Report 83-46
Stanford -- KSL

RESIDUE: A Deductive Approach to Design.
J.J. Finger, Michael R. Genesereth,
Dec 1983

Scientific DataLink

card 1 of 1

Stanford Heuristic Programming Project
Report No. HPP 83-46

December 1983

RESIDUE: A Deductive Approach to Design

by

J.J. Finger, and Michael R. Genesereth

Computer Science Department
Stanford University
Stanford, California 94304

Table of Contents

List of Figures

Residue - A Deductive Approach to Design

J. J. Finger and Michael R. Genesereth

Computer Science Department
Stanford University,
Stanford, California 94305

1. Introduction

Automatic theorem proving methods, such as resolution or backwards-chaining, are useful for verifying the correctness of a pre-existing design, but how might we use theorem proving methods to both *generate* and *verify* a design? This paper describes the RESIDUE system, an attempt to answer this question.

1.1. Example: Verifying a Design

To verify the design of a half adder circuit (see figure 1-1), we *assert* the description of the circuit as being given, and then have as a goal *to prove the theorem* $\text{SUM} = I_1 \otimes I_2 \wedge \text{CARRY} = I_1 \wedge I_2$.

1.2. Example: Generating a Design

Given a description of legitimate components of a circuit, say Nor-gates and Wire s, and the goal (as in Example 1.1) $\text{SUM} = I_1 \otimes I_2 \wedge \text{CARRY} = I_1 \wedge I_2$, our goal is to *describe a circuit which would cause this goal to be true.*

Figure 1-1: A Half Adder Circuit

2. The Basic Intuition: Assuming a Subgoal is Possible

In our half adder example, we start only with what we know about circuits. Clearly, we cannot yet prove that the goal $\text{SUM} = I_1 \otimes I_2 \wedge \text{CARRY} = I_1 \wedge I_2$ is true about our database. But suppose we say to ourselves:

> *What else would have to be true in order to prove the goal? We will make a list of such items. If all the items on the list can simultaneously be accomplished, then we have a valid design.*

We are willing to *assume* that we can make certain classes of facts true, although we know that we might not be able to make arbitrary combinations of such facts true. This class of facts will be referred to as **assumables**, and these are exactly the items we are willing to add to the list of facts needed to prove our goal. For example, we can always put Nor-gates or Wire s into the circuit, and we are always given inputs I_1, I_2, $\sim I_1$, and $\sim I_2$ to use. On the other hand, arbitrary combinations of Nor-gates and Wire s might cause a loop in the circuit, an *infeasible* design, as no combinational circuit can contain a loop. We will guard ourselves against such a combination of assumptions as will be explained further in section 3.3 on feasibility checking.

3. Generation of a Digital Design

We are now ready to walk through the generation of the design of the half adder circuit as shown in Figure 1-1. First, in Figure 3-1 we see the state of the system prior to our search though the design space. In Figure 3-2, we see the results of successive steps along a path through the design space leading to the design of the half adder. Finally, in Figure 3-3 we see the list of assumptions we made during the generation of the design (in Figure 3-2). This ASSUMPTION LIST represents our design for half adder circuit and is identical to the circuit in Figure 1-1.

GOAL:	**EXPLANATION:**
Out (CARRY,$I_1 I_2$) ∧	[G_0] Original Goal
Out (SUM,$I_1 \oplus I_2$)	
ASSUMABLES:	**EXPLANATION:**
NorGate (?x)	[A_0] Always can add Nor-gates
Wire (?w,?x,?y,?n)	[A_1] Always can add Wire
In (?d,?n,I_1)	[A_2] I_1 is always available as input to any device ?d
In (?d,?n,I_2)	[A_3] I_2 is always available as input to any device ?d
In (?d,?n,~I_1)	[A_4] ~I_1 is always available as input to any device ?d
In (?d,?n,~I_2)	[A_5] ~I_2 is always available as input to any device ?d
WORLD MODEL:	**EXPLANATION:**
$\forall x,y_1,y_2$ In (x,1,~y_1) ∧ In (x,2,~y_2) ∧ NorGate (x) ⇒ Out (x,$y_1 y_2$)	[R_1] Definition of Nor-gate
∀x,y,n Wire (x,y,n) ∧Out (x,z) ⇒ In (y,n,z)	[R_2] Definition of Wire
∀w,x,y,z,n Path (x,y) ∧ Wire (w,y,z,n) ⇒Path(y,z)	[R_3] Definition of Path
∀w,x,y,n Wire (w,x,y,n) ⇒ Path (y,z)	[R_4] Definition of Path
DESIGN CLASS:[1]	**EXPLANATION:**
∀x ~Path (x,x)	No loops allowed in combinational circuit
Meanings of Relations:	
Out (c,v)	The output of component *c* has value *v*
In (c,n,v)	$Input_n$ of component *c* has value *v*
NorGate (d)	*d* is a Nor-gate
Wire (w,o,i,n)	Wire *w* connects the output of *o* to $input_n$ of *i*
Path(x,y)	There exists a path from component *x* to component *y*.

Figure 3-1: Initial State for Half Adder Generation

[1] The "Design Class" is a rather artificial construction made here for the sake of simplicity of notation. The point is that the system must be told it is building a combinational circuit (so that it knows there are no loops or shorted outputs, etc). It might be better to add a term to the NORGATE and WIRE relations specifying the name of their circuit and then so add to the goal COMBINATIONAL(A) where A is the name of the circuit. COMBINATIONAL is then treated as an assumable to be checked as we go.

3.1. Initial State of the Design

In order to generate a design, it is necessary that our engine (as well as a human designer) understand what the goals are, and with what it may work. We see the following breakdown of design components:

- **The Goal Specification:** a description of what the design must achieve. In our case, this is simply SUM = $I_1 \otimes I_2$ ∧ CARRY = $I_1 \wedge I_2$. These specs might also include the goals a robot plan must accomplish, or constraints on a house design, etc.

- **A Class of Assumables:** a description of the set of assumptions one is willing to make in generating the design. In our case, we are willing to assume Nor-gates and connections (wires) between them. We might also include various constraints as assumable, e.g., temporal constraints on robot actions, or constraints on the value of some resistor, etc. The important point is that these assumables are in some sense *primitive*. We do not want to design a Nor-gate; we simply want to put it in the circuit. Note that the ASSUMABLEs in Figure 3-1 are *patterns* of atomic formulae rather than a single formula. Any formula matching one of these patterns is allowed to be assumed.

- **A World Model:** the knowledge base telling the design engine what is known to be true about the world. Among other things, it must capture the definition of the design operators, that is, to capture knowledge about the relations of ASSUMABLES to the rest of the world, for example, [R_1] gives the relation of a Nor-gate to its inputs and output. In a robot planning example, we would have rules about the robot *actions* (which are ASSUMABLEs). We would also have any information known about the initial state of the robot world.

- **Design Class:** knowledge of what constitutes a *feasible* design (see section 3.3). In our case, we know that we are designing combinational circuit and thus loops are forbidden. Any other circuit we put together out of the components given will be acceptable.[2]

3.2. The Design Engine

In Figure 3-2 we take a goal and reduce it to a set of ASSUMABLEs, this being the function of the *design engine*. The engine itself need not be specified; it need only be capable of deductive inference of some sort with the added proviso that it can add *assumable* goals to an ASSUMPTION LIST. For example, both of the authors have implemented numerous residue systems, some based on resolution and others on backwards chaining. It is important, however, to note that the engine is conducting a search through the space of possible designs. Thus, at each point along the way it is making a decision - which goal conjunct to expand, which rule to use for expansion of the conjunct, what bindings to make, etc. For a useful system, this search must be guided heuristically, an issue which we not addressed in this paper.

The search path shown in Figure 3-2 was the successful path along which a half adder circuit was generated. The steps taken fall into only a few categories:

- **Rule Expansions:** (steps 1,3,4,6,9, and 11) expansion of a conjunct of the goal list according to some rule in the database.

- **Making Assumption:** (steps 2,5,7,10, and 12) adding an ASSUMABLE goal conjunct to the ASSUMPTION LIST.

---

[2] For example, our engine would not produce designs which would short together outputs of two Nor-gates. This we know, and thus needn't guard against it. Otherwise, we would need explicit axioms about shorted outputs just as we have axioms about loops.

STEP:	GOALS:	ASSUMPTION LIST:	COMMENTARY:
0	Out (C,I_1I_2) Out (S,$I_1 \otimes I_2$)		Original Goal
1	NorGate (C) In (C,1,~I_1) In (C,2,~I_2) Out (S,$I_1 \otimes I_2$)		[R_1]
2	Out (S,Out (S,$I_1 \otimes I_2$)	NorGate (C) In (C,1,~I_1) In (C,2,~I_2)	[A_0] Assume NorGate (C) [A_4] Assume ~I_1 available [A_5] Assume ~I_2 available
3	Out (S,~(I_1I_2) $\wedge$ ~(~I_1~I_2))		Definition of $\otimes$
4	NorGate (S) In (S,1,I_1I_2) In (S,2,~I_1~I_2)		[R_1]
5	In (S,1,I_1I_2) In (S,2,~I_1~I_2)	NorGate (S)	[A_0] Assume NorGate (S)
6	Wire (W_1,C,S,1) Out (C,I_1I_2) In (S,2,~I_1~I_2)		[R_2]
7	Out (C,I_1I_2) In (S,2,~I_1~I_2)	Wire (W_1,S,C,1)	[A_1] Assume Wire (W_1) from S to In (C,1)
8	In (S,2,~I_1~I_2)		Out (C,I_1I_2) already achieved
9	Wire (W_2,N_1,S,2) Out (N_1,~I_1~I_2)		[R_2]
10	Out (N_1,~I_1~I_2)	Wire (W_2,N_1,S,2)	[A_1] Assume Wire (W_2)
11	NorGate (N_1) In (N_1,1,I_1) In (N_1,2,I_2)		[R_1]
12	Λ	NorGate (N_1) In (N_1,1,I_1) In (N_1,2,I_2)	[A_0] Assume NorGate (N_1) [A_2] Assume I_1 available [A_2] Assume I_2 available

Figure 3-2: Generation of Half Adder Circuit

- **Known Subgoals:** (step 8) elimination of a subgoal which is already in the database or a previous subgoal on this search path.

Assumption List Made in Designing Half Adder

NorGate (C)
  In (C,1,~I_1)
  In (C,2,~I_2)
NorGate (S)
  Wire (W_1,C,S,1)
  Wire (W_2,N_1,S,2)
NorGate (N_1)
  In (N_1,1,I_1)
  In (N_1,2,I_2)

Figure 3-3: Generated Design for Half Adder Circuit

3.3. Checking for Feasibility

Our design can to be implemented from the ASSUMPTION LIST assuming that all of the assumptions can be simultaneously true. Let us define this condition as *feasibility*, that is, that the entire ASSUMPTION LIST can be true at the same time as the rest of the database.[3,4] But how can we determine whether an ASSUMPTION LIST is feasible?

> In our HALF ADDER example, the only infeasible designs which can be derived are circuits with loops. Thus, we must check for instances of loops. In the example there are no generated loops in the circuit, and so we know are circuit is good. We might consider, however, a design for register swapping $\mathcal{A} \leftrightarrow \mathcal{B}$ in which the assumable $\mathcal{A} \rightarrow \mathcal{B}$ must precede $\mathcal{B} \rightarrow \mathcal{A}$, and *vice versa*. In this case, we have failed to realized that we must assume existence of a third register, and as a result, we have postulated an infeasible design.

Let us now make the notion of feasibility sharper. First we will make explicit an important assumption: *if a design is infeasible, this fact is provable from our database.* More precisely,

> INFEASIBLE(D) $\Leftrightarrow$ UNSATISFIABLE(DB$\wedge$D), where (1)
> **DB** represents the conjunction of facts in the database, and
> **D** represents the conjunction of facts in the ASSUMPTION LIST.

In other words, a design is infeasible if it has no model, that is, all interpretations are false. For a more useful formulation, we note[5] that:

> UNSATISFIABLE(DB$\wedge$D) $\Leftrightarrow$ VALID(~(DB$\wedge$D)) $\Leftrightarrow$ VALID(DB $\Rightarrow$ ~D) (2)

The complexity of proving feasibility depends strongly on the class of formulas in DB$\wedge$D. For propositional logic, the VALIDITY problem is solvable, so we can find an algorithm for deciding whether a design is

---

[3] See [Reiter 80] for a discussion of the similar problem of determining that an extension of a theory is consistent with the theory.

[4] Manna and Waldinger ([Manna 80] or [Manna 82]) are synthesizing *applicative* programs, that is, side effect-free programs. For them, *any* program produced is feasible since the only ASSUMABLES are pure functions, which can be composed in any order.

[5] See, for example, [Manna 74], p. 91

feasible or not. For unrestricted first order logic, the VALIDITY problem is only partially solvable. Thus if a design is infeasible we will eventually prove it to be so, but a feasible design might cause our feasibility checker to execute forever without returning an answer.[6] Note that partial solvability does not mean that we can never prove feasibility. If a proof for infeasibility terminates in failure, then we have guaranteed a feasible plan (modulo, of course, our earlier assumption).

Because a proof of feasibility might take an arbitrarily long time, we might specify that we will *assume* that our design is feasible if the system cannot prove that it is infeasible in a reasonable amount of time. We might think of this as corresponding to a human engineer mulling over a building plan for only so long before deciding to go ahead and build it. He does this in spite of the fact that he knows that there still might be uncovered dangers.

3.4. Non-Monotonicity of Residue

In RESIDUE we start with a given database, and we add to it facts (assumables) which were not provable from the initial database. This feature of RESIDUE puts it squarely in the camp of non-monotonic logic (see [McDermott 80], for example), in that facts which were once provable are no longer provable, and *vice versa.*

> Before starting to design our adder, the requirement $\forall x$ ~Path (x,x) was vacuously true. However, as we added components, there might have been formed a loop. Hence, it does not suffice to check an assumable one time for consistency and check it off as being OK.

Suppose we have a design D known to be feasible, and we want to add to D a new assumable A. An easy manipulation of Equation 2 gives us an easy way of stating the goal of feasibility checking for the new design:

$$\begin{aligned} \text{INFEASIBLE}(D \wedge A) &\Leftrightarrow \text{UNSATISFIABLE}(DB \wedge (D \wedge A)) \\ &\Leftrightarrow \text{UNSATISFIABLE}((DB \wedge D) \wedge A) \\ &\Leftrightarrow \text{VALID}((DB \wedge D) \Rightarrow \sim A) \end{aligned} \tag{3}$$

We see that iff we can prove ~A from the old ASSUMPTION LIST and database, then we have an infeasible design. So, if before adding an assumable to the ASSUMPTION LIST, if we first fail to prove its negation we maintain a feasible plan.

3.5. Truth Maintenance in Feasibility Checking

As we have stated, the addition of new assumables can invalidate previous feasibility proofs, but this does not mean we must start from scratch each time. Two notions motivated from truth maintenance ([Doyle 79], [Doyle 83]) present themselves to help us:

- **[Caching the Proof]:** Suppose we keep a trace of the previous feasibility proof (or failed infeasibility proof, depending on how one views it), and we derive from the trace a set of facts one of which would have to be true in order to invalidate the proof (or further the infeasibility proof). It might be far simpler to see that a new assumption does not invalidate any of the these facts than starting the proof from scratch.

- **[Caching a Counterexample]:** If a design is infeasible, then there is *no* model for it, that is, every interpretation is false. So, if we have an interpretation which was true for a previous version of the design and it stays true for an added assumption, we still have a counterexample to INFEASIBLE(D).

---

[6] For examples of other subclasses of first order logic for which the VALIDITY problem is solvable, see [Manna 74], p. 107.

For example, suppose we design a robot plan in which the ordering of actions is only partially specified. If we show that the design is feasible for some assumed total ordering, then if we add some new time constraint consistent with the previously assumed total ordering, our counterexample stays valid.

3.6. A Trade-Off: Pruning the Search via Feasibility Checks

The theorem proving engine will, in general, be searching down numerous paths trying to reduce the original goal to a set of ASSUMABLEs. If at any time an ASSUMPTION LIST for a given path can be shown to be infeasible that path can be pruned. For example, an ASSUMPTION LIST describing a loop in a combinational circuit should eliminate the circuit from consideration despite having left parts of the circuit undesigned. On one hand, we do not want to perform many largely redundant feasibility checks. On the other hand, an early feasibility check may immediately eliminate a bad design from consideration. So, we have a trade-off between blind generation and too much testing. Clearly the right course of action is situation dependent.

Noteworthy, however, is that we need only *complete* a feasibility check after the design is fully specified. If there are some inexpensive checks which will catch obvious infeasibilities, there is no reason not to use these checks to take a quick look at feasibility all along the way.

4. Current Implementation

Currently, there exist both resolution based and backward-chaining implementations of RESIDUE in MRS [Genesereth 82a]. The authors and our coworkers have implemented numerous RESIDUE systems ranging in application from circuit fault diagnosis in DART [Genesereth 82b], to robot planning [Finger 83], to WUMPUS players and academic course schedulers among others in [Genesereth 83].

5. Summary

6. References

[Doyle 79] Doyle, John.
A Truth Maintenance System.
Artificial Intelligence 12:231-272, 1979.

[Doyle 83] Doyle, Jon.
The Ins and Outs of Reason Maintenance.
In *IJCAI8*, pages 349-351. Karlsruhe, West Germany, August, 1983.

[Finger 83] Finger, J. J. and Genesereth, Michael R.
Sensory Planning - Planning to Gather Information.
Technical Report HPP-83-47 (working paper), Stanford University, December, 1983.

[Genesereth 82a] Genesereth, Michael R.
An Introduction to MRS for AI Experts.
Technical Report HPP-82-27, Stanford University, November, 1982.

[Genesereth 82b] Genesereth, Michael.
Diagnosis Using Hierarchical Design Models.
In *AAAI-82*, pages 278-283. Pittsburgh, Pennsylvania, August, 1982.

[Genesereth 83] Genesereth, Michael R., ed.
The MRS Casebook.
Technical Report HPP-83-26, Stanford University, May, 1983.

[Manna 74] Manna, Zohar.
Mathematical Theory of Computation.
McGraw-Hill Inc., San Francisco, 1974.

[Manna 80] Manna, Zohar and Waldinger, Richard.
A Deductive Approach to Program Synthesis.
ACM Transactions on Programming Languages and Systems 2(1):90-121, 1980.

[Manna 82] Manna, Zohar, and Waldinger, Richard.
Special Relations in the Program-Synthetic Deduction.
Technical Report STAN-CS-82-902, Stanford University, March, 1982.

[McDermott 80] McDermott, D. and Doyle, J.
Non-monotonic Logic - I.
Artificial Intelligence 13:41-72, 1980.

[Reiter 80] Reiter, R.
A Logic for Default Reasoning.
Artificial Intelligence 13:81-132, 1980.