January 1985

Report No. STAN-CS-85- 1035
Also numbered: HPP-85-1

RESIDUE
A Deductive Approach to Design Synthesis

by

J. J. Finger
Michael R. Ccncscrcth

Department of Computer Science

Stanford University
Stanford, CA 94305

Stanford Heuristic Programming Project
Memo HPP-85-l

January, 1985

RESIDUE

A Deductive Approach to Design Synthesis

J. J. Finger
Michael R. Genesereth

Submitted to IJCAI-85

COMPUTER SCIENCE DEPARTMENT
Stanford University
Stanford, California 94305

Contents

Abstract

We present a new approach to deductive design synthesis, the ***Residue Approach,*** in which designs arc represented as sets of constraints. Previous approaches, such as PROLOG [18] or the work of Manna and Waldinger [11], express designs as bindings on single terms. We give a complete and sound procedure for finding sets of propositions constituting a legal design. The size of the search space of the procedure and the advantages and disadvantages of the Residue Approach are analysed. In particular we show how Residue can avoid backtracking caused by making design decisions of overly coarse granularity. In contrast, it is awkward for the single term approaches to do the same. In addition we give a rule for constraint propagation in deductive synthesis, and show its use in pruning the design space. Finally, Residue is related to other work, in particular, to Default Logic [16] and to Assumption-Based Truth Maintenance [1].

1 Introduction

In this paper we describe the "Residue Approach" to deductive synthesis of solutions to design problems such as finding robot plans, designing digital circuits, picking circuit parameter values, and synthesizing computer programs. By a legal design, we mean an arrangement of the world which achieves the desired goals for known reasons. In the Residue Approach, we solve design problems by finding a ***residue*** for a given design goal. We define a residue as follows:

Definition 1.1 (Residue) ***Suppose that W is the initial world model and G the design goal, a set of facts R is*** a residue ***if*** it ***has the following three properties:***

RI. (Achieves Goal) $W \cup R \vdash G$

R2. (Consistent) $W \cup R$ ***is consistent.***

R3. (Realizable) ***The executor of the design can make R true.'***

R1 means that the goal G is true of the world obtained by adding facts ***R*** to the world ***W.*** However, we cannot add just any facts to obtain a legal design. As specified by R2 they must be consistent with the initial description ***W.*** For example, a serial computer program which specifies that two actions happen simultaneously is in contradiction with the world ***W.*** Secondly, as stated in It3 the facts ***R*** specifying the design must be such that we can implement them. A circuit specified as a VLSI layout may meet requirements Rl and R2, but is not the desired design if the executing agent has only discrete transistors at its disposal.

R2 also means that one can express ***a priori*** constraints on designs generated by Residue by asserting such constraints in ***W*** prior to design generation. This has proven difficult in other approaches.

As an example of a residue, consider a program for swapping the contents of two registers. Initially register R_a contains the value A and register

[1] In Definition 2.2 we make condition R3 more precise.

R_b contains the value B. The goal G is that at some future time t_f the contents have been switched, i.e.,

$$Contains(R_a, B, t_f) \wedge Contains(R_b, A, t_f) \wedge t_f \geq 1.$$

Let $M(x, y)$ be an action of the target machine for moving the contents of register x to register y. The set of propositions

$$execution(M(R_a, R_c), 1)$$
$$execution(M(R_b, R_a), 2)$$
$$execution(M(R_c, R_b), 3)$$

satifies requirements R1, R2, and R3 is thus a residue.

In Section 2 we present the Residue Procedure, a complete and sound procedure for finding residues in first order logic. Consistency checking, by means of which Residue gains many advantages, is potentially unsolvable. However, consistency checking need not always be computationally difficult, as will be discussed. We also add constraint propagation [20,21,22] to Residue. Though not necessary for completeness, constraint propagation can greatly speed up the search for a design. To date, however, constraint propagation has not been part of deductive design systems. Here, we give a rule for constraint propagation and show how it fits into deductive design synthesis.

Section 3 is an example of use of Residue With Constraint Propagation to find inputs values for a circuit with known outputs.

Section 4 discusses the motivation for the Residue Approach. Whereas Residue represents an evolving design as a set of propositions, the traditional approach to deductive synthesis has been to represent the evolving design as a single term of a language. Reprcscntion of an evolving design as a set of propositions gives Residue very fine granularity in making design decisions. As a result, Residue can avoid making unwanted design decisions as inscparnble parts of other design decisions. In the single term approach unnecessary backtracking is caused by the inability to specify design decisions at a sufficiently detailed lcvcl. Furthermore, adding new vocabulary to be able to specify more detail in decision making proves to be difficult.

Section 5 compares Residue to related research, in particular with Ray Reitcr's Default Logic [16] and Johan de Klecr's Assumption-Based Truth Maintenence [1].

Finally, Section 6 summarizes the results of our work.

2 The Residue Procedure

In this section we present the Residue procedure. To simplify the presentation, we have used standard clausal resolution [17]. However, there is no dependence on using clausal resolution, and non-clausal resolution [**14,1 1**] would have worked just as well. In fact, the principles of Residue work with any complete deduction system.

2.1 Preliminary Definitions

Recall from Section **1** that ***W*** is the set of propositions comprising our world model. G is the goal we aim to achieve. We will refer to **W** and $\overline{\mathbf{G}}$ as the set of clauses in the conjunctive normal forms of ***W*** and $\neg G$, respectively. WC refer to an ***indexed clause*** as a pair $(c \mid r \rangle$, where c is a clause and r is a set of propositions. We say that ***q is a literal of clause*** c if clause c is the disjunction of q with zero or more other literals. WC refer to two clauses A and ***B*** being resolved via unifier σ to yield C if A = $A' \vee a$, $B = B' \vee b$, σ is a unifier for the literals a and $\neg b$ and C = $A'a \vee B'\sigma$, where ***Pa*** is the result of applying the substitution σ to ***P.*** **U** is the current set of indexed clauses in the system, Λ is the literal *false*, and $\emptyset$ is the null set.

Let us now make more precise the Realizability Condition R3 of **Definition 1.1.** First WC define the notion of ***nssumabifity.***

Definition 2.1 (Assumable) ***A proposition p is*** assumable ij ***the executor can force p to be true.***

As illustrated in the example residue of Section **1,** example **assumables** in a planning domain might bc that 'a given action will take **place** at some time t_3 or that time t_3 is **before** some time t_f . **Residue** requires a **user-supplied procedure** for **deciding whether** a **given** proposition is in **the** class of ***assumables.*** **In** addition, combinations of **assumables** which **cannot** simultaneously bc made true must be **inconsistent** with ***W.*** For example, for a serial **machine,** ***W must*** contradict two simultaneous actions. Let us now **define *realizable*** as follows:

Definition *2.2* (Realizable) ***A set of propositions is realizable if it consists only of assumable propositions.***

2.2 The Procedure

We now present the Residue Procedure (See Figure 1 on page 5). We are building a dababase U of indexed clauses $\langle c \mid r \rangle$. The first member c of each indexed clause is simply a clause as in normal resolution. The second member r is the candidate residue, that is, the set of assumptions made to date in deriving c. We initialize U in Step 1 so that U is the set of indexed clauses comprising $W \cup \neg G$. The second member of each is $\emptyset$, indicating that no assumptions have been made in deriving any one of these clauses. Step 3a allows us to assume any primitively assumable proposition resolving with a clause to be refuted. Steps 3b, 3c, and 3d are a modified resolution rules in which we keep track (in the second member of each indexed clause) of the assumptions we have made to date in deriving the clause. The exit-condition is that (1) the resolution has succeeded, i.e., we have derived the null-clause A, and (2) we know of a binding σ for all of the variables in the candidate residue R such that the candidate design $R\sigma$ is consistent with the world model W.

Let us illustrate use of the Residue Algorithm with an example in propositional logic. Suppose we have a goal H, assumablcs B, $\neg D$, and *I,* and the following rules $A \wedge B \Rightarrow H$ and $\neg D \wedge E \Rightarrow$ A. We would then start out with indexed clauses

$$\langle \neg H \mid \emptyset \rangle \quad (1)$$
$$\langle \neg A \vee \neg B \vee H \mid \emptyset \rangle \quad (2)$$
$$\langle D \vee \neg B \vee A \mid \emptyset \rangle \quad (3)$$

By Clause-Clause Resolution of (1) and *(2)* we get the indexed clause $\langle \neg A \vee \neg B \mid \emptyset \rangle$. This clause may bc used with "Make Assumption" to yield the indexed clause $\langle \neg A \mid \{I?\} \rangle$, which is in turn resolved by Clause-Clause Resolution against (3) to yield $\langle D \vee \neg B \mid \{B\} \rangle$. Assumption-Clause Resolution is then used to derive $\langle D \mid \{B\} \rangle$. Finally "Make Assumption" is used again to derive $\langle A \mid \{B, \neg D\} \rangle$. Thus, if consistent, the set $\{\neg D, B\}$ is the desired residue.

Procedure 2.1 (Residue)

1. **(Initialize)** $\mathbf{U} \leftarrow \{(c \mid \emptyset\rangle \mid c \in (W \cup \overline{G})\}$.

2. **(If Finished, Consistency Check)** If for some R, $(A \mid R) \in \mathbf{U}$, and there exists a ground substitution σ such that $W \cup (Ro)$ **is satisfiable, then** R is the desired residue.

3. Execute one of 3a, 3b, 3c, or 3d. If none of these can be executed, then fail.

   (a) **(Make Assumption) If** $(C \mid D) \in \mathbf{U}$ **and proposition S is assumable** and C and S can be resolved via unifier σ **to yield C', then** $\mathbf{U} \leftarrow \mathbf{U} \cup \{(C' \mid (Du \{S\})\sigma)\}$.

   (b) **(Clause-Clause Resolution)** For $\langle C_i \mid D_i\rangle, \langle C_j \mid D_j\rangle \in$ U, **if** C_i and C_j can be resolved via the unifier σ to give the rcsolvent C_k, then $\mathbf{U} \leftarrow \mathbf{U} \cup \{\langle C_k \mid (D_i \cup D_j)\sigma\rangle\}$.

   (c) **(Assumption-Assumption Resolution)** For $(C \mid D) \in$ U, $d_1, d_2 \in D$, if d_1 and d_2 can be resolved via unifier σ to yield d_3, then $\mathbf{U} \leftarrow \mathbf{U} \cup \{\langle C\sigma \mid (D \cup \{d_3\})\sigma\rangle\}$.

   (d) **(Assumption-Clause Resolution)** For $(C \mid D) \in$ U, $d \in D$ and d and C can be resolved via, unifier σ to give the rcsolvent C', then $\mathbf{U} \leftarrow \mathbf{U} \cup \{\langle C' \mid Do)\}$.

4. **Go to 2.**

Figure 1: The Residue Procedure

In [5] we show completeness and soundness of the procedure. By completeness we mean that *given* W, G, and the class of assumables, *if* there exists a residue ***R,*** the procedure will eventually find it. Soundness says that any residue the procedure finds is indeed a residue, i.e., it meets conditions R1, R2, and R3 for being a residue.

2.3 Consistency Checking

Checking satisfiability of a set of first order *wffs* is undecidable, and more precisely, ***not semi-decidable.*** This means that no algorithm is guaranteed to give us an answer (in a finite amount of time) as to whether or not a set of *wffs* is consistent. This would seem to make Residue entirely useless, but in practice it does not. First of all, the world model *W* and set of assumables may be known to be such that it can bc inconsistent in only known ways. For example, in designing circuits WC need to avoid loops in the circuit, but we know that any combination of assumables not containing a loop is consistent. Checking for loops can easily be done in polynomial time. Similarly, we might know a set number of ways in which actions in a plan might interfere with each other. Secondly, we may bc able to live with approximations to consistency checking. If after a certain outlay of resources we have not proven a residue to be inconsistent, we assume it to be consistent.

2.4 Constraint Propagation in Residue

By constraint propagation WC mean deducing further constraints upon the solution from constraints already known. Constraint propagation has been been thoroughly discussed in the literature [20,21,22,1,2], and WC show here how it fits into deduction. Propagated constraints are ***necessary*** rather than *sufficient* conditions for finding a solution on a given path. Thus, the propagated constraints arc not needed for completeness, and can indeed be wasteful. A propagated constraint can only serve as an early warning of a conflict that would be eventually discovered anyway. On the other hand, if the constraint propagation is simple, it can be well worth the effort to avoid paths which must eventually fail.

Example: Suppose that we must refute $2a + b \neq 14$ (Eyuivalcntly, one can think of needing to prove $2a + b = 14$.) Since b must be ***even*** if this literal is to be refuted, then by noting this fact we can avoid generating search paths where b will be ***odd.*** Because ***even(b)*** is a ***necessary,*** but not *sufficient,* condition for finding a solution, we cannot derive ***even(b)*** by applying a rule

$$2a + b = c \text{ A } even(c) \Rightarrow even(b)$$

to $2a + b \neq 14$; its polarity is wrong.

Because a propagated constraint is ***a necessary,*** but not necessarily *sufficient* condition for proving a goal, standard deduction techniques such as resolution do not perform constraint propagation. Instead we need an additional rule to perform constraint propagation. The derived constraints are added (in negated form) to the set of literals which must be contradicted.

Constraint propagation may be intuitively expressed as follows:
For $\langle C \mid A \rangle \in \mathbf{U}$, $\overline{C}$ is the set of clauses in $\neg C$, $a_i \in \{\overline{C} \cup A\}$.

$$\text{if } a_1 \wedge \cdots \wedge a, \Rightarrow B, \text{then } B \leftarrow C \cup \{B\}. \tag{4}$$

However, we have no guarantee that B will be an assumable, so we are forced to add $\neg B$ to C. Furthermore, if B is a conjunction of n literals, then $\neg B$ will contain n clauses. Rather than give the entire rule here, we give the rule for B being a single literal. This will suffice for the Example of Section 3. See [5] for the complete rule.

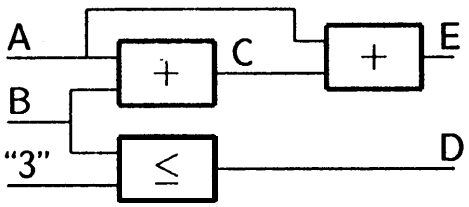
WC add the following rule to the Residue Procedure (Procedure 2.1):

3. **(e) (Constraint Propagation)** Let $\langle C_1 \mid R_1 \rangle$, $\langle C_2 \mid R_2 \rangle \in$ U and let $c_1, \cdots c_n$ be the the literals of C_1. If there is a clause $A = a_i \vee \cdots \vee a$, where $a_i \in \{R_1 \cup \{\neg c_i\}\}$, and A and C_2 resolve via unifier σ to yield a literal c', then $\mathbf{U} \leftarrow \mathbf{U} \cup \langle C_1\sigma \vee \neg c' \mid (R_1 \cup R_2)\sigma \rangle$.

Example: Suppose (C | ***R)*** ∈ U and 2a + $b \neq 14$ is a literal of C. Suppose also ($a_2 = 2a + b \neq$ **14 v** $\neg even(c) \vee$ ***even(b)*** $\mid \emptyset \rangle \in$ U. Let A = $a_1 \vee a_2$, where $a_1 = 2a + b =$ ***14,*** that is, the negation of the disjunct of C. Then, A and C_2 can be resolved to give the literal ***even(b).*** Thus, ***we*** can add $\langle C \vee \neg even(b) \mid R \rangle$. If ***even(b)*** is assumable, our constraint propagation gives us U ← U U (C | R U ***{even(b)}***) as desired.

3 Example of Residue

Consider the circuit below:



We will attempt to find values for A and B such that E = 14 AD = 1.[2] The initial database W contains the following rules:

$$A = x_1 \text{ A } B \text{ -- } x_2 \wedge x_1 + x_2 = x_3 \Rightarrow C = x_3 \quad \text{(C4)}$$
$$A = x_1 \text{ A } C = x_2 \text{ A } x_1 + x_2 = x_3 \Rightarrow E = x_3 \quad \text{(C5)}$$
$$B = x_1 \text{ A } x_3 \leq 3 \Rightarrow D = 1 \quad \text{(C6)}$$
$$2x + y = z \text{ A } \textit{even(z)} \Rightarrow \textit{even(y)} \quad \text{(C7)}$$
(C8)

The ***assumable*** clauses are any clauses matching the patterns below (the boldface names match any variable or constant).

$$x = y \quad \text{(A1)}$$
$$\mathbf{a} \leq \mathbf{b} \quad \text{(A2)}$$
$$\textit{even(x)} \quad \text{(A3)}$$

We start with an indexed clause representing the negation of the goal:

$$\langle E \neq 14 \vee D \neq 1 \mid \{\} \rangle \quad (E1)$$

Next we apply Clause-Clause Resolution (CCR) to (E1) and (C5) to obtain:

$$\langle A \neq a \text{ v } C \neq b \text{ v } a + b \neq 14 \text{ v } D \neq 1 \mid \{\} \rangle \quad (E2)$$

[2] Our saying that "E = 14" is means that the value of the line at point E is 14.

Again we apply CCR. to (C4) and (E2):

$$\langle A \neq d \vee B \neq e \vee d+e \neq b \vee A \neq a \vee a+b \neq 14 \vee D \neq 1 \mid \{\} \rangle \quad (E3)$$

Make Assumption (MA) $A = d$ on (E3) yields:

$$(B \# e \vee d+e \neq b \vee A \neq a \vee a+b \neq 14 \vee D \neq 1 \mid \{A = d\}) \quad (E4)$$

Assumption- Clause Resolution (ACR) of (E4) yields:

$$\langle B \neq e \vee d+e \neq b \vee d+b \neq 14 \vee D \neq 1 \mid \{A = d\}) \quad (E5)$$

Make Assumption of B = e on (E5) yields:

$$\langle d+e \neq b \vee d+b \neq 14 \vee D \neq 1 \mid \{A = d, B = e\}) \quad (E6)$$

CCR on the first two literals of (EG) yields?

$$\langle 2d+e \neq 14 \vee D \neq 1 \mid \{A = d, B = e\}) \quad (E7)$$

Via Constraint Propagation on (C7) and (E7) we get:

$$\langle 2d+e \neq 14 \vee D \neq 1 \mid \{A = d, B = e, even(e)\}) \quad (E8)$$

CCR on (E8) and (C6) yields:

$$\langle 2d+e \neq 14 \vee B \neq f \vee f \not\leq 3 \mid \{A = d, B = e, even(e)\} \rangle \quad (E9)$$

ACR on (E9) yields:

$$\langle 2d+e \neq 14 \vee e \not\leq 3 \mid \{A = d, B = e, even(e)\}) \quad (E10)$$

We assume e $\leq$ 3 in (E10) to get:

$$\langle 2d+e \neq 14 \mid \{A = d, B = e, even(e), e \leq 3\} \rangle \quad (E11)$$

Here WC make an arbitrary choice of $B = 2$ in (E11) to get:

$$\langle 2d+2 \neq 14 \mid \{A = d, B = 2, even(2), 2 \leq 3)) \quad (E12)$$

(E12) is simplified and $d = 6$ to yield:

$$(A \mid \{A = 6, B = 2, even(2), 2 \leq 3)) \quad (E13)$$

[3] We are assuming a manipulator for these algebraic equations.

4 Motivation for the Residue Approach

It is desirable that a design be output in such a manner as to be easily understood and executed. However, in arriving at the final program design, the design goes through numerous intermediate stages which may not resemble the final design. **It is emphatically *not* the case that the intermediate states of an evolving design are necessarily best expressed as they will be in the final design.** The space of intermediate designs not only contains the space of final designs, **but is considerably** larger.

The above observation has motivated development of the Residue Approach. In this section WC explain what sort of additional information is ideally present in intermediate design states and why, how the Residue Approach allows for ease of its representation, and why the traditional design synthesis approach has difficulty with such information. As we shall see, representation of the proper information at intermediate design stages can decrease backtracking required in design synthesis. If Residue is to be better than the traditional approach, the speed-up here must more than offset the additional overhead of consistency checking.

Size of the Search Generated by the Residue Procedure Residue, as well as the system to which we will compare it, fall loosely into the class *we* call ***goal-directed enumeration.*** Before looking at this class let us look at perhaps the simplest way to generate designs:

Blind enumeration is a trivial and complete procedure for design generation. One can simply enumerate candidate designs, starting with the shortest and proceeding to longer designs. Interleaved with their generation, the candidate designs are tested by a theorem prover to see if Rl (see **Section** 1) **holds. Unfortunately, however, this procedure is very slow. For a system** with k operators, one will have to generate $O(k^n)$ designs to find a design consisting of n operations.[4]

Rather than use blind enumeration, a goal-dircctccl enumeration is gen-

[4]Note that by *operator* we are referring to the operator itself as well as any arguments it may have. We are assuming that these are finite in cardinality. If countably infinite we can still write an enumerator for such designs, but the $O(k^n)$ no longer makes sense.

ernlly used. Instead of blindly enumerating designs, one starts with the goal and adds operations to the design only if the operation is ***applicable,*** i.e., only if application of the operator achieves some subgoal. Most of the planning systems from Green's [8] in the late 1960's through the 1970's used this approach. In Chapter 7 of [15] Nils Nilsson very clearly characterizcd this approach to plan generation. It is difficult to say precisely what the complexity is in such a search. If all operators are everywhere applicable, we still have a search space which is $O(k^n)$. On the other hand it is almost never the case that all operators are applicable at every point. If on the average there are k_0 operators applicable at every decision point, then the search space will be $O(k_0^n)$, where $k_0 \ll k$. Because Residue allows resolutions of two clauses from W it is not strictly a goal-directed enumeration. On the other hand, we can easily reduce Residue to a goal-directed enumeration; by using a ***set of support*** resolution search strategy. [15], for instance, Residue becomes goal-directed.

Contrast with the Single Term Approach By the ***Residue Approach*** we mean describing intermediate design states via sets of propositions. One may state both what is to be in the final design and what is ***not*** to be in the final design. In using propositions to constrain a design, one achieves great power of expression. Firstly, *all* designs consistent with the constraints are considered to be candidate designs without the set of candidate designs being enumerated." Secondly, the preexisting vocabulary of logic, namely, ordering relations, arithmetic relations, and set operations enable us to say virtually anything WC want to say about an evolving design.

In constrast to the Residue Approach, previous work in deductive design synthesis has rcprcscnted both intermediate and final stages of a design as a single term of the language. This approach, to which we refer as the **Single term approach,** is exemplified by the work of Green[8], PROLOG[18,23] or by the work of Zohar Manna and Richard Waldinger [11,12,13]. To illustrate the Single Term Approach, we look at Manna and Waldinger's program synthesis system. Here, we are given an input condition I' on lhc input CL, and a desired output condition ***R.*** The system attempts to prove

[5] This is the notion called *partial programs* in Genesereth [G]. His paper contains a discussion of the theory behind representing procedures as sets of constraints.

the goal

$$P(a) \Rightarrow R(a, z), \tag{1}$$

where z is the desired program. Note that the desired design has been reified as the term z. Rather than simply keeping track of the binding of z, Manna and Wnldingcr work with pairs (c, z), where c is a clause to be refuted and z is the evolving design[6]. Their inference rules specify how to combine both the clauses c and the design terms z of two such pairs. For example, the following rule of inference introduces an IF-THEN-ELSE into a design:

$$\frac{(F \vee P_1, z_1)\ ,\ (G \vee \neg P_2, z_2)}{(F\theta \vee \textit{GO}\ ,\ \textit{if}\ P_1\theta\ \textit{then}\ z_1\theta\ \textit{else}\ z_2\theta)}, \tag{2}$$

where θ is a most general unifier of P_1 and P_2. Note that the second member of the pair is a program segment in some programming language (which may contain free variables which will be bound later as the deduction continues). Another rule about mathematical induction on well-founded sets introduces recursive program calls into their output programs.

Difficulties in **the Single Term Approach** In searching for a **design** it is desirable to represent partial information about the design. For example, we might know that two actions A and B need to be executed without knowing their ordering. Ideally we should note the constraints that A and B will be in the plan and await more information before deciding upon an ordering. This is known as the ***least commitment approach*** [20,21]. In the Residue Approach, wc may state $execution(A, t_A)$ A $execution(B, t_B)$ without further information on t_A and t_B. In the single term approach, we usually have no way of stating these constraints without also specifying a temporal ordering. Instead of waiting for more information, the representation may force us to make an arbitrary decision, that is, a problem of **inseparable design decisions.**

In order to avoid the ad d i tion al backtracking caused by inseparable design decisions, wc might wish to reformulate our axioms with new terms expressing weaker design commitments. For example, wc might invent an

[6] Actually Manna and Waldinger use *non-clausal* resolution rules, allowing them to leave *wffs* in non-clausal form and to work with non-negates goal *wffs*.

ND-PROGN($x_1, \ldots, x_n$) function, the arguments of which are executed in any order. Doing so leads to other difficulties. First, we have an **explosion of ad hoc vocabulary.** Suppose we discover that B cannot be executed in a window between 3 and 6 seconds after A is executed. Do we really want to reformulate our axioms to include a NOT-IN-WINDOW($action_1, action_2, time_1, time_2$) function, when adding the constraint $t_A + 3 > t_b \vee t_B > t_A + 6$ would suffice? Secondly, **even if we do add** vocabulary to express partial information, there is no easy way to manipulate it. If we have noted ND-PROGN(A,B,C) and later discover $t_A < t_C$, further binding the design term or composing it with another term does not give us the desired result. Instead, we must eliminate the **ND-PROGN** and rebuild the design term more or less from scratch, that is, a **problem of discontinuity of representation.**

As we stated in Section 1, in the Residue Approach one can easily state **a priori constraints on designs** generated by including the constraints either in W or in G. In the Single Term Approach, there is no consistency check. Thus, in order to guarantee such constraints, we must start out with the output specification $R(a, z)$ already containing the necessary information. Again we have all the same problems as above of being able to express the desired constraints.

To avoid all these problems, one might choose to leave the design constraints as conjuncts of the goal. When enough information is present to represent a decision in the final design language, one **can bind the design term.** In that fashion sentences of logic rather than a **single term are used** to represent design constraints. The system uses these constraints (that is, goal conjuncts) by generating choices consistent with them. But, this is another way of describing the Residue Approach!

5 Related Research

5.1 Default Logic

Ray Reiter [16],[4] develops what he calls a "Logic for Default Reasoning". His goal is to develop a logic for drawing plausible conclusions which are unprovable, but consistent with the initial world model. For example, if

Fred is known to be a bird, Reiter's system will conclude that ***Fred*** can fly unless it can prove otherwise. In [5] we explain in detail the close connection between Default Logic and Residue. However, we present the results here without explanation. Our assumables correspond to Reitcr's **normal defaults.** The set of assumptions in a residue corresponds Reiter's **default support,** and the Residue Procedure is virtually identical to Reiter's **Top Down Default Proofs.** The correspondence of designs to Reiter's **extensions** is a bit trickier. If two different designs arc inconsistent, then they cannot both belong to the same extension. On the other hand, there will in general be numerous extensions of which a given design is a member. The reason for this is that a design only specifies part of the world. What happens outside of the design is irrelevant to the design, but changes thc extension. For example, a plan might specify all the actions in the world from time=0 until time=10, but says nothing about events after time=10. For every inconsistent course of events after time=10 there will be a separate extension.

5.2 Assumption-Based Truth Maintenance

The database **U** of Residue contains numerous indexed clauses, not all of which are consistent. Instead, each indexed clause (c, r) explicitly represents the set of assumptions it has made such that W U r U c I- G. In [1], Johan de Klecr has suggostcd a very similar idea. Rather than trying to keep a single consistent of assumptions and ramifications as in Doyle's Justification-13ased Truth Maintenance [3], dc Klecr suggests keeping numerous interpretations (set of assumptions) at the same time. One particular advantage is that contradictory interpretations can sometimes still use each other's work. For example, suppose that in trying to reach a goal in interpretation I_1 a proposition p can be derived from assumptions $R_1 \subseteq I_1$. Then another interpretation I_2 such that $R_1 \subseteq I_2$ could use p without further deduction on its part even if I_1 and I_2 arc in contradiction. Residue, as it currently stands, does not take advantage of such caching of assumptions. However, given that the assumptions are already present, it would bc an easy task to do so.

6 Summary

The approach of Residue makes a major change to the standard approach to deductive design synthesis. Rather than use a single term in which to build up a design, it builds a set of constraints which describe the design. As a result, Residue must check that the set of constraints describing the design is consistent with the original database, and that the set of facts must be at the proper level such that the design is realizable. A very beneficial side effect of the Residue Approach is the ability for the user to express a priori constraints upon the generator simply by asserting the constraints into the database.

We have stressed that the space of evolving designs is bigger than that of final designs. In particular, one wants to represent partial information about the design rather than individually consider all possible cases. Deduction systems in which the evolving design is represented as a single term have great difficulty in representing such partial information.

By representing evolving designs as sets of facts, we avoid several pitfalls of single term systems. First, because single term systems often cannot represent partial information they must do a case analysis for each possible case. By being able to easily represent partial information we avoid the need for such backtracking. Secondly, we avoid proliferation of special vocabulary in the attempt to make the single term handle certain special cases of partial information. In addition we need not build the logical machinery for handling the manipulation of the above ad hoc vocabulary.

On the other hand, a single term system has design consistency built into it. The rules are stated such that the single term always remains a legal design. The Residue Approach must pay the price of (I) stating axioms for what constitutes a legal design, (2) stating preexisting vocabulary for the set of facts which the system is willing to try to execute, (3) proving that any design found is consistent with (1) and (2). While the consistency check is in theory undecidable, in practice it need not be terribly difficult.

References

[1] de Kleer, Johan, Choices Without Backtracking, *Proc. of the AAAI-84 Nat'l Conf.* (August, 1984).

[2] Dietterich, Thomas G., Learning About Systems That Contain State Variables, *Proc. of the AAAI-84 Nat 'l Conf.* (August, 1984).

[3] Doyle, John,, A Truth Maintenance System, *Artificial Intelligence* **12** (1979) 231-272.

[4] Etherington, David and Reiter, Raymond, On Inheritance Hierarchies With Exceptions, *AAAI-83* (August, 1983) 104-108.

[5] Finger, J. J. and Michael R. Genesereth, RESIDUE: A Deductive Approach to Synthesis of Designs, Tech. Rept. IIPP-84-47, Stanford University (December, 1984), (in preparation).

[6] Genesereth, Michael, Partial Programs, Heuristic Programming Project Memo HPP-84-2, Stanford University (November, 1984).

[7] Green, Cordell C., Theorem Proving by Resolution as a basis for question-answering systems, in: Meltzer and Michie (Ed.), *Machine Intelligence 4* (Edinburgh University Press, Edinburgh, 1969).

[8] Green, Cordell C., Application of Theorem Proving to Problem Solving, *IJCAI-1* **(1969) 219-239.**

[9] Loveland, D. W., A linear format for resolution, *Proc. IRIA Symp. Automatic Demonstration* **(1968).**

[10] Luckham, D., Refinements in resolution theory, *Proc. IRIA Symp. Automatic Demonstration* (1968).

[11] Manna, Zohar and Waldinger, Richard, A Deductive Approach to Program Synthesis, *ACM Transactions on Programming Languages and Systems* **2 (1) (1980) 90-121.**

[12] Manna, Zohar and Waldinger, Richard, Problematic Features of Programming Languages: A Situational-Calculus Approach, *Acta Informatica* **1.6 (1981) 371-426.**

[13] Manna, Zohar , and Waldinger, Richard, Special Relations in the Program-Synthetic Deduction, Tech. Rept. STAN-CS-82-902, Stanford University (March, 1982).

[14] Murray, N.V., Completely Non-Clausal Theorem Proving, ***AI* 18 (1)** (January, 1982) 67-85.

[15] Nilsson, ***N.*** J., ***Principles of Artificial Intelligence*** (Tioga Publishing Co., Palo Alto, 1980).

[16] Reiter, R., A Logic for Default Reasoning, ***Artificial Intelligence* 13 (1980) 81-132.**

[17] Robinson, J. A., A tnachinc-oriented logic based on the resolution principle, ***Journal of the ACM 12* (1) (1965) 23-41.**

[18] Roussel, P., Prolog: Manual de reference et d'utilisation (1975), Groupe d'Intelligence Artificielle, Marseille-Luminy; September.

[19] Saccrdoti, Earl D., A Structure for Plans and Behavior, Tech. Rept. Technical Note 109, SRI (August, 1975).

[20] Stallman, Richard M. and Sussman, Gerald J., Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, ***Artificial Intelligence* 9** (2) (October, 1977) 135-196, Reprint in **"AI-MIT",** vol. **1, pp.31-91. Also** MIT AI Memo 380,'76.

[21] Stefik, M., PIanning with Constraints (MOLGEN: Part 1), ***Artificial Intelligence* 16 (2) (1981)** 111-140.

[22] Sussman, Gerald Jay and Steele, Guy Lewis Jr., Constraints - A Language for Expressing Almost-Hierarchical (1) **(1980) 1-39.**

[23] Warren, D.H.D., and L.M. Pereira, **PROLOG - The Language and** its implementation compared with LISP, ***SIGPLAN Notices, 12(8); and SIGART Newsletter, no. 64, pp. 109-115*** (1977), Also Proc. of the Symp. on AI and Programming Languages **(ACM).**