XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Convergence Properties of Stepwise Refined Sequences of Petri Nets

Ricardo Ferré
Departament of Computer Sciences
Lund University
Sweden

Abstract

This paper consists of two parts.
In the first part, the idea of using top-down, stepwise refined sequences of Petri nets as a tool for program derivation[Fer] is motivated.
The particular need for program derivation in the special case of the construction of large programs is particularly considered in this paper, based on the work of Dana Scott and others.
The SPE classification of Lehman has been used to be able to define what a large program is, using precise and technical considerations.
The life-cycle model is applied to describe the need for change which is built in in P and E programs. The economical demand of avoiding totally new program constructions for each change, motivates an appropriate program derivation to be able to follow the logical steps that lead from a specification to its corresponding implementation in order to restrict the changes only to the necessary steps.
The inferential programming's point of view is given as the most important contribution to clarify the process of program construction.
In the second part, properties of Places/Transitions Petri nets which make them useful for program derivation are studied. The property of a place to be expanded as a whole subnet and conversely, is used here to propose the definition of a partial order for Petri nets. This partial order is proposed to be used to prove convergence properties of sequences of Petri nets, applying the partial order's topology. The limit operation can then be added to these sequences of Petri nets.The stepwise refined sequences that lead from a specification to its implementation are then increasing in this partial order.
These ideas are exemplified in an appendix by giving the Petri net derivation of a parallel regulator and a summary of the whole algorithm for the regulator described in the programming language ADA because of its suitability for the design and description of parallel processes, as an application of these ideas in real-time programming.

The SPE-classification

The definition of the concept 'large program' varies depending on the author. The intuitive definition, that can apply to many cases, takes the number of program lines as the principal factor and establishes several thousand lines as an approximate limit.
It is clear that this depends on many factors, for example the particular programming language in which these lines are written, the particular computer, the compiler, etc.
The SPE-classification[Leh] introduces more technical and precise considerations which are independent of the particular features of the implementation.

S-programs

Its function is formally defined and derivable from a specification. A problem of the real world or of the universe of the discourse is presented as a formal statement which in turn originate a program specification, this one in turn gives a program which brings the solution of the problem.

If we redefine the problem and change the specifications, then a new program must be written, even though the changes in the specifications are small, and therefore the complete correctness of the program must be proved all over again.

P-programs

The problem statement of these programs can not be precise. It is a model of an abstraction of a real-world situation.

The correctness of a solution is determined by the program environment, after comparison with it.

The concern is centered on the value and validity of the solution obtained in its real-world context and therefore can cause a change in the program itself. Typical examples are chess programs, weather prediction programs and programs for reservation system for an airline.

E-programs

The program communicates with the real world of which it has become a part, it is contained an embedded in it. The variables of its implicit function are about the same than those of P-programs, with the addition of predictive views.

In this type of programs the pressure for change is built in.

In this paper we study in particular the application of Petri net sequences to the last two types of programs (P and E).

The production of large programs must consider management aspects to organize and control the engaged personnel, software design methodologies, and the software environment which include software tools to assist the design and maintenance of the program. In the last years we have seen an increased importance of the maintenance aspects, specially in the large systems because that the statistical study has shown that 70% of the total costs are on program maintenance and only 30% in development.

The importance of these data increase in direct proportion with the role of computers in the modern society. A fail in the computer systems for defense of either of the superpowers could lead to the total extermination of the humanity.

In such large programs the following characteristics are specially important:

Reliability - A small error can be extremely costly both economically and in other aspects.

Modifiability - These programs can not be substituted often. The requirement of being easily released and modified implies that they must be well documented, explained and derived. It should be possible to change the internals of one module without requiring changesto the entire system.

Modularization - The job must be divided in modules that can be designed by a group of tens or perhaps hundreds of persons.

Verifiability - It must be possible to show the correctness of the system based on the correctness of the partial modules.

Research in software design methodologies stresses the importance of top-down, structured programming as an effective way facing the special difficulties embedded in large systems.

The requirements for appropriate theory, models, methodologies, techniques and tools are specially hard in these cases.

The life-cycle model

The P- and E-systems are long lived. The cost associated with producing such a system is so high that it is not practical to replace it with a totally new system.

During its lifetime, the system undergoes considerable modification. Both changes in the real-world environment and proved weaknesses of the system itself can determine the need of changes in the system. These considerations origin the concepts of a program's life-cycle and to techniques for life-cycle management.

The life-cycle involves three levels

- Successive generation of system sequences
- The sequence of releases of the system
- The sequence of activities that constitute the life-cycle phases of the development of an individual release.

The grossest level is constituted of system definition, implementation and maintenance.

These three phases require a careful planning in order to achieve long life software and lifetime cost effectiveness.

The laws which govern the dynamics of the evolution process indicate that project plans must be related to dynamic characteristics of the process and the system, and to the statistics of change.

Program derivation

We have already remarked the importance of proving the correctness of large programs and software systems because requirements are specially hard in these systems. It is in this aspect that the concept of program derivation arises as a logical way of showing the correctness of them.

We can define program derivation as the sequence of insights that is required to derive the implementation from straightforward specifications. By representing the program development as a sequence of programs such that the final implementation is reached from the initial specifications by a series of refinement steps we can structure the process so that we can improve both clarity and efficiency.

Conventional mathematical proofs of programs do not use to justify the structure of a program, and they do not help to develop new programs that may be similar to the program that was proved. This implies that they would be useless to the process of programming.

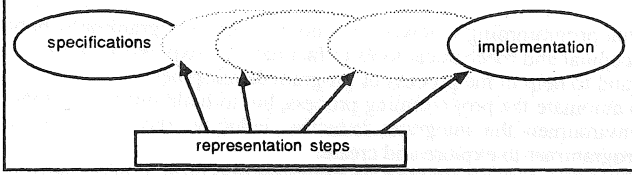
The program derivation as an idealized record of the sequence of design decisions can instead help in both these two aspects.

In the development of new similar programs the sequence of conceptual steps can be followed and changes can be made at the appropriate levels. Also the structure of the program is well clarified by the derivation.

These qualities make them particularly useful when programs must be adapted. This is specially important when we remind that the proportion of program modification, in particular in the case of large software systems is 70% with respect to design and code.

In the traditional framework modification is more difficult because it requires rediscovering concepts used under the development.

fig. 1



It is sufficient to change some steps to adapt a program to new similar specifications.

The derivation structure makes explicit the rational form of the programming structure.

Specifications are different from programs in that they do not constrain how the functionality of an algorithm is to be achieved. It has been proved that this difference is only one of degree,

and that there is a mutual dependency between specifications and programs.

The role of mathematical logic in the construction of large systems

With the aid of program derivation we can get an insight on the program's logical structure, but we can go even further : we can reason about program derivations as formal structures.

In mathematics, it can be reasoned about proofs as legitimate mathematical objects. In the same way, we could reason about program derivations as formal structures representing the evolution of programs.

It has been shown[Sch&S] that the programmers must produce provably correct software, at least in the sense that documentation is generated concerning the extent to which a program is reliable, but even more, we must tend to get a derivation with the right observations and commitments at the right points in the derivation.

Specially in large systems, documentation must be provided that shows the program derivation with all the inferential steps (commitments, deductions, analogies, simplifications) that have led to the particular implementation from the specifications so that necessary maintenance and the future releases would be easier. It is clear that there is a relation between mathematical logic and programming.

It is logical reasoning that leads the derivation steps, consciously or unconsciously.

On the other side, this leads to comparisons between mathematical theorems, proofs and processes and programs, proofs of programs and data processes.

Perhaps the best example of the need for program derivation is the proof of the four-colour theorem. The correctness of the computer program that is an essential part of this proof must also be proved.

The computer-aided check of steps in the proof of mathematical theorems is going to be a increasingly more common and useful resource, as we learn to use the power of the computer. The systems for symbolic computation configure a step in this direction.

The logicians have also shown that every closed axiom-system for arithmetic is insufficient.(Gödel). This destroyed the hope of some mathematicians of constructing indestructible, self-sufficient systems and must bring computer scientists a word of warning.

The inferential programming

We must distinguish the difference between program derivation and inferential programming.

"Inferential programming is the real, somewhat undisciplined and exploratory process of building, manipulating and reasoning about program derivations"[Sch&S].

On the other side "program derivation is the idealized and stylized record of the sequence of design decisions that leads from the specifications to the particular implementation"[Sch&S].

The programming activity involves nevertheless lots of decisions in which aesthetics, among other considerations can influence, and much backtracking, experimentation, revision and many corrections are included.

The purpose of the inferential programming's viewpoint is not to coerce programmers, but that they be provided with its conceptual and mechanical tools to facilitate the expression of the program and its justification and to help in the process of program development.

The meaning is not at all to automate the programming process, but to build interactive tools in an inferential programming environment that integrates deductive, inductive and even analogical facilities, which permit the programmer to explore and create.

The use of powerful modularization techniques for both programs and program derivations will permit them to scale up and be applied to very large software.

These techniques would allow modules of a large system to be derived independently and combined together in such a way that all possible interactions could be anticipated, and should make version control to become a more precise activity, by which changes in a module would

need to be propagated only to those modules that were truly affected.

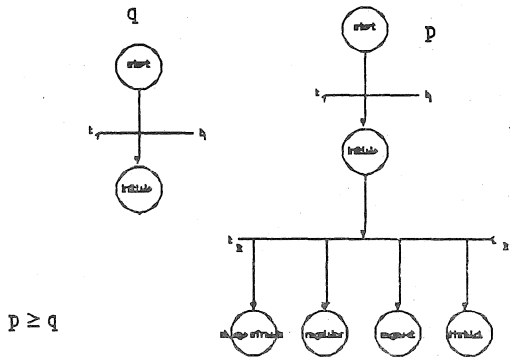
Convergence in a partial ordering

Definition A Petri net p is a refinement of a Petri net q **iff** p is the result of replacing the empty place of q by a proper Petri net.

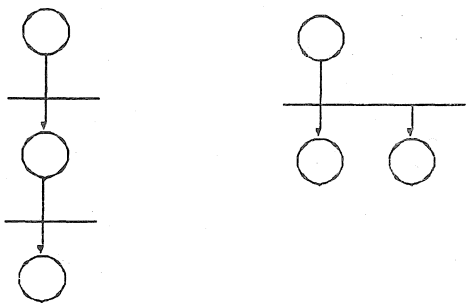
Definition $p \geq q$, where p and q are in a subset of directed Petri nets **iff** q is (equivalent to) a part of p.

We use the notation of Halmos **iff** to mean **if and only if**

Examples



incomparable Petri nets



The properties of the partial ordering $^{\text{Petri}}$

1. $\leq \supseteq$ id reflexive; i.e.: $x \leq x$ for each Petri net x
2. $\leq \supseteq \leq^2$ transitive; i.e.: $x \leq y$, $y \leq z$ imply $x \leq z$ for Petri nets x, y, z

The partial ordering restricted to stepwise refined sequences of Petri nets $x \geq y$ **iff** x is a refinement of y

Theorem A stepwise refined sequence of Petri nets is a chain, that is for every pair p,q in the

sequence, $p \geq q$ or $q \geq p$.

The partial ordering's topology.

There is a natural way of introducing an algebraic topologization through ordering relations[Petri]. In any partially ordered system least upper bounds(l.u.b.), greatest lower bounds(gr.l.b.), limits superior and inferior, and limits can be defined successively, without assuming the existence of none of them. This can be considered as a definition of limits, thus of closure. Besides this, a property closed related to completeness can be given: any partially ordered system can be corupleted so as to ensure the existence of at least: least upper bounds and greatest lower bounds in a certain determinate way. This is based on the specialization of Dedekind cuts for numbers.

To any partially ordered set corresponds a unique complete lattice formed by adjoining as ideal elements all Dedekind sections[Birkh].

Definition Directed sets are functions of a poset .

In this complete lattice, an algebraic topology can be introduced by defining for directed sets x_α, $\lim x_\alpha = x$ to mean:

$$\text{l.u.b}_\alpha \{ \text{ gr.l.b.}_{\beta>\alpha}\, x \cup x_\beta \ \} = x$$

$$\text{gr.l.b.}_\alpha \{ \text{ l.u.b.}_{\beta>\alpha}\, x \cap x_\beta \ \} = x$$

An algebraic topology can then be introduced by letting $\lim x_\alpha = x$ in the original partially ordered set mean that $\lim x_\alpha = x$ in the extended set

Then, the topology of the original poset is a relativization of the topology of the extended set.

The idea of using stewise refined sequences of Petri nets has been also presented by B. Krämer[Kräm] in a more formal way. The convergence properties of these sequences could then be used for these sequences to the derivation of computer programs.

Appendix

The derivation of a regulator modeled with a stepwise refined sequence of Petri nets

In this paper we use stepwise refined sequences of Petri nets as a valuable way of describing program derivation.

We show then the concrete example of the derivation of a parallel regulator, whose final algorithm is described in the programming language ADA, which offers facilities for the treatment of parallelism.

The extensive motivation for the need of program derivation has been sufficiently developed, but it hasn't originated the appearance of tools which are enough formal and that simultaneously permit the programmers experiment the freedom and the joy of creating their works.

This idea is illustrated by presenting the derivation of a parallel regulator whose final version is expressed as an algorithm in the programming language ADA because its possibilities of describing parallel processes.

The simple regulator is a theoretical machine composed by a power source and controls which permit choosing two basic states: automatic and manual.

In the manual state the user can increase or decrease directly the power.

In the automatic state the regulator must obey the equation: $du/dt = k(r - y)$ (integrating regulator).

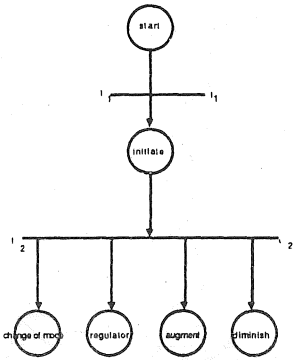
The variable r is called the reference and can be changed manually. The variable k can be read and changed, the same as the sample interval.

We represent the regulator by means of marked P/T Petri nets.

In fig. 1 the transition t_1 is enabled in the beginning of the execution of the regulator by placing a token in the place called *start*. This transitions fires by moving the token to the place called *initiate*.

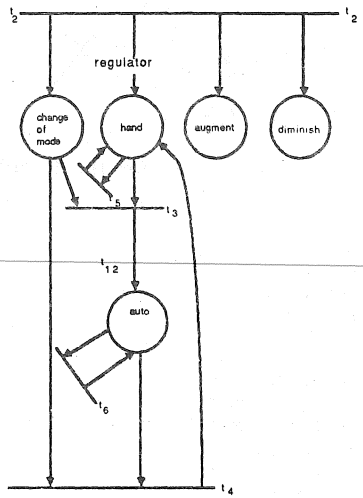
In this marking, the transition t_2 enabled. We fire then the transition t_2 by removing the token from the place *initiate* and placing tokens in all its output places.

Fig. 2



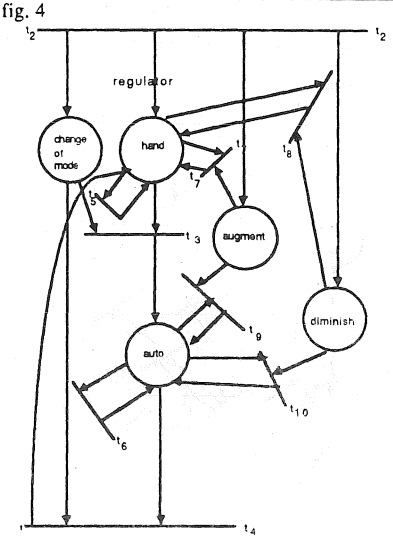
The tasks *change of mode*, *augment* and *diminish* are simple and they influence only the regulator task.

fig. 3



The regulator is composed by two parts with transitions t_3 and t_4 between HAND and AUTO. The transitions t_5 and t_6 are the final of the HAND-LOOP and the AUTO-LOOP in the program. The transition t_4 represents the REG-LOOP.

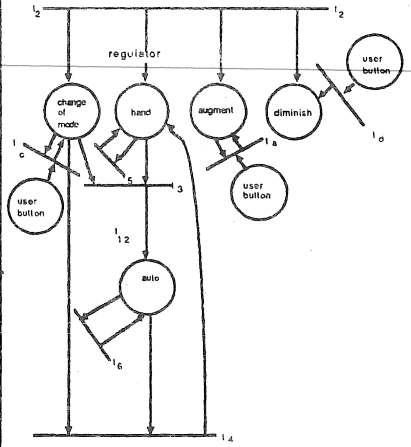
fig. 4



The transitions t_7,t_8,t_9 and t_{10} represent the rendezvous between the regulator and the tasks AUGMENT and DIMINISH.

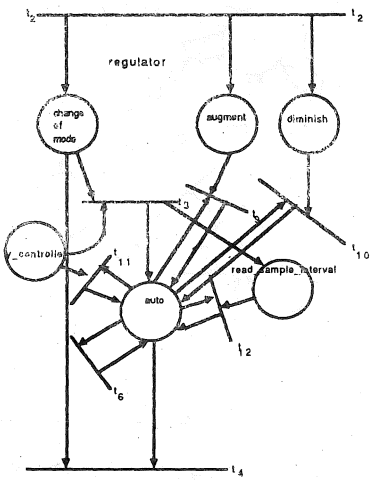
The task CHANGE OF MODE determines the transitions HAND-AUTO and AUTO-HAND(t_3 and t_4).

fig. 5



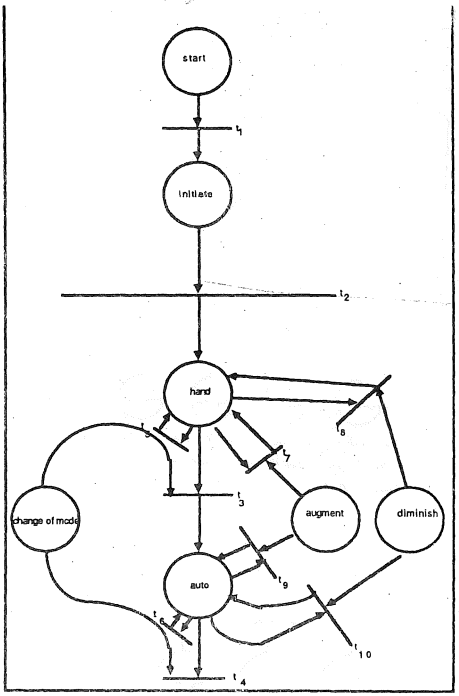
The tasks augment and diminish have actually an interface with the user that who should place a token in his place to allow the firing of the corresponding task.

fig 6 The task AUTO



The transitions t_{11} and t_{12} model the rendezvous between the AUTO-part of the REGULATOR-task and the tasks Y_CONTROLLER and READ_SAMPLE_INTERVAL.

fig. 7 The parallel algorithm of the regulator as a Petri net



The ADA Algorithm for the Regulator

```
   with text_io;use text_io;
   with calendar;use calendar;
procedure main is
-- declarations
package real_io is new float_io(float);
with real_io;use real_io;
type mode_type is (hand,auto);
--- declarations: du_dt ,x,y,l,r,k,change_unity ----
    hand_mode : boolean:=true;
    mode      : mode_type:= auto;
    signal    : character;
    sample_interval : duration:=0.01;
    next_time       : time:=clock;
--- declarations: hand_auto_button, ---
--- declarations: console, sensor, ---
--- declarations: increase_button,
--- declarations: decrease_button,out_device
 procedure initiate is
 -- opens the files, initiates some variables.
 begin
-- open files, etc. ----
 end initiate;
begin -- main
 initiate;
 main_block:
 declare
 task regulator is
```

```
entry change_to_auto(mode :in mode_type);
entry change_outsignal_hand(l : in float);
entry change_to_hand(mode :in mode_type);
entry change_reference(l  : in float);
entry change_y(y : in float);
entry change_sample_interval(
      sample_interval : in float);
end regulator;
task change_of_mode;
task augment;
task diminish;
task body regulator is
begin
  reg_loop:
  loop
      hand_loop:
      while mode=hand loop
       du_dt:=du_dt+l*change_unity;
        select
         accept change_to_auto(mode :
                          in mode_type);
         or
         accept change_outsignal_hand(l :
                                in float);
         else null;
        end select;
      end loop hand_loop;
      auto_loop:
      while mode=auto loop
        du_dt:=k*(r-y);
        select
         accept change_to_hand(mode :
                          in mode_type);
         or
         accept change_y(y : in float);
         or
         accept change_reference(l : in float) do
           r:=r+l*change_unity;
         end change_reference;
         or
         accept change_sample_interval
               (sample_interval : in duration);
         else null;
        end select;
       auto_block:
       declare -- auto_block
        task y_controller ;
        -- reads the actual state of the process
        -- from sensors and transmit it
        --- to the main task
        task read_sample_interval;
        -- reads the sample interval
        --- and transmit it to the
        -- regulator task
        task body y_controller is
        begin
         loop
          x:=y;
          get(sensor,y);
          if x/=y
           then regulator.change_y(y);
          end if;
          delay next_time-clock;
          next_time:=next_time+
                    sample_interval;
         end loop;
        end y_controller;
        task body read_sample_interval is
        begin
          loop
           select
            accept fin do exit;
            else  get(console,sample_interval);
            regulator.change_sample_interval
                          (sample_interval);
           end select;
          end loop;
        end read_sample_interval;
       begin          --------- block ----------
         null;
       end auto_block; --------- block --------
      end loop auto_loop;
    end loop reg_loop;
end regulator;
task body change_of_mode is
-- receives the signal hand/auto
-- and transmits it
-- to the regulator task
begin
  loop
   hand_mode:=not hand_mode;
   if hand_mode
     then
       mode:=hand;
       regulator.change_to_hand(mode);
       read_sample_interval.fin;
     else
       mode:=auto;
       regulator.change_to_auto(mode);
   end if;
  end loop;
end change_of_mode;
task body augment is
begin
  loop
     case mode is
       when hand=>
          regulator.change_outsignal_hand(1.0);
       when auto=>
          regulator.change_reference(1.0);
     end case;
  end loop;
end augment;
task body diminish is
begin
  loop
     case mode is
       when hand=>
          regulator.change_outsignal_hand(-1.0);
       when auto=>
          regulator.change_reference(-1.0);
     end case;
  end loop;
end diminish;
 begin -- main_block
   null;
```

```
end main_block;
null;
```

```
end main;
```

References

Fer R. Ferré 1987 *A Parallel Version of a Regulator with a Petri Net Derivation.* NordDATA 87, Proceedings.

Sch&S W. L. Scherlis, D. S. Scott 1983 *First steps towards inferential programming* I.P. 83 , IFIP 83

Leh M. Lehman 1980 *Programs, Life Cycles and laws of software evolution* Proc. IEEE 68

Wir N. Wirth 1971 *Program development by stepwise refinement..* CACM 14 4

Petri C. A. Petri, E. Smith 1986*Concurrency and Continuity.* Advances in Petri Nets 1987, Springer Verlag, Heidelberg, 1987.

Birkh G. Birkhoff 1937 *Moore-Smith Convergence in General Topology.* Annals of Mathematics, Vol. 36, No. 1, Jan. 1937.

Kräm B. Krämer 1984 *Stepwise Construction of Non-Sequential Software Systems Using a Net-Based Specification Language.* Advances in Petri Nets 1984. Springer Verlag, Heidelberg, 1984.