DERIVING PROPERTIES OF SYSTEMS FROM PROPERTIES OF PARTS
AND LISTS OF CONNECTIONS

John R. Gabriel and Richard O. Chapman

Mathematics and Computer Science Division

December 1986

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

† R. O. Chapman's permanent address: Wadham College, Oxford OX1 3PN, U.K.
Contents

Abstract ........................................................................................................................................1

1. Introduction ................................................................................................................................1

2. Notes on the Program and Description Language ......................................................................2
   2.1 Elementary Objects ...................................................................................................................2
   2.2 Syntactic Axioms .....................................................................................................................3
   2.3 Procedural Semantic Axioms ...................................................................................................4

3. Outline of the Computation .......................................................................................................5
   3.1 The "evalSig" Procedure .........................................................................................................5
   3.2 The PROLOG Clause for "findcase" ......................................................................................7
   3.3 The PROLOG Procedure for "docase" ...................................................................................9
   3.4 The Utilities ............................................................................................................................14

References .....................................................................................................................................21
Deriving Properties of Systems from Properties of Parts and Lists of Connections*

J. R. Gabriel and R. O. Chapman

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844

ABSTRACT

This paper presents an algorithm in PROLOG for compiling recursively computable descriptions of system behavior from computable descriptions of behavior for parts and lists of interconnections. We give a set of conditions that must be satisfied by various data structures in the computation. It seems possible to provide an informal verification (by hand) that these conditions are true also of the output.

1. Introduction

This paper continues the work of Gordon [1981] and Barrow [1983, 1984] on proof of properties for systems of interconnected parts.

Although Gordon and Barrow described their results in terms of digital logic systems, we feel their work can be extended to address a larger domain. The grounds for this belief will be discussed in detail elsewhere; preliminary material has been already published [Chapman and Gabriel 1986]. In brief, the reasons for this view are twofold:

1. Physical systems whose state evolves with time are characterized by a "classical contact transformation" deriving the state at time $t+dt$ from that at time $t$. If this model is made discrete (giving rise to a discrete semigroup having a representation by the motion of the system), then the system state at any finite time can be determined from its initial state by a recursively computable function.

2. Since the system description language given by Barrow has the descriptive power sufficient to characterize a finite state digital logic system, and a finite state machine can mimic any recursively computable function, the language has the potential to describe a representation of any physical system characterized by a discrete semigroup. It seems likely, but not proved, that the language can also describe systems characterized by continuous semigroups and infinitesimal classical contact transformations, since such systems usually have descriptions of cause that can be specified by block diagrams (although such diagrams are more useful in some cases than others). In any event the continuous case is less interesting since it must usually be cast in discrete form for numeric computation.

For our purposes it is desirable that the verification algorithm be verifiable. Such verification requires a more detailed description of the algorithm than those given previously.

* This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.
and if possible at least an informal proof of correctness. This paper gives an executable
description and supplies an informal proof we hope to use in future as a foundation for a
verification of the algorithm by means of itself.

The publications of Barrow [1983, 1984] are more concerned with describing the impres-
sive results obtained by use of Gordon's methods rather than the details of the PROLOG pro-
gram used. Here we describe in detail a compiler, written in PROLOG, that is able to translate
a system description given in Barrow's terms. In addition, we give a set of conditions that the
"signals" computed by the system must satisfy for the computation to succeed. These condi-
tions are satisfied by the input. It seems likely that the following steps would lead to a proof
(perhaps by a program) that they are also satisfied by the output, that is, that they are invari-
ants of a recursion on the procedure evalSig(...) which is the "computational engine" of this
system:

(1) Compile PROLOG to a dataflow description in Barrow's terms.

(2) Use evalSig to obtain an input/output relation for evalSig. These are essentially Church's
lambda functions.

(3) Use a theorem prover such as ITP [Lusk and Overbeek 1984] or a rewriting system such
as SARA [Gabriel 1986] to prove that the axioms are invariant under transformation by the
lambda function of Step (2).

It should be noted here that an "understandable" (i.e., more or less manually verifiable)
algorithm was difficult to obtain. Explicit consideration of the semantics captured by a number
of "axioms" was needed to develop the algorithm. For the sake of completeness, we have
therefore included these axioms in our report, along with the code in PROLOG.

It is also worth noting that PROLOG proved a remarkably efficient programming
language for this task. In particular, the last several versions of the algorithm were developed
quite fast—three or four days to think, one to code, and one or two to debug. Since the form
of each new version bore only superficial resemblance to its predecessor, this is strong testi-
mory to the power of PROLOG.

2. Notes on the Program and Description Language

Our discussion of semantics focuses on a compiler from dataflow descriptions to recur-
sively computable functions describing behavior.

2.1. Elementary Objects

The elementary objects are PROLOG unit clauses describing the dataflow. Their mean-
ing is further described in later sections. This section defines otherwise undefined objects in
the universe of discourse and gives (informal) partially complete descriptions of their mean-
ings.

module(Sys) - succeeds when Sys is the "name" of a system.

port(Sys, Signal, Direction, Datatype) - defines input and output ports of a system. The port-
names are instances of Signal.

outputEqn(Sys, Signal := Function) - describes an input/output mapping implemented by a
system and not depending on past history. Signal is an output port, and Function has as argu-
ments input ports, constants, state variables (described later), or instances of the generic object
Function. Thus, at some sufficiently deep level of nesting, arguments to Function are inputs, state variables, or constants.

\texttt{stateDef(Sys, StateVar, Datatype)} - defines a state variable of the given datatype in the module Sys.

\texttt{stateEqn(Sys, StateVar := Func)} - defines StateVar and Func.

\texttt{part(Sys, Instance\_of\_Sysl, Sysi)} - defines a submodule (subsystem) of Sys, of generic type Sysi, having the name Instance\_of\_Sysl. Input/output signals of Sysi have the syntactic form \texttt{<signal\_name>(Instance\_of\_Sysl)}. The Instance\_of\_Sysl is a nested data structure containing an instance of Sys at the next innermost level. If Sys is an outermost system, the instance of Sys is syntactically \texttt{Sys(Module\_Name)}, where Module\_Name may be any legitimate PROLOG term.

\texttt{stateMap(Sys, SysState, Sys1Func)} - defines the mapping between a state variable in Sys, and a state variable in one of its parts; that is to say, Sys1Func has the same recursive definition as Func in \texttt{stateEqn}.

\texttt{connected(Sys, Source, Dest)} - describes connections between ports, either of Sys and its parts, or between parts of Sys.

2.2. Syntactic Axioms

Syntactic axioms describe legitimate terms or structures in the PROLOG dataflow compiler. They repeat and amplify statements from the preceding section.

\textbf{Axiom 1}: If module(Term) succeeds, Term is a system.

\textbf{Axiom 2}: If Term is a system, Term is also an atom in the PROLOG sense. This needs revision for implementations supporting arrays of modules.

\textbf{Axiom 3}: If part(Sys, Term, Sys1) succeeds, Sys and Sys1 are systems, and Term is an instance of Sys1 and a part of an instance of Sys.

\textbf{Axiom 4}: If Term is a part of an instance of Sys, then Term is a structure of arity 1, whose argument is the instance of Sys.

\textbf{Axiom 5}: If the principal functor of Term is a system, Term has arity 1. and part(_,Term,_) fails, then Term is an instance of an outer system. The argument of Term is the name of the instance of the outer system.

\textbf{Axiom 6}: If port(Sys, Term, Dir, Type) succeeds, Term is a signal in an instance of Sys.

\textbf{Axiom 7}: If Term is a signal in an instance of Sys, then Term has arity 1.

\textbf{Axiom 8}: If Term is a signal in an instance of Sys, and the argument of Term is an instance of Sys, then Term is an external signal of the instance of Sys.

\textbf{Axiom 9}: If Term is a signal in an instance of Sys, and the argument of Term is an instance of a part of Sys, then Term is an internal signal of the instance of Sys.
Axiom 10: If port(Sys, Term, input, Type) succeeds, then Term is an input signal.

Axiom 11: If port(sys, Term, output, Type) succeeds, then Term is an output signal.

Axiom 12: External inputs and outputs are external signals that are inputs or outputs.

Axiom 13: Internal inputs and outputs are internal signals that are inputs or outputs.

Axiom 14: If connected(Sys, Source, Dest) succeeds, then one of the following must be true:

(i) Source is an external input, and Dest is an internal input bound to a part of Sys.

(ii) Dest is an external output, and Source is an internal output of a part of Sys.

(iii) Source is an internal output bound to a part of Sys, and Dest is an internal input of a different part of Sys.

Axiom 15: If connected(Sys, Source, Dest) succeeds, then (port(_,Source,_,Type),port(_,Dest,_,Type)) succeeds (i.e., source and Dest are signals of the same type).

Axiom 16: If stateDef(Sys, Term, Type) succeeds, then Term is said to be a state variable of Sys.

Axiom 17: State variables are functors of arity 1; their arguments are instances of the systems for which they are state variables.

Axiom 18: If Term is a state variable of Sys, then stateEqn(Sys, Term := StateFunc) succeeds.

Axiom 19: If part(Sys, _, _) fails, then Sys is a primitive system.

Axiom 20: If Sys is a primitive system, then for each output signal of Sys, outputEqn(Sys, Signal := Func) succeeds.

Axiom 21: If outputEqn(Sys, Signal := Func) succeeds, then Func is a data structure having input signals of Sys as substructures at some depth determined by the form of Func.

Axiom 22: If Sys is not a primitive system, Sys is a compound system.

Axiom 23: Every compound system has at least one part; that is, part(Sys,Instance_of_SubSys,SubSys) succeeds at least once.

Axiom 24: Parts of compound systems are either primitive or compound.

2.3. Procedural Semantic Axioms

Procedural semantic axioms define how outputEqn and stateEqn predicates are to be constructed for compound systems.

This statement of procedural semantics omits the use of arrays of systems, which are not at present part of the domain of this program, and it omits discussion of recognizing cyclic data flows not broken by state variables. An implementation addressing these issues is likely
to have additional arguments of the predicates described below, and additional predicates.

3. Outline of the Computation

The complete PROLOG program is divided into five parts. The first four are as follows:

1. **The "evalSig" procedure**, which returns the symbolic value of a signal when invoked.
2. **The "findcase" procedure**, which performs a case analysis on the symbol for the signal requested of evalSig to determine which of the possible computations should be performed,
3. **The "docase" procedure**, which performs the evaluation, making recursive calls to evalSig and asserting unit clauses of facts derived about the system.
4. A collection of utilities used by the other three parts, including the "main" procedure test(Sys, Id), and the procedure cleanup which retracts facts developed in a computation, and, if the system definition contained a clause definition_file(FileName), reconsults the file FileName in case cleanup accidentally deleted any definitions that were properly part of the problem definition.
5. The pp(Struct) output (pretty-print) procedure, which was developed by Barrow and is in the PROLOG library at SRI. Since it is copyrighted, we do not reproduce it here; for quick experiments, however, pp may be defined as pp(S):- init(S), nl.

The program is run by starting PROLOG and consulting several files as follows:

- The file "evaluate" containing the algorithm.
- The file "utilities" containing general-purpose utilities and Barrow's pp(S) printing procedures.
- The system definition.
- Any additional predicates providing easy access to often-used definitions such as t:- cleanup, test(m,a).

3.1. The "evalSig" Procedure

The predicate evalSig is invoked with Sys and Sig fully instantiated. If an outputEqn (Sys, Sig:=Func) exists, Val is instantiated to the instance of Func obtained by unification in outputEqn with the instance of Sig in the head of evalSig. Otherwise a recursive signal trace is done to outputEqns and stateEqns of interior subsystems often more than one level deep. In this case outputEqns and stateEqns for Sys are asserted with the instance of Sys present as an argument of Sig replaced by SysVar, which is usually uninstantiated. Any necessary stateEqn stateDef and stateMap predicates are also asserted.

```
evalSig(Sys, Sig, Val, Type, SysVar, Trail, Trace):-
  nonvar(Sys), module(Sys),
  findcase(Sys, Sig, Type, Sys1, Instance_of_Sys1, Dir, Case),
  (  
    (  
      % normal case no dataflow cycles  
      not member(Sig, Trail),  
      docase(Case, Sys, Sig, Val, Type, Sys1,
```
Instance_of_Sys1, SysVar, Trail, Trace)
);
(
  % this case breaks a connection and inserts a state
  % a suitable decision could have been included in
  % findcase, but since this is a very special case
  % it was left in evalSig
  docase(insert_state, Sys, Sig, Val, Type, Sys1,
         Instance_of_Sys1, SysVar, Trail, Trace)
)
).

When entered, Sys must a module descriptor; that is, it must be instantiated and
module(Sys) must succeed. Sig must be the Source or Dest of a unit clause in the database
connected(System, Source, Dest).

System must be either Sys or the module identifier (Sys1) of a part of Sys. Trail must be
instantiated to a list in a later version of the program not given here, and will record a path
already followed through the dataflow. SysVar is uninstantiated; it is used in constructing the
outputEqn for Sys.

When evalSig succeeds, Val should have been instantiated to the value of Sig, Type
should have been instantiated to its datatype, an outputEqn for Sig in Sys will have been
asserted if one did not exist when evalSig was invoked, and any necessary stateDefs, sta-
tMap, and stateEqns needed to compute Sig will have also been asserted.

As can be seen from its definition, evalSig consists of two goals: (1) a case analysis to
determine the nature of Sig, and (2) one of a number of possible computations for Sig, depend-
ing on the results of the case analysis. The search for cyclic dataflow happens to be done in
evalSig for the code as published here. It could almost equally well be divided between
findcase(...) and docase(...) if desired.
3.2. The PROLOG Clause for "findcase"

The clause for findcase follows. It expresses the various conditions on Sig corresponding to the preceding syntactic axioms, which lead to use of different computations for Sig.

findcase(Sys, Sig, Type, Sys1, Instance_of_Sys1, Dir, Case):-

( % if Sig is a port
  port(System, Sig, Dir, Type),

  ( % Sig was a portname of Sys
    Sys = System,
    ( % case of an outer system
      (Dir=input,Case=external_input);
      (Dir=output,
       (outputEqn(Sys, Sig := _),
        Case = has_outputEqn
       );
        Case = external_output
      )
    )
  )
)
)
)
)
); % end of cases for ports

( % if Sig is a state variable for Sys
  stateDef(Sys, Sig, Type),

  ( % Sys has a stateEqn
    stateEqn(Sys, Sig := _),
    Case = has_stateEqn
  );

  ( % Sys has only a stateMap
    stateMap(Sys, Sig, Sys1_Sig),
    arg(1, Sys1_Sig, Instance_of_Sys1),
    part(Sys, Instance_of_Sys1, Sys1),
    Case = no_stateEqn
  )
)
);
(% Case where Sys1 has a stateEqn
  functor(Sig, _, 1), % a signal has only one argument
  arg(1, Sig, Instance_of_SYS1),
  part(Sys, Instance_of_SYS1, System),
  Sys1 = System,
  stateEqn(Sys1, Sig := _).
  Case = sys1_has_stateEqn
 );
 % various constants
 (integer(Sig),
  Case = integer,
  Type = integer
 );
 (constant(Sys, Sig, _ Type)
 );
 % anything else must be a function
 Case=func
 ).

The various cases correspond to computations as described below.

**external_input.** Sig is an input port at the boundary of a module. The computation should terminate recursion. Sig should be such that port(Sys, Sig, input, DataType) succeeds. Sig should have the form <signame>(Instance_of_SYS). The value returned by evalSig should be <signame>(SysVar).

**has_outputEqn.** The term Func in outputEqn(Sys, Sig := Func) should be unified with Sig, and returned as Val.

**no_outputEqn.** There is no outputEqn(Sys,.....), but the Source for Sig is the output of a subsystem of Sys. The dataflow through subsystems of Sys should be traced by use of connected(.....) and recursion using evalSig, until the recursions terminate at inputs to Sig, constants, or state variables. On return from the recursion, Val will have been instantiated to the term Func of an outputEqn(Sys, <signame>(SysVar) := Func) and the outputEqn can be constructed and asserted.

**internal_input.** A connected(Sys, Source, Sig) clause leads to Source, which must be evaluated by evalSig.

**internal_output.** Sig is an output of a subsystem of Sys. Either the subsystem has an outputEqn, or it does not. If it does not, and evalSig(Sys1,Sig,.....) is invoked, one will be constructed. Once this has been done, and evalSig(Sys, Func,.....), where Func is the Func of the subsystem outputEqn, will trace inputs back to the boundary of Sys.

**has_stateEqn.** If Sys has a stateEqn(Sys,.....), then recursion terminates and the state variable is returned.
sys1_has_stateEqn. Sys has no stateEqn, stateDef, or stateMap for Sig, but Sig is a state variable for a subsystem. Here a unique state variable name for Sys must be constructed, stateDef(Sys, <state_name>(SysVar),DataType) asserted where DataType is the type of the stateVar in the subsystem. Once this has been done a stateMap(Sys, <state_name>(SysVar), State1) must be asserted, where State1 is Sig with the instance of Sys within it replaced by SysVar.

no_stateEqn. An outputEqn has just been constructed for Sys, and a stateDef and stateMap for Sys may have been asserted by the case sys1_has_stateEqn. If this is true, the third argument of stateMap(Sys, StateVar, State1) must be evaluated by evalSig. The value returned is the Func of a stateEqn(Sys, StateVar := Func) this case must assert.

integer. Integers are recognized, and terminate a recursion by returning their value in the Val argument of docase(....).

constant. System-dependent constants, which may depend on the Instance_of_Sys through its presence in the Sig argument, have their values determined through a constant(Sys, Sig, Val, Type) unit clause.

function. The case of <function_name>(Arg1, Arg2, ..... ArgN) is processed by evaluation of the arguments one by one, using evalSig, and the return of a value <function_name>(Val1, Val2, ..... ValN).

cyclic dataflow. When a cyclic dataflow or insert_state is encountered, the recursion must be terminated or the evaluation of the outer signal will not terminate. This termination is done by insertion of a state variable. Five conditions must be met:

- **Condition for non-termination.** If a computation of an output leads to recomputation of the same output, the evalSig recursion will not terminate.

- **Condition for termination.** Since the system has only a finite number of outputs, if no output appears twice in the history of a dataflow computation, the computation must terminate with the processing of all outputs.

- **Termination decidable within a single level in the hierarchy.** A computation of an output of some instance of Sys, if it does not terminate, will fail because of a cycle among the network of subsystems comprising the instance, or within a subsystem. If the cycle occurs within a subsystem, then it will be recognized by an evalSig for that subsystem.

- **UsedLists needed only within a level.** Because termination can be decided within a single level of a hierarchy, a list of used nodes need only be kept within that level, and a new one can be started for each subsystem.

- **The trail of nodes.** Owing to the restriction to a single level, and the termination of a recursion at an input to that level, the only outputs kept on the trail of nodes need be internal outputs of the level.
In cases where a single output has connections to several inputs a slightly more complex situation arises. The "connected" clause where the cycle was found must be replaced by an inserted state. No signals already computed need to be changed since their computations must have terminated. Any subsequent computation involving the output where a state was inserted and not having a cyclic dataflow will be computed from the "old" output, not this "inserted" output. If a subsequent computation finds a cycle at the same point, either it may be sourced from the inserted state (preferable) or have its own state inserted (easiest to do).

If a function value is to be computed, then the Trail when the function value is requested is the relevant trail for determination of cycles. A separate trace of outputs seen must be kept for each function argument. When the function computation is complete, these traces must be concatenated and multiple signals arising from different arguments removed. In the PROLOG used here, the "sort" intrinsic performs this task. If no suitable intrinsic is available, an appropriate sort must be written. Since the lists will be quite small, and possibly already partially ordered from inner function evaluations, an insertion sort seems suitable [Clocksin and Mellish 1981].

The computational procedure is as follows. A Trail list is kept of output signals evaluated by evalSig. If the Sig argument of evalSig is a member of this list, then a cyclic dataflow is present. Since for internal dataflows either Dest or Source of a connected(Sys, Source, Dest) predicate uniquely determines the other, and Sig is an output when the cycle is found, the input destination can be found and the connected(Sys, Source, Dest) predicate replaced by a connection from Source to the input of a synthesized module containing a state variable of the same datatype as Sig, and whose output is connected to the old Dest. A stateDef and stateMap for Sys are asserted, just as if a state variable in the system definition had been encountered, and the value of the state variable returned. When the docase(external_output, Sys,...) completes, a state variable for Sys is asserted just as if it had been present from the start.

3.3. The PROLOG Procedure "docase"

/*************************************************************************/
outline code for docase
Invocation:-
    docase(Case, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace)
*************************************************************************/

/*************** CASES FOR Sig is a port ***************/
docase(external_input, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
% should unify Trail and Trace because it terminates recursion
%external_input
    Trace = Trail,
    (var(SysVar),
     functor(Sig,F,1),
     Val =..[F,SysVar]
    );
(% case where SysVar is an Instance of Sys
nonvar(SysVar),
arg(1, Sig, SysVar),
Val = Sig
).

docase(internal_input, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
% internal_input
  connected(Sys, Source, Sig),
evalSig(Sys, Source, Val, Type, SysVar, Trail, Trace).

docase(has_outputEqn, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
  outputEqn(Sys, Sig := Val).

docase(external_output, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
  % we must make an outputEqn by going down one level in
  % the hierarchy.
  % find a source for Sig
  connected(Sys, Sig1, Sig), %Sig1 is a source
  % because findcase did not follow the connected predicate,
  % we must find Sys1, etc
  port(Sys1, Sig1, output, Type),
  arg(1, Sig1, Instance_of_Sys1),
  part(Sys, Instance_of_Sys1, Sys1),
  % evaluate Sig1 in Sys1
  % if there is an outputEqn(Sys1, Sig1 := Func)
  % evalSig will use it, otherwise one will be made
  % by recursion to this this case for Sys1
  % start a new Trail/Trace stackframe
  evalSig(Sys1, Sig1, Func, Type, SysVar1, [], Sys1_Trace),
  % Func will have arguments being inputs to Sys1
  % evaluate them in Sys
  % and record the output of Sys1 on the trail
  % since we have used an evaluation of Sig1, we must put it
  % on the trail for Sys so as to avoid circular traces in
  % finding outputEqn for Sys, Sig
  evalSig(Sys, Func, Rhs, Type, SysVar, [Sig1|Trail], Trace),
  % make the Lhs from the principal functor of Sig
  functor(Sig, SigName, 1),
  Lhs =.. [SigName, SysVar],
  assert(outputEqn(Sys, Lhs := Rhs)),
  % unify the Rhs with Sig to obtain Val
  outputEqn(Sys, Sig := Val),
( % see if we need a stateEqn made
stateDef(Sys, State, Type),
% fully instantiate State
arg(1, Instance_of_Sys1, Instance_of_Sys),
arg(1, State, Instance_of_Sys),
findcase(Sys, State, Type,
  Instance_of_StateSys1, _, no_stateEqn),
% start a new Trail/Trace stackframe for a state variable
docase(no_stateEqn, Sys, State, Val, Type, StateSys1,
  Instance_of_StateSys1, SysVar, [], State_Trace)
)
;true).

docase(internal_output, Sys, Sig, Val, Type, Sys1,
  Instance_of_Sys1, SysVar, Trail, Trace):-
% Sig is an output of a subsystem Sys
% evaluate Sig in an outputEqn for Sys1 if possible
% otherwise build an outputEqn and evaluate it
(outputEqn(Sys1, Sig := Func);
  (evalSig(Sys1, Sig, Func, Type, _, [], Sys1_Trace),
   outputEqn(Sys1, Sig := Func))
).
% Func has arguments being inputs to Sys1
% The following will trace these to outputs of other
% subsystems of Sys
evalSig(Sys, Func, Val, Type, SysVar, [SigTrail], Trace).

/******* CASES INVOLVING STATE VARIABLES **********/

docase(insert_state, Sys, Sig, Val, Type, Sys1,
  Instance_of_Sys1, SysVar, Trail, Trace):-
% This case arises when Sig is a member of the list Trail
% The connection leading to Sig must be broken and a
% state variable module inserted
  assert(dataflow_cycle(Sig, Trail)),
  replace_connection(Sys, Sig, State_Output, Type),
evalSig(Sys, State_Output, Val, Type, _, Trail, Trace).

docase(external_stateEqn, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar,
  Trail, Trace):-
% Since this ends a recursion, bind Trace to Trail
% If we got to here, the recursion ends, and, having found
% Sig is a stateVar for Sys, set Val = Sig and return
  Trace = Trail,
  stateEqn(Sys, Sig := Val),!.
docase(sys1_has_stateEqn, Sys, Sig, Val, Type, Sys1, 
    Instance_of_Sys1, SysVar, Trail, Trace):-
% In this case Sys1 has a stateEqn, but Sys has no stateDef, Map or Eqn
    functor(Sig, F1, 1),
    % Sys1 state name is F1
    % we also need the name of the instance of Sys1
    arg(1, Sig, Instance_of_Sys1),
    functor(Instance_of_Sys1, Name_of_Instance_of_Sys1, 1),
    % build a unique statename for Sys
    build_state_name(Sys, StateName),
    % Val is <statename for Sys> (SysVar)
    Val =..[StateName, SysVar],
    % the third argument of the stateMap will have
    % principal functor F1 and argument the generalization
    % of the instance of Sys1
    functor(MapArg, F1, 1),
    MapSys =..[Name_of_Instance_of_Sys1, SysVar],
    arg(1, MapArg, MapSys),
    % now assert the stateDefs using assertz so that innermost
    % appear first
    assertz(stateMap(Sys, Val, MapArg)),
    assertz(stateDef(Sys, Val, Type)),
!.

docase(no_stateEqn, Sys, State, _, Type, _, _, Trail, Trace):-
% a subsystem has a stateEqn, and a stateDef(Sys,...) and
% stateMap(Sys,...) have been asserted, but the stateEqn(Sys,...)
% needs to be asserted
    stateMap(Sys, State, Sys1State),
    stateDef(Sys1, Sys1State, Type),
    % build the Lhs
    functor(State, StateName, 1),
    Lhs =..[StateName, SysVar],
    % find the lower level value of the state variable
    stateEqn(Sys1, Sys1State := Sys1Func),
    % evaluate it in Sys to give the Rhs of stateEqn(Sys,...)
    evalSig(Sys, Sys1Func, Rhs, Type, SysVar, [], Sys1_Trace),
    assert(stateEqn(Sys, Lhs := Rhs)),!,
    % allow for the possibility that evalSig may have
    % asserted more stateDefs etc
    % recurse through any stateDefs w/o Eqns
    do_next_state(Sys),!.

/*************** INTEGERS AND CONSTANTS *******************/
% These end recursions so bind Trace to Trail
docase(integer, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, 
    Trail, Trail):-
% integer
    Val=Sig.
docase(constant, Sys, Sig, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trail):-
%constant
custom(Sys, Sig, Val, Type).

/******* CASE FOR FUNCTION Sig=f(Arg1, Arg2, ....) **********/

docase(function, Sys, Func, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
% if not any of the above, it must be an ordinary f(g(h(...)))
% treat arity 1 specially
  functor(Func,F,1),
  arg(1,Func,Arg),
  evalSig(Sys, Arg, ArgVal, Type, SysVar, Trail, Trace),
  functor(Val, F, 1),
  arg(1, Val, ArgVal),!.

docase(function, Sys, Func, Val, Type, Sys1, Instance_of_Sys1, SysVar, Trail, Trace):-
  functor(Func, F, Arity),
  functor(Val, F, Arity),
  evalArgs(Arity, Sys, Func, Val, SysVar, Trail, Trace, []).!,

docase(function, _ , _, _ , _, _, _, _, _):-
  nl,nl,nl,write('docase(function,....) FAILED. Entering trace'),
  nl,nl,nl,trace.

3.4. The Utilities

Following are the procedures performing various operations which are seen more clearly if separated from their invocations.

/******* UTILITIES *****************/
evalArgs(0, _ , _ , _ , Trail, Trace, Arg_Trace):-
  % remove duplicates from Arg_Trace
  % and return as Trace, the set of dependencies of all arguments
  sort(Arg_Trace, Temp), % remove duplicates
  append(Trail, Temp, Trace).

evalArgs(N, Sys, Func, Val, SysVar, Trail, Trace, Arg_Trace):-
  arg(N, Func, Arg),
  evalSig(Sys, Arg, ArgVal, Type, SysVar, Trail, ArgN_Trace),
  arg(N, Val, ArgVal),
  N1 is N-1,
  append(Arg_Trace, ArgN_Trace, Next_Arg_Trace),
  evalArgs(N1, Sys, Func, Val, SysVar, Trail, Trace, Next_Arg_Trace).
do_next_state(Sys):-
  ( % search for the next stateVar w/stateDef, stateMap & no stateEqn
    stateDef(Sys, NextState, _),
    not stateEqn(Sys, NextState := _), % if stateEqn fail ASAP
    stateMap(Sys, NextState, _),
    evalSig(Sys, NextState, _, _, [], Trace),! % if no next state end the recursion )

make_state_module(Type, Sys):-
  typed_state_module:(Type, Sys);
  ( % make the name Type_state_module
    name(Type, Type_String),
    append(Type_String, ".state_module", Sys_String),
    name(Sys, Sys_String),
    % assert the attributes of the typed state module
    assert(typed_state_module(Type, Sys)),
    assert(module(Sys)),
    assert(port(Sys, in(_, input, Type)),
    assert(port(Sys, out(_, output, Type)),
    assert(stateEqn(Sys, statevar(S) := in(S))),
    assert(outputEqn(Sys, out(S) := statevar(S))),
    assert(stateDef(Sys, statevar(_, Type))))).

make_instance(Sys, Sig, Type, State_Sys, Instance_of_State_Sys):-
  % make a typed state module
  make_state_module(Type, State_Sys),
  % make a unique name for the Instance of State_Sys
  unique_name(State_Sys, I_Name),
  % and make the instance with Outer_Sys uninstantiated (SysVar)
  arg(1, Sig, Temp),
  arg(1, Temp, Instance_of_Sys),
  functor(Instance_of_State_Sys, I_Name, 1),
  arg(1, Instance_of_State_Sys, SysVar).

replace_connection(Sys, Sig, State_Output, Type):-
  % find the outer system from Sig
  arg(1, Sig, Old_SubSys),
  arg(1, Old_SubSys, Outer_Sys),
  % make all the data structures for an instance of a typed
  % state module
  make_instance(Sys, Sig, Type, State_Sys, Instance_of_State_Sys),
  % instantiate the SysVar argument of Instance
  arg(1, Instance_of_State_Sys, Outer_Sys).
% instantiate the input and output signals
port(State_Sys, State_Input, input, Type),
port(State_Sys, State_Output, output, Type),
arg(1, State_Input, Instance_of_State_Sys),
arg(1, State_Output, Instance_of_State_Sys),
% make the new connection
retract(connected(Sys, Sig, Dest)),
assert(connected(Sys, Sig, State_Input)),
assert(connected(Sys, State_Output, Dest)),
assert(part(Sys, Instance_of_State_Sys, State_Sys)).

unique_name(Atom, Name):-
  atom(Atom),
  unique(Atom, N),
  name(N, NName),
  name(Atom, Atom_String),
  append(Atom_String, NName, NameList),
  name(Name, NameList).
unique_name(String, Name):-
  name(String_atom, String),
  unique(String_atom, N),
  name(N, NName),
  append(String, NName, NameList),
  name(Name, NameList).
unique(Type, A):-
  retract(unique_number(Type, N)),
  N1 is N + 1,
  assert(unique_number(Type, N1)), !,
  A is N1.
unique(Type, 0) :-
  assert(unique_number(Type, 0)).
build_state_name(Sys, StateName):-
  name(Sys, Sysname),
  append(Sysname, "_state_" , Sname1),
  name(Sname, Sname1),
  unique(Sys, Number), % unique number for Sys
  name(Number, NumList), % convert to list
  append(Sname1, NumList, State_Name_List),
  name(StateName, State_Name_List).
The following utilities use H. Barrow's pp(...) pretty print utility, which because it is copyright, is not presented here. It may be obtained from the PROLOG archive at Stanford in machine readable form.

For simple experiments, pp is not needed, and may be replaced by

\[ \text{pp}(X) \leftarrow \text{write}(X), \text{nl}. \]

----------

:- op(500, fx, 'vi').
:- op(500, fx, 'edit').
:- op(500, fx, 'less').
:- op(500, fx, 'lpr').
:- op(500, fx, 'set').
:- op(500, fx, 'unset').

\text{test}(\text{Sys}, \text{Id}) :-
  \text{listing}(\text{part}),
  \text{listing}(\text{connected}),
  \text{nl}, \text{write}(\text{"go?")},
  \text{getO(Junk)},
  \text{port}(\text{Sys}, \text{Sig}, \text{output}, \text{Typ}),
  \text{test1}(\text{Sys}, \text{Id}, \text{Sig}, \text{Typ}),
  \text{fail}.

\text{test1}(\text{Sys}, \text{Id}, \text{Sig}, \text{Typ}) :-
  \text{functor(\text{System}, \text{Sys}, 1),}
  \text{arg(1, \text{System}, \text{Id}),}
  \text{arg(1, \text{Sig, System}),}
  \text{evalSig(\text{Sys}, \text{Sig}, \text{Val, Type, \_}), Trace),}
  \text{write(\text{"Evaluate "}, \text{write(\text{System}),nl,}}
  \text{write(\text{"Output = "}, pp(\text{Sig}), nl,}
  \text{write(\text{"Value = "}, pp(\text{Val}), nl,}
  \text{write(\text{"Trace= "}, pp(\text{Trace}), nl,}
  \text{write(\text{"Type = "}, write(\text{Type}), nl,}
  \text{!}.

\text{test(_, _):- write(\text{"Done")}, nl}.

\text{cleanup:}
  \% retract outputEqns for systems having parts
  \text{\_part(\text{Sys, \_})},
  \text{\_do_remooves(\text{Sys}),}
  \text{\_remove_typed_state(_),}
  \text{fail}.
cleanup:-  
    definition_file(Defs),  
    reconsult(Defs).
cleanup:-  
    nl, write('Cleanup done'), nl.
do_removes(Sys):-  
    (  
        retract(unique_number(_,_,_));  
        retract(outputEqn(Sys,_,_));  
        retract(stateEqn(Sys,_,_));  
        retract(stateDef(Sys,_,_,_));  
        retract(stateMap(Sys,_,_,_,_))  
    ),  
    fail.
do_removes(Sys):- write(Sys), write(' done '), nl.
remove_typied_state(Type):-  
    typed_state_module(Type, Sys),  
    remove_use(Sys),  
    fail.
remove_typied_state(Type):-  
    retract(typed_state_module(Type, Sys)),  
    retract(module(Sys)),  
    retract(port(Sys, Input, input, Type)),  
    retract(port(Sys, Output, output, Type)),  
    retract(outputEqn(Sys, Output := State)),  
    retract(stateEqn(Sys, State := Input)),  
    retract(stateDef(Sys, State, Type)),  
    nl, write('Typed state '), write(Sys), write(' removed'), nl.
remove_use(Sys):-  
    retract(part(Outer, Instance, Sys)),  
    retract(connected(Outer, out(Instance), Dest)),  
    retract(connected(Outer, Source, in(Instance))),  
    assert(connected(Outer, Source, Dest)),  
    nl, write(connected(Outer, Source, Dest)), write(' restored').
less(File):-  
    Less = "less ",  
    name(File, F),  
    append(Less, F, Edit),  
    Cmnd = ..[system \ [Edit]],  
    call(Cmnd).
lpr(File):-  
    Func = "lpr -Plw3 ",  
    name(File, F),  
    append(Func, F, SysCall),  
    Cmnd = ..[system \ [SysCall]],  
    name(CmdName, SysCall),  
    nl, write('Calling UNIX command '), write(CmdName), nl,  
    call(Cmnd).
v :- vi_file(File), editor(File).
v :- write('No old vi file present, use vi <file>.'), nl.
editor(File):-
    Vi = "vi ",
    name(File,F),
    append(Vi,F,Edit),
    Cmd =..,[system 1 [Edit]],
    call(Cmd),
    reconsult(File),% restore initial state
    (retract(vi_file(_));true
    ),
    assert(vi_file(File)).
vi(File):-
editor(File).
edit(File):-
editor(File).

/****************** Debugging print utilities **********************/
wait_print(Msg,Term):-
    not tracing;
    (nl,write(Msg),
    pp(Term),nl,
    write('Keypress?'),
    get0(Junk)
    ).
trace_print(Msg,Term):-
    not tracing;
    (nl,
    write(Msg),
    pp(Term)
    ).
/****************** end Debugging print utilities **********************/

/****************** SERVICE UTILITIES **********************/
/* two utilities to assert and retract a flag that always succeed 
and make only one copy. */
set(Flag):-
call(Flag);assert(Flag).
unset(Flag):-
    not retract(Flag).
%findall(X,G,[]):
    %asserta(found('.')),
    %call(G),
    %asserta(found(X)),
    %fail.
%findall(_,_,L):- collect_found([],M),!,L=M.
%collect_found(S,L):- getnext(X),!,collect_found([X|S],L).
%collect_found(L,L).
%getnext(X):- retract(found(X)),!,X = ' '.

/* Error message */
errmsg(Cause, Data):-
    nl,write(Cause),write(' failed, Data was '),
    nl,write(Data),
    trace.

/* append second list to first, return in third */
append([],X,X).
append([H|T],X,[H|Y]):- append(T,X,Y).

member(X,[X|Restoflist]).
member(X,[Y|Restoflist]):-
    not X = Y,
    member(X,Restoflist).

wtargs([H|T]):-
    /* print an argument list */
    write('('),write(H),wtargs1(T).
wtargs1([|]):- write(')').
wtargs1([H|T]):- write(' '),write(H),wtargs1(T).
cma:- write(')').

/******** END UTILITIES *******************/
References


Distribution for ANL-86-51

Internal:

J. M. Beumer (2)
J. R. Gabriel (10)
E. P. Steinberg
A. B. Krisciunas
P. C. Messina
G. W. Pieper (160)
R. W. Springer (2)

ANL Patent Dept.
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (168)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
    J. L. Bona, Pennsylvania State University
    T. L. Brown, U. of Illinois, Urbana
    P. Concus, LBL
    S. Gerhart, MCC, Austin, Texas
    G. H. Golub, Stanford U.
    W. C. Lynch, Xerox Corp., Palo Alto
    J. A. Nohel, U. of Wisconsin, Madison
    D. Austin, ER-DOE
    H. Barrow, Fairchild
    R. Chapman, Wadham College, England (10)
    S. Daniel, Motorola G.E.G. Phoenix
    G. Michael, LLL
    J. Mills, ASU