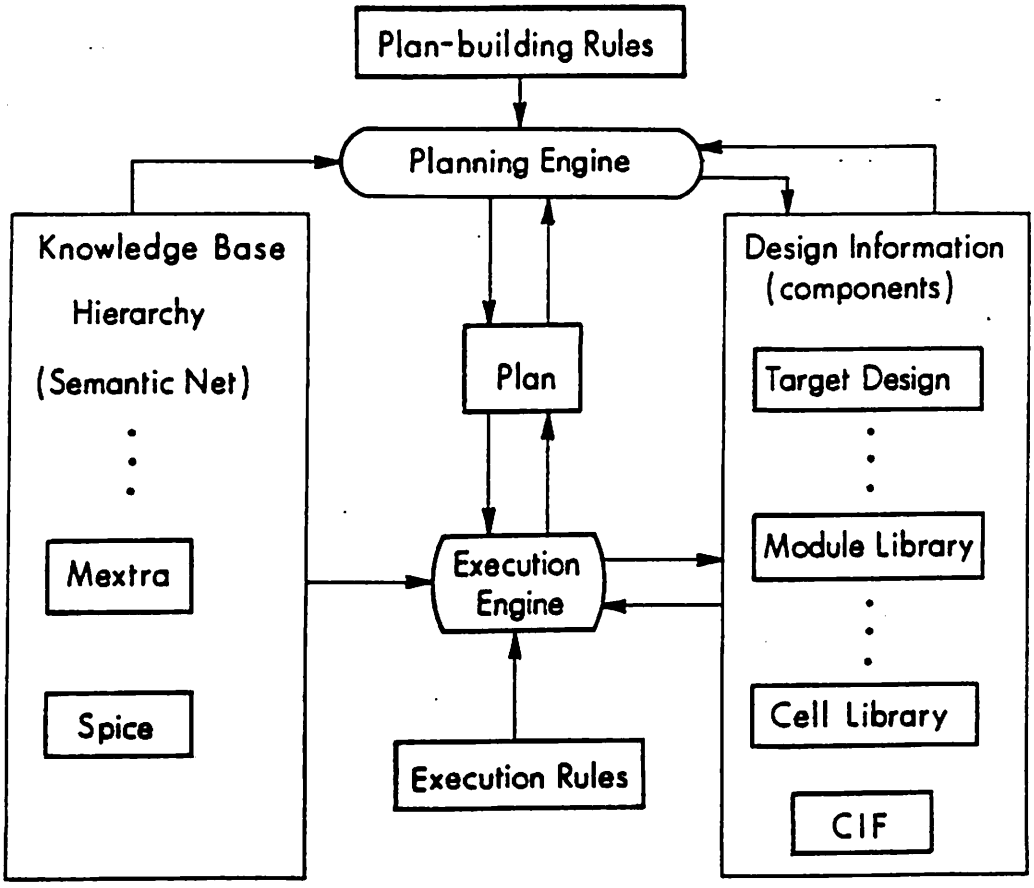

Figure 4-1: The ADAM planning engine

4.1. Planner Rules

Planning engine functions which allow the planner to construct plans are divided into nine classes. Each class (or expert) has its own subdivision of the planner rule set and its own local collection of assertions (data set). The experts communicate by means of messages, which are passed as a result of rule firing (i.e. it is possible for a rule's then-part to consist of mixed assertions and messages to other experts and/or primitive objects). The planning engine experts are described in Table 4-1.

name	rules	area of expertise
planner	34	building plans (overseer)
extender	9	concatenation of operators onto plans
expander*	7	expansion of hierarchical tasks and task execution
frame-expert	7	general frame system navigation
frame-extend	8	frame system navigation for the purpose of concatenating operators
frame-expand	6	frame navigation for execution and hierarchy expansion
donex*	14	determine completeness of plans
evaluator*	5	calculation of numeric properties of designs
estimator*	1	used in constructing estimators on the fly

* Partial rule set, augmented at run time by rules taken from frames.

Table 4-1: Planner Rule Set Divisions as of 11-10-85

Planner rules are domain independent in the sense that they only concern the syntax of plans and frames. This allows planning expertise to be separated from the specific digital system expertise in the collection of frames; in theory it is possible to implement design planners for many kinds of application domains merely by changing the contents of the frame set. It is also possible to build smarter planning strategies into the planner rule set without modifying the domain expertise: for example, the depth-first successive-

approximation strategy could be switched to the A* strategy [2] with only rule changes and the addition of the lookahead function required by A*.

Two typical rules are shown in Fig. 4-2. Each rule is composed of three parts: a **type**, in both of these cases **F1**, which controls the firing of the rule, an **if-** part, and a **then-** part. The type flag **F1** means that the rule can fire only once in a given context without explicitly being reset. The **if-** part describes the conditions under which the rule can fire; in the first example rule, that part consists of four clauses. The first clause states that no assertion can exist of the form **(prime leaf ?leaf)**, where **?leaf** will match any name; meaning that there can be no leaf considered to be a **prime** candidate for expansion. The other three clauses must be matched; there must exist at least one **?leaf** (which can have any name) which is both a **leaf** and which is **done**, and the **goal** must be **expand**. If all of these things are true, the **then-** part of the production is evaluated. This consists of two subclauses: the first states that the **?leaf** is to be declared **terminal**, and the second results in a request being passed to an expert **manager** to **count** the **terminal** leaves. **(terminal ?leaf)** is simply an assertion added to the data set, whereas the **manager** will first compute the required number and then assert it in the data set.

The second rule specifies that some **?leaf** must exist which is both a **leaf** and **done**, but which is not **dead**; only variables which match all of these criteria will be declared **dead** and passed to the **donex** (done expert) to determine if they represent finished designs.

```
(F1 ((no (prime leaf ?leaf)) (goal expand) (leaf ?leaf) (done ?leaf))
    (and (terminal ?leaf) (ask manager count terminal)))
        ; if there is no prime candidate leaf, and the current goal
        ; is expansion, and at least one leaf represents a complete
        ; design, then assert that the finished leaves are terminal
        ; and ask that the manager count the terminal leaves.

(F1 ((nor (dead ?leaf)) (goal extend) (leaf ?leaf))
    (and (ask donex finished ?leaf) (dead ?leaf)))
        ; if the goal is extension, and a set of leaves is not marked
        ; dead, then mark all such leaves dead and ask if the designs
        ; represented by those leaves are complete.
```

Figure 4-2: Two Rules Taken Verbatim from the Planner

4.2. The Design Knowledge Base

The design knowledge base is a collection of frames containing information about VLSI design, organized into three classes.

1. *Task* frames describe design activities that can be carried out. An example of a task frame is the description of a pipelined hardware allocator.

2. *Hardware* frames describe ways to implement particular functions and classes of functions in hardware. An example of a hardware frame is one that describes controllers.

3. *Style* frames describe variations on basic hardware structures. An example of a style frame is one that describes various kinds of adders, such as carry-lookahead and carry-bypass.

A simplified pair of frames from the current implementation of the planner's knowledge base are shown in Figs. 4-3 and 4-4. The frame of Fig. 4-3 is called **sehwa-fast**, and it describes an operator or **task**, which in this case is a Lisp program named Sehwa that does pipelined datapath synthesis [10]. It has five subfields: **invariant**, **add**, **advisability**, **subtasks**, and **hardware**. The invariant field describes the assertions that must exist for the Sehwa program to be applicable: there must be a design focus of attention (**dfoa**), which may have any name (**?it**); the leading question mark implies a match variable, such that any string will be matched and unified. There must also be a dataflow graph **?itsdfg** and a library family **?library-family**, which is a set of components the allocator can choose from.

Five assertions will be **added** to the sink state of this operator: one to the effect that the design will be pipelined, one that a control table will exist, as well as assertions of the existence of a schematic (RTL) structure graph, a timing graph, and a set of relations expressing the relationships between dataflow, structure, and timing graphs.

The **advisability** field represents a nonprocedural, limited-information estimator of the overall advisability of using this operator in a given situation. The advantage of this approach is that no particular fact or set of facts need be present for the estimator to work; therefore the estimator can be constructed without precise knowledge of the situations in which it will be applied. Moreover, the estimator is designed to operate on

unexecuted plans, i.e. on collections of assertions such as the ones that populate **invariant** and **add** sets. This has the important advantage that plans can be heuristically guided on the basis of hypotheses (e.g. that end-product area will be a problem) until information that confirms or denies those hypotheses is produced by actual plan execution.

The estimator is much like a production system: it consists of a set of rule-like clauses, each of which has implicit **if-** and **then-** parts. The **if-** parts enable the transformations in the **then-** parts if and only if they are matched in the design state. Hence the first of these specifies that if the design is already supposed to be pipelined, and the design time and the speed of the design (**dtime** and **ctime** respectively) are both important, then the number **advis** should be incremented by 20. The next production specifies that the existence of a timing constraint graph makes this operator even more preferable. As of this writing this is the only estimator directly attached to the sehwa-fast frame. Another, more elaborate estimator that operates on partially executed plans is attached to the frame of Fig. 4-4.

The **subtasks** field in this case is a pointer to a Lisp program description frame. That frame contains details and rules describing the actual invocation of the Sehwa program under Unix; the intention of such frames is to make it possible to have numerous communication and calling disciplines, from a simple function call within Lisp to the starting of an independent coroutine process under Unix.

The **hardware** field is a pointer to a **hdwe** frame. This frame has more details on the estimation of hardware properties, on the various other styles and tasks related to the **task** frame, and so on. Part of that frame is shown in Fig. 4-4.

The **hdwe** frame of Fig. 4-4 describes a particular kind or class of hardware. The name of the frame is **generic-rtl**, meaning that it is a description of register-transfer-level hardware of unspecified style and structure. The **level** field is used to encode this fact in machine-usable form. The **to-build** field is an unorganized (**random**) collection of task frame names: this organization puts the most burden on the planner because no ordering

```
(task sehwa-fast        ; pipelined allocator
  (invariant (          ; all of the following must exist for this to apply
    (dfoa ?it)          ; a particular hardware unit. Any name fits
    (datfl-graph ?it ?its-dataflow-graph)       ; its dataflow graph
    (libra-famly ?library-family)))             ; a library family
  (add (and             ; all of the following will be added to the design
    (pipelined ?it)     ; it will be pipelined
    (ctrl-table ?it %ctbl)              ; there will be a control table
    (struc-graph ?it %stg)              ; there will be a structure
    (timng-graph ?it %tmg)              ; and a timing graph
    (datfl-timng-struc-binds ?it %dfsb))) ; and a set of ternary bindings
  (advisability (       ; type 1 estimator of advisability for this task
    (((pipelined ?it) (tight ctime) (tight dtime))      ; if all three exist
      (+ advis 20))                                     ; then add 20 to advisability
    ((timng-const-graph ?it ?itstmcg) (+ advis 10))     ; timing constraints
    ((struc-graph ?it ?itssg) (- advis 6))              ; structure exists
    ((timng-graph ?it ?itstmg) (- advis 15))            ; timing graph exists
    ((tight dtime) (+ advis 9))                         ; design time is tight
    ((tight ctime) (+ advis 14))                        ; speed is important
    ((or (tight area) (tight power)) (- advis 7))))     ; area, power penalty
  (subtasks (lisp sehwa-fast-lisp))     ; how it is done, by invoking lisp
  (hardware (generic-rtl)))             ; pointer to related tasks, analysis
                                        ; tools, and descriptions
```

Figure 4-3: Example Task Frame

or mutual exclusion information is given, but must be discovered.

There are two estimators attached to the frame in Fig. 4-4. They are different in two ways from the estimator of Fig. 4-3: in their syntax and in the level of detail they are capable of accommodating. The syntax of the estimator rules is identical to that of the planner's rules. They are interpreted by the same mechanism. The estimator is built just before it is invoked by concatenating rules from all hardware, style, and overlay frames that are applicable to the given situation, as well as a set of rules that are always appended to this class of estimator. The result is a set of rules that has been built using rules from several sources; hence the rules for **area**, the first three productions, represent only part of the rule set of the estimator that is actually interpreted. The result of estimator rule set interpretation may be the evaluation and reporting of numbers, e.g. by the **then-** part (**ask counter count nodes ?itsdfg**), which results in a message being passed to an actor named **counter**, which returns the number of nodes in the dataflow graph in the form of an assertion in the estimator's data set. Numeric expressions are evaluated and asserted by a postfix interpreter.

```
(hdwe generic-rtl
  (level (rtl))
  (to-build (random (maha sehwa-fast sehwa-slow setlib-fast setlib-cheap
                     estb-ctrl-random estb-ctrl-PLA estb-ctrl-mpg
                     build-ctrl)))
  (Estimators
      (area
           ((F1 ((dfoa ?it) (datfl-graph ?it ?itsdfg))
                (ask counter count nodes ?itsdfg))
            (F1 ((dfoa ?it) (datfl-graph ?it ?itsdfg)
                 (number ?itsdfg nodes ?n))
                (ask evaluator set-me area (?n 80000 *)))
            (F1 ((dfoa ?it) (struc-graph ?it ?itstg))
                 (ask counter count modules ?itstg)))
       dtime
           ((F1 ((nor (number ?itsdfg nodes ?n))
                 (dfoa ?it) (datfl-graph ?it ?itsdfg))
                (ask counter count nodes ?itsdfg))
                          . . .                          ]
```

Figure 4-4: Example Hardware Frame (Simplified)

5. Current Status

As of this writing (11-6-85), the system is capable of constructing and executing plans under the direction of class 1 estimators. It is written in Franz Lisp and runs under Berkeley 4.2 Unix and SUN 1.4 Unix. It consists of approximately 3000 lines of uncompiled and unoptimized code including the current planner rules and knowledge base. The rate of rule firing averages about 0.7 rules/second, and the rate at which operators are added to the plan is about a fifteenth of that (Table 5-1). The majority of that time is spent in the unification algorithm which matches rule condition parts to the contents of data sets. These results do not include the execution time of the procedural utilities and activities, not all of which are interfaced to the execution monitor at this time. They do, however, include the time necessary to invoke the procedures. This rate is expected to stay roughly constant as both the plan and the knowledge base grow, but to decrease roughly linearly as the number of planner rules increases.

The results given in table 5-1 are for an otherwise unloaded SUN model 2/120 workstation, with 2 megabytes of semiconductor RAM, running SUN 1.4 Unix. The plans so constructed were not carried out in detail, so the operation of the utilities is not included in the overall run time, although the planner's supervision overhead for the

Test Case #	goals	tasks scheduled	runtime (seconds)
1	maximize speed	1. choose fast library 2. exhaustive pipelined allocator 3. use microprogrammed control 4. build controller	279
2	minimize area and power	1. choose cheap library 2. run nonpipelined allocator 3. choose PLA controller 4. build controller	333
3	minimize power and design time maximize speed	1. choose cheap library 2. run fast pipelined allocator 3. choose microprogram controller 4. build controller	287
4	minimize design time and area	1. decide PLA control 2. choose cheap library 3. run nonpipelined allocator 4. build controller	272

Table 5-1: Run time results for the ADAM planner on SUN 2/120

utilities is included. Because the plans were not carried out in detail, only type 1 estimators were used in these plans. They therefore represent explorations more than final implementations. The more sophisticated type 2 estimators, although present and working, did not figure in constructing plans but only in evaluating the terminal states of the plans.

5.1. Future Work

In the immediate future several further capabilities are to be added to the planner, including:

- interfacing rules for the several activities and utilities so that they do not have to be invoked by hand,
- integration with the ADAM database management system so that designs, specifications, frames, rule sets, and procedures can be accessed in a uniform way,

- addition of rules to compute and allocate budgets,
- implementation of task-subtask hierarchies,
- addition of design-time budgeting, monitoring and interrupt capabilities so the planner can abort programs that exceed their run-time budgets, and
- addition of rules to cope with plan failure.

There are many other capabilities that are envisioned for the ADAM planner, such as the ability to deal with partitioned and hierarchical designs, interfacing with the design-for-testability system, interfacing with richer and more elaborate activities and utilities, and the expansion and validation of ever-growing sets of frames and rules. Because ADAM is a unified system that has the capability for expansion built into it, it is not expected that it will ever be finished; rather it will only grow more capable with the passage of time.

References

[1] H. Afsarmanesh, D. Knapp, D. McLeod, and A. Parker.
An Extensible Object-Oriented Approach to Databases for CAD/VLSI.
In *Proceedings of the 11th International Conference on Very Large Data Bases.* VLDB Endowment, 1985.

[2] Avron Barr and Edward Feigenbaum.
The Handbook of Artificial Intelligence.
William Kaufmann, 1981.

[3] Ronald J. Brachman.
I Lied About the Trees.
The AI Magazine 6(3), 1985.

[4] S. W. Director, A. C. Parker, D. P. Siewiorek, D. E. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
IEEE Transactions on Circuits and Systems CAS-28:634-645, July, 1981.

[5] S. W. Director,
1985
Presentation at the 1985 CANDE Workshop.

[6] John Granacki, David Knapp, and Alice Parker.
The ADAM Advanced Design AutoMation System: Overview, Planner, and Natural Language Interface.
In *Proceedings of the 22nd Design Automation Conference.* 1985.

[7] Knapp, D. and Parker, A.
A Unified Representation for Design Information.
In *CHDL-85 Conference Proceedings.* North-Holland, August, 1985.

[8] Marwedel, P.
The MIMOLA Design System: Detailed Description of the Software System.
In *16th Design Automation Conference Proceedings*, pages 50-63. ACM SIGDA, IEEE Computer Society-DATC, June, 1979.

[9] M. Minsky.
A Framework for Representing Knowledge.
The Psychology of Computer Vision.
McGraw-Hill, 1975.

[10] Nohbyung Park.
Synthesis of High-Speed Digital Systems.
PhD thesis, University of Southern California, 1985.

[11] Wolfgang Rosenstiel and Raul Camposano.
Synthesizing Circuits from Behavioural Level Specifications.
In *Proceedings of CHDL-85*, pages 391-403. North-Holland, 1985.

[12] Earl D. Sacerdoti.
A Structure for Plans and Behavior.
Elsevier Scientific Publishing, 1977.

[13] Mark J. Stefik.
Planning With Constraints.
PhD thesis, Stanford University, 1980.

[14] Louis I. Steinberg and Tom M. Mitchell.
A Knowldege Based Approach to CAD: The Redesign System.
In *Proceedings of the 21st Design Automation Conference*, pages 412-418.
IEEE/ACM, IEEE, 1984.

[15] Shigeru Takagi.
Design Method Based Logic Synthesis.
In *Proceedings of CHDL-85*, pages 49-63. North-Holland, 1985.

[16] Thomas, D.
The Design and Analysis of an Automated Design Style Selector.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University,
Pittsburgh, Pa., April, 1977.

[17] Wilensky, Robert.
Planning and Understanding.
Addison-Wesley, 1983.