A System Level Synthesis Framework for Computer Architectures

Oryal Tanir*, V.K. Agarwal and P.C.P. Bhatt

MACS Laboratory, McGill University
* (on loan from Bell Canada)

Introduction:

Current state-of-the-art in design automation presents a slew of design aids for low level logic construction, layout planning and chip design. Synthesis, in its various forms, have been a major contribution in the design automation area. Synthesis is an operation whereby a given specification of system behaviour with its associated set of constraints and goals are used to find a suitable structure that implements the behaviour while satisfying the constraints and goals. This paper will present a framework for system level synthesis and propose a suitable language for capturing design specifications and generating control graphs amiable to synthesis.

Digital system design can be described at different levels of abstraction and synthesis can occur at any one of these design levels. Bell and Newell [BEL] first classified the various levels of computer design and a complete representation of the design hierarchy based upon their classification is presented in table 1. Traditional synthesis and design literature defines various forms of synthesis possible between levels [McF]. Although the terminology used may fluctuate in the literature, these levels are typically defined as (in increasing order of abstraction): i) Logic level, ii) Register transfer level, iii) High level and iv) System level synthesis. It is useful to provide an overview of the different issues at each level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Behaviour</th>
<th>Structure</th>
<th>Physical Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>System (Architecture)</td>
<td>Communicating Process</td>
<td>Processors, Memory, Switch</td>
<td>Cabinets, Cables</td>
</tr>
<tr>
<td>Algorithm</td>
<td>I/O</td>
<td>Memory, Ports, Processors</td>
<td>Board, Rooplan</td>
</tr>
<tr>
<td>Register Transfer Level (RTL)</td>
<td>Register Transfers</td>
<td>ALUs, Regs, MUXes, Busses</td>
<td>ICs, Macro Cells</td>
</tr>
<tr>
<td>Logic</td>
<td>Logic Equations</td>
<td>Gates, Flip Flops</td>
<td>Standard Cell</td>
</tr>
<tr>
<td>Circuit</td>
<td>Network Equations</td>
<td>Transistors, Connections</td>
<td>Transistor</td>
</tr>
</tbody>
</table>

Table 1: The Design Hierarchy

1. Logic synthesis: is the process of generating circuit descriptions from a logic level description of a system. Logic synthesis is usually attributed with dealing with all facets of combinational logic - such as its optimization, verification and testability. Work in this area can be primarily divided into two thrusts: PLA synthesis and Multi-level logic synthesis.

In both cases, synthesis issues are the same: minimize space and delay. PLA synthesis has gener-
ally revolved around minimizing the number of product terms, which usually results in savings in chip area. The research in this area has produced a significant amount of reliable algorithms which can routinely produce optimum or near-optimum results.

Multi-level logic synthesis has more degrees of freedom in the design solution space with the potential reuse of sub-logic. It possesses the promise of smaller layout areas, minimization of critical paths and automatic generation of a complete set of test vectors.

ii. Register transfer level synthesis: implies producing a logic description of the register-transfer level description. Behaviour is described as data transfers and transformations between storage elements. To realize this, the functional units and their data paths are well defined at this level. The basic task is to utilize a data flow graph (or some other form of data representation) and map it to a set of structured register transfer level resources. Issues that arise at this level are design trade-offs such as using a pipeline approach versus combinatorial arrays. Research is oriented towards defining suitable scheduling and resource selection algorithms to solve these problems.

iii. High level synthesis: implies going from an algorithmic level specification of behaviour to a register transfer level structure that implements the behaviour. This usually implies that some form of a dataflow and control precedence graph is generated at this level from a formal description of the behaviour. Behaviour can be expressed by several different languages such as VHDL [VHD], ISPS [BAR] and CATHEDRAL [DeM].

The major issues dominating this area are scheduling and allocation. Once an internal representation of the data flow and control is compiled, synthesis assumes the role of scheduling events in order to minimize the overall execution delay. This is usually done in conjunction with allocation, which defines which components are available for the scheduling task. These two components of synthesis are a major ingredient of the synthesis task in general and will be elaborated upon later in the paper.

iv. System (architectural) level synthesis: implies taking a system level design specification and generating an algorithmic (and sometimes register-transfer) level specification of the behaviour. Current automation is concentrated on the RTL (Register Transfer Level) synthesis domain and lower. The synthesis issues at the architecture (system) level is still a research issue requiring significant innovative approaches.

Although more abstract, the synthesis issues are similar to those found at other levels. The major issues are scheduling and allocation. The scheduling problem is to define and assign operations to control steps whilst meeting given objectives and constraints. With high level synthesis, scheduling implies the proper ordering and planning of operations between hardware (register) components. At the system synthesis level, the problem is transformed into one of creating a control graph and scheduling tasks at a functional level.

Similarly, the allocation problem at the system level is one whereby operations are assigned to hardware blocks. These blocks are essentially functional components, representable by some form of dataflow and control precedence graphs. Allocation at this level entails some form of component binding. This implies defining possible libraries of real-world components that are available to the designer and binding them to particular operations.

The binding operation prevents the synthesis task from allocating realistically unfeasible components to operations. What is significantly different from other levels is that the binding operation also applies to communication channels between components. A communication path can be bound to a set of more rudimentary descriptors. For example, a system level channel defined as a "bus" could be bound to three other channels defined as: "control", "data" and "address". Although communication is explicitly described as occurring through a "bus", the communication will im-
licitly be occurring over the three other communication elements.

System level synthesis is a major design aid. Automation at this early portion of the design stage seeks to identify design trade-offs and problem areas early on in the product life cycle. Dealing with design problems early possesses many advantages reflected by the other synthesis levels, but also additional ones that are amiable to current design practices. These are:

i. Shorter design cycle. Generating a quick yet reasonable design from the very conception of the product can weed out poor design decisions from good ones, thus reducing expensive design reworks later on in the design stage. As a result, the time from system conception to integration is greatly reduced.

ii. Verification of designs; Current designs are verified later in the design stage. At that point, poor architectural decisions can be hidden in the complexity of the overall design details. By synthesis of a system design one obtains a formal representation or model. Algorithms or rules can be applied at this level to test the correctness of the design versus the design specifications. Ideally, synthesis would generate designs that are correct by construction. These lead to less design errors later on in the design cycle.

iii. Solidification of design requirements: Vague or ambiguous design specifications can be identified and corrected during this early stage. Traditionally this has been an area where designers would interpret the specifications. This may not necessarily have been the intent of the specifications.

iv. Procedures for formal design specification: With the advent of system level synthesis tools, formal requirements for design specifications can be defined. Formal specifications following a well defined structure or methodology is currently lacking in the design community.

This paper is centred upon defining a framework for System Level synthesis. The work entails the design of a specification language (a meta-language) that will allow designers to input meaningful information at the system level. The meta-language will then generate a pseudo-model of the system in an intermediate language which can then be exercised and tested.

Once the first stage of simulation and testing is complete (to the designer’s satisfaction), the model will “synthesize” a more detailed lower (algorithmic or RTL) level model. Such a model can be used by lower level synthesis tools to further continue the synthesis process.

Framework overview:

The synthesis framework is designed to work by accepting high level design specifications and exporting a representation amiable for lower level synthesis. The information flow is illustrated in figure 1. Design specifications are captured by a language (DSL). The language distinguishes control flow from data transformation type operations. The latter are operations that affect data (register changes, memory writes, screen updates) whereas the former indicates the ordering of these operations. The control flow representation is transformed into a set of communicating finite-state machines (FSM). The FSMs with the data transform information are then used to synthesize the lower level representation.

The proposed synthesis environment consists of the following stages (depicted in figure 2):

i. The Design Specification stage.
ii. The Intermediate representation (IR) stage.
iii. The synthesis stage.
Each stage represents a completion of a step in the overall system synthesis task. The design specification stage facilitates the representation of the system level design in a simple and meaningful manner. The main means for doing this is with the use of a design specification language.

The IR stage allows simulation and testing of the high level model to occur. At this stage data transformation versus control flow operations are identified and appropriate control graphs for the DSL model are created. These graphs will be used to synthesize communicating finite state machines during the synthesis stage. Several iterations of the model may be undergone at the IR stage before acceptance by the designer.

The final stage is where the synthesis operation is executed. The stage is concerned with binding, scheduling and allocating components. Upon completion of the synthesis, re-simulation of the resulting model is necessary to test its behaviour and characteristics versus the design specifications.

Figure 1: Synthesis Information Flow

Figure 2: System Level Synthesis Framework
i. The Design specification stage:

This stage primarily consists of the Design Specification Language (DSL). The DSL is a high level “meta-language” and is the main interface between the designer and the synthesis environment. The language must be diverse enough to allow a designer to input his design criteria in a straightforward and “natural” manner at multiple levels of abstraction.

At the system level of description, the design specifications are specified at significantly more abstract levels than that of hardware specifications. Hence the DSL should not burden the designer with requirements for inputting data at lower layers of abstraction.

For example, the designer need not concern himself of the details of a particular known protocol for handshaking between two machines. A descriptor defined by the designer such as “send message ‘A’ to destination ‘X’ after ‘Y’ time units” should be sufficient for defining the communication model. The language should be able inject the necessary details of the protocol as required. However, if desired, access by the designers to modelling handshaking or lower level details should also be permitted.

One of the primary functions of the DSL is to accept abstract high level design data and generate a reasonable high level model capturing all the designer’s specifications. The model should also be capable of being exercised with simulation stimuli and amiable to lower level model synthesis.

Considering the different capabilities that the DSL must possess, some basic preliminary requirements for the DSL are given below:

- A loosely typed language is a strong motivation for DSL. Due to the vague and at times ambiguous system modelling terms employed by different designers, various levels of abstraction for an element may exist within the same model. For example elements passing through a communication channel can be referred to as bits, groups of messages or sets of packets. Hence at this early stage in the framework, it is an asset if type limitations on data are minimal.

- The language should have some reasoning power. System level synthesis differs from that of other levels with respect to the type of reasoning that is typically associated with the task. At lower levels, reasoning could be interpreted as different chip area layout optimization, transistor technology suggestions or data path minimization. Applied at the system level, this could be construed as rules to bind possible architectures to given design goals and constraints. It could also be applicable to the decomposition of abstract communication channels into well defined data paths. The synthesis task requires such rules as a potential starting point for the synthesis process to occur. Consequently there should be hooks in the DSL to access libraries of modules that can be used to generate the detailed levels. This implies some form of rules about module interconnection, constraints, speed etc. to be available to the DSL.

Prolog was chosen as the implementation language for DSL since it facilitates programming within the above mentioned framework. Prolog is also amiable to generating meta-languages. The DSL syntax is specified with as few constructs as possible to facilitate the design of the higher level models, yet possess the capabilities of modelling at different levels of abstraction and utilize basic Prolog predicates.

The primary constructs of the DSL are “modules” and entities that interact between the modules (messages). Modules are the primitive model building blocks. They are similar to “objects” in object oriented programming, however they are slightly more abstract. Basically a module is a construct defined to possess a “name”, a behaviour or set of possible behaviours and possible states. This is graphically represented in figure 2(a). In Prolog-like syntax it can be defined as in the left column below while a pseudo-english description is given on the right.
module( Name,  
    Behaviour(1),  
    Behaviour(2),  
    ...Behaviour(n)).

name of the module defined,  
behaviour type 1 specified,  
behaviour type 2 specified,  
etc.  
behaviour type n specified.

The name of a module is a unique and meaningful identifier of the module. The behaviour(s) of the module represent the actions undertaken by the module when an event occurs. The occurrence of an event implies the arrival of a message to the module. The module will then execute the behaviour associated with the message. Behaviour is a procedural description of actions to be performed by a module - such as sending a message to another module or initiating an internal state change by sending an internal message. To support the communications activities, modules contain data structures which facilitate the internal queuing of messages.

Module behaviour can be very complex in nature. As in object oriented constructs, a module can inherit the behaviour of another module or a standard “template” module.

This can be realized with the rule:

inherits(module1, module2).

In the above clause, module1 will inherit the behaviour of module2. Module1 can also possess its own unique behaviour that would be defined within its module statement.

The execution of a behaviour results in a change of module state. Apart from behaviour states, a module possesses two internal states: “idle” and “busy”. Behaviour execution places a module into a busy state, whilst the completion of a behaviour moves it into an idle state. When in a busy state, newly arrived messages are placed in an internal queue and serviced at the next opportune moment. These states are transparent to the programmer.

Modules are “interconnected” through the definition of higher order (HO) modules. These modules are defined graphically in figure 2(b). The Prolog type representation is provided below with a description on the right-hand column of the code:

ho_module ( name,  
    [ Module_1, Module_2,...Module_n]).

int_spec(Module_1, Module_2,  
    (port1.1, port2.1),  
    ... (port1.n, port2.n))

port_spec(Module_1, (port0.1, port1.1) ... )

Define name of ho-module.  
List sub-modules and sub-ho-modules.  
Define the connections between sub modules; these are defined on a pair of modules - port by port basis.  
Define the ho-module connections to the sub modules in a similar fashion.

The list of modules consists of the different modules that are part of the HO module. The int_spec (internal specifications) defines the way in which the various modules are connected.

Connections are maintained through the use of “ports”. A port is a virtual communication channel between the module and its environment. No direction need be specified to the port. The language deduces the direction from the information flow across ports. Port specification are not typed so that different levels of abstracted information may flow through the same port.

The int_spec indicates the module interconnections that are part of the HO module, while the port_spec (port specification) defines how the HO is connected to the internal modules. A hierarchy of HO modules can then be constructed by interconnecting HO modules.
Messages are passed between modules and HO modules using the “send” primitive:

\[
\text{send}(\text{Destination}, \text{Destination} \_\text{port}, \text{Message}, \text{delay}).
\]

\text{Destination} and \text{Destination} \_\text{port} are the destined module and its respective port. \text{Message} is the information that is passed to the destination module. Specifically, it is a request to generate some form of behaviour in response to the message. This may be simply a change in state or other internal variables, or it can be the generation of another message. Temporal characteristics of the model are captured using the “delay” parameter. This parameter is used with the simulator control graph in the IR stage.

As an example of model construction with DSL let us consider a very simple dual processor communication scenario. Our example consists of processor \_\text{a} which accepts two consecutive integers generated by some entity in the “outside” environment, sums them and forwards the result to processor \_\text{b}, whereupon no further processing of the message occurs. A state diagram representation is shown in figure 4 depicting the behaviour of the modules in the system.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Basic DSL Modeling Constructs}
\end{figure}

In our example we assume that a generic module “processor” exists. This low level module possesses the necessary behaviour information to maintain the simple handshaking protocol described by the state diagrams over a single channel.

The code for the generic processor (proc) is:

```prolog
module(proc, 
  (mess_ready:- 
    retract(module_state(Module1, idle)), 
    asserta(module_state(Module1, busy)), 
    send(proc\_b\_, ack\_mess)),
  (ack\_mess:- 
    retract(module_state(Module1, busy)), 
    asserta(module_state(Module1, idle)), 
    send(proc\_b\_, tx\_mess)),
  (tx\_mess:- 
    retract(module_state(Module1, busy)), 
    asserta(module_state(Module1, idle)), 
    retract(buff[\_X]),
    asserta(buff[\_0]))).
)
```

Define the name of module (proc).
Describe the various behaviours of the module:
i.e. Mess\_ready is described as the following:
- Execute the behaviour if the “proc” is in state “idle”.
- Change the proc state to busy.
- send message “ACK\_MESS” to proc\_b (processor b), through any port (does not care about port names).
This module can be reused in different models. It can exhibit a behaviour for three types of messages: mess_ready, ack_mess, and tx_mess. For example, when it receives a “mess_ready” message, it will execute the following actions:

1. It will check its current state. If busy, it will do no further action (message will be internally queued for execution).
2. It will change its internal state to busy.
3. It will send an “ack_mess” (acknowledgement) message.

Processor_b exhibits the identical behaviour of the generic processor, however processor_a possesses two additional distinct behaviours (new_mess(X) and new_mess2) that enable it to interact with the outside world and receive the incoming integers. The following code would depict the processors:

```prolog
module(proc_a, [\]).
inherits(proc_a, proc). 
inherits(proc_b, pm). 

modulefproc-a,
\l
eu new-mess(X):-
\l   retract(buffa(0)),
\l   asserta(buffa(X))),
\l
eu new-mess(Y):-
\l   retract(buffa(Y)),
\l   E is X+Y,
\l   asserta(buffa(E)),
\l   send(proc_a,_,new_mess2)
\l
eu new-mess2:-
\l   retract(module_state(proc_a_idle)),
\l   asserta(module_state(proc_a_busy)),
\l   send(proc_b,_,mess_ready)
\l).

module(proc_b,[\]).
inherits(proc_a, proc). 
inherits(proc_b, proc).
```
In this example, "new_mess(X)" will store a non-zero integer in a temporary local buffer in processor_a (buffa). It will be used to add two consecutive integers together. The result is transmitted to processor_b. The request to transmit is indicated by a "mess_ready" message generated to processor_b. All other internal behaviour for the two processors are identical and inherited from the generic processor description through the "inherits" statement.

Now we require a HO module called "comm_procs" (communications processors) that describes the interconnection of the system:

```
ho_module(comm_procs, [proc_a, proc_b]).
  int_spec (proc_a,proc_b,
             [cont_a,cont_b],
             [data_a,data_b])
  port_spec (comm_procs,proc_a,
             [data_s,data_a])
```

In graphical notation the complete system is shown in figure 5. The model can now be exercised. Typically a simulator will interact with the model. For example, a command such as,

```
send(comm_procs,data_s,new_mess(2),5)
```

will send an integer "2" to the HO module’s data_s port after a delay of 5 time units. This will then propagate to processor_a and the behaviour for new_mess(2) will be executed.

![Figure 5: Dual Processor Example in DSL Representation.](image)

ii. The Intermediate representation (IR) and PetriNet Control Graphs:

After the design has been captured through the DSL, an intermediate representation (IR) is generated to mimic the design criteria. The IR is expected to be in C++ to facilitate interaction with an
existing C++ simulator. The control graph for the model can be input into a Petri-net simulator such as Voltaire [PAR] [TYS] and allow designers to monitor the behaviour of the model. Once the behaviour is verified to be to the modeller's satisfaction, the process can progress to the next stage.

This section is devoted to introducing some extensions to standard petri-nets (PN) in order to facilitate the construction of control graphs for the DSL model. The control graph must have a representation which will allow transformations of sub-sections into finite-state machines that the synthesis stage can manipulate.

Before describing the extensions, the section will first summarize progress in analytical techniques and expressiveness in the petri-net field [NIE] that will be a basis for the new extensions. Petri-nets have been used successfully to prove system level properties [KLJ]. The analytical techniques described will be the basis for formal model and synthesis verification.

ii.1. Place Transition Nets:

A good starting point of interest in the literature is the place-transition net (PTN). The modelling power of a PTN lies somewhere between that of a finite-state machine and a Turing machine. Concurrency can be modelled quite naturally, however there is no concept of an inhibitor as in the case of a Turing machine. PTNs are probably the most widely used PNs and can be analysed using techniques that have been developed. Several variations exists for defining PTNs. The definition we use is (based upon [PET]) as follows:

A PTN is defined as a 4-tuple:

\[
\text{PTN} = (P, T, W, m_0)
\]

where
1. \(P\) is a set of places
2. \(T\) is a set of transitions,
3. \(W \in \{P \times T \rightarrow \mathbb{Z}\}\) is the incidence function
4. \(m_0 \in \{P \rightarrow \mathbb{N}\}\) is the initial marking
5. \(P \cap T = 0, P \cup T \neq 0\)

A place \(p\) is an input place to a transition \(t\) iff \(W(p, t) < 0\) and is an output place iff \(W(p, t) > 0\).

Markers called tokens, move from place to place through the "firing" of a transition. The firing rule for a transition \(t\) in a PTN is satisfied when all input places contain at least one token. The execution of the firing rule results in the removal of a token from each input place connected to \(t\), and the placing of a token in each output place connected to \(t\). There are two popular methods for analysing PTNs: reachability trees and the Invariant method.

The reachability tree (RT) is a tree where each node consists of possible markings for a given PTN. Sibling nodes connected to a parent node define the possible reachable markings from the parent to a sibling.

It is not difficult to see that the reachability tree can be infinite. This is the case where there is a "feedback" of tokens to a "generating" place or if a place is both an input and output place to a transition.

To facilitate analysis, a method is employed whereupon an infinite marking set is mapped onto a finite reachability tree. This is performed by observing sequences of marking that are essentially the same, except for a different number of tokens at a given place. Having more tokens at a place does not change the firing rules (as long as there is a token) and the markings representing these places can be mapped into a single marking. This marking is labelled with a symbol "w" in one of
its places representing an infinite possible of tokens at that place [REI].

A limited but useful set of questions can be addressed using RTs:

**ii.1. Boundedness and safeness.**
A PTN is safe if there is always at most 1 token in each place. A PTN is k-bounded if there is at most “k” tokens within each place - where k is some finite integer value. Basically, if there is no occurrence of the symbol “w” in the reachability tree, the PTN is bounded. If the bound on each place is 1, then the PTN is safe.

The concept of safeness is important in describing and analysing finite-state systems. If a net is bounded it represents a finite state system where the RT is equivalent to a state graph whose nodes corresponds to possible markings. It is important to be able to identify different safe sections or sub-sections in an overall PTN since these sections can then be translated into finite state machines and handled in the synthesis stage.

**ii.1.2. Liveness:**
Liveness is another important property of PTNs. A transition is live if there is a possible firing sequence for it. Alternatively, if a system is not live, there are markings causing deadlocks to occur in the system. Naturally it is useful to be able to identify any such marking.

In general it is difficult to determine liveness of a PTN from a RT. If the PTN is safe, then the system is live if each leaf node in a RT can “fire” a transition. If the system is k-bounded, the analysis can be extrapolated by unfolding leaves containing k tokens in any place k times. However, liveness cannot in general be deduced from a RT if the net is unbounded. This is due to the fact that the introduction of “w” implies a loss of information such as the sequence of firings in the RT.

**ii.1.3. Conservation.**
A PTN is conservative if the total number of tokens in the net is constant. Alternatively, tokens are neither created nor destroyed in the net. A primary observation is that bounded nets are conservative. Conservation can be used to verify known properties of a model represented by a PTN. These properties can be statements such as “the number of resources (tokens) in the system is constant” or “the total number of simultaneous users of a resource cannot exceed k”. The RT can be used to determine conservation (even for nets that temporarily lose tokens) if weights are applied to the arcs of the net. i.e. if a transition requires 2 tokens to fire, but has 1 output place, the resulting token can have a +2 weight. In this manner conservation is maintained if the sum of all the weights for each marking is constant.

**ii.1.4. Reachability.**
A marking m' is reachable from m if there exists a series of valid firing sequences from m to m'. A bounded RT can be used to verify this property (by finding a path from m to m'). However if the net is unbounded, then a RT cannot be used.

To be able to answer some of the outstanding questions invariance and linear algebraic methods can be employed [LAU]. With these approaches, the PTN is given in a matrix notation. Rows and columns represent places and transitions, while the elements of the matrix are the incidence functions for place transition pairs. For example an indicie \((p,t)<0\) implies that p is an input place to transition t. Conversely if \((p,t)>0\) then p is an output place. The value (or weight) of \((p,t)\) indicates how many tokens are removed (placed) from (into) an input (output) place.

If this matrix is called \(M\), then a given transition firing can be represented by \(f=M.x\) where \(x\) is a vector whose non-zero element indicates the “enabled” transition. \(f\) is a column vector of \(M\) for the enabled transition. If we assume a current marking of \(m\), then the resulting marking would be:
\[ m' = m + M \cdot x \]  

[1]

It is important to note that \( m' \) is directly reachable from \( m \) with the firing of the transition defined by \( x \). If \( m' \) is reachable from \( m \) by a sequence of firings, then \( x \) will be a vector whose elements represent the number of times transitions are enabled in the firing sequence.

Reachability questions can be addressed to a certain degree with equation [1]. However, a solution for equation [1] is a necessary but not sufficient condition for reachability. The solution may contain a firing vector \( x \) that is not a valid firing sequence. This is due to the fact that the method does not contain information regarding the sequences or ordering of firings. When output and input places of a transition are the same, the structure is not reflected properly by matrix notation. Questions regarding liveness can also be addressed by examining the different markings (\( m' \)) reachable from a known marking (\( m \)) and solving for \( x \).

Another important linear algebraic analysis approach is that of invariance. An invariant \( i \) is a weighted vector where multiplied by any reachable marking produces a constant value. Hence this implies that:

\[ m \cdot i = m' \cdot i \]  

[2]

It can also be shown that if \( i \) is an invariant and \( M \) is the incidence matrix, then \( M \cdot i = 0 \).

The reasoning behind invariance is that certain properties of a given net hold (or are constant). By examining these properties in terms of linear equations, critical sections in a PN can be analysed.

**ii.2. Coloured Petri-Nets:**

The next level of abstraction is Coloured Petri-Nets (CP) [JEN1]. CPs are based upon High level (predicate-transition) petri-nets [GEN] [HUB]. There has also been significant work performed in terms of abstraction (not as high level as CPs though) and are summarized in [VAL].

CPs add no new modelling power to the PTN, but provide a more compact notation form reducing the number of places and transitions. A CP can always be "unfolded" into a PTN. However, due to the compactness of the information, the number of unfolded states can grow exponentially.

In the CP representation tokens are assigned colours that can distinguish one token from the next. Places, hold coloured tokens, while transitions perform mappings (or some function) on the colours. Arcs from places to/from transitions are also assigned operations defining where different tokens will be placed.

The primary difference in notation with CP and Predicate/Transition nets is that CPs express the PN as functions, whereas the latter uses expressions on arcs. Analytical techniques can be extended to CPs, although in a restrictive manner [JEN2] [JEN3].

**ii.3. Proposed Net Extensions:**

The motivation behind using a petri based graph is to capture control flow information representing correct sequences of data operations and transformations of our system level model. Typically modules will be competing for different resources which are expressed as part of different modules. It should be emphasized that a module processes one "message" at a time and queues messages. A message can also identify data to be used in data transformations. These type of operations require the ability to represent a "client-server" type relationship between messages (client) and the modules (server). Furthermore, the identity of each client is maintained.
In lieu of the above, we require a scheme similar to the colouring of tokens (messages). However, to model the client-server relationship, we do not require the full descriptive power of coloured nets which can unnecessarily restrict our analysis. Also tokens maybe used as a data type for a transition, hence we will refer to our tokens as typed tokens and establish a stricter firing rule: A transition is enabled iff all input tokens are of the same "type".

As with PTNs, maintaining safeness in sub-nets of the petri net is a desirable situation. This will enable us to transform the PN into FSMs later in the synthesis process. Modules cannot process more than one message at a time, hence allowing only safe PN control graphs for modules is a realistic restriction. However, since we have different types of tokens, we will introduce the concept of type-safe.

Definition: A net is defined to be "type-safe" if for every marking \( m \), where \( m(p) \subseteq \lambda \) and \( \lambda \) is the set of all possible types, there exists only one \( x \), where \( x \in m(p) \).

According to this definition, a place can hold many tokens, but only one token of a particular type. As a consequence, the net is bounded by the number of types in the set \( \lambda \).

Conceptually the firing of a transition implies the execution of one or more data transformations in our DSL representation. These transformations can also result in the generation of different typed tokens, hence the transitions represent a transform function \( f(x) \), where \( x \) is an arbitrary type inputted to a transition \( t \). This is really nothing new in terms of representation. Defining a transition in terms of a function is equivalent to saying that it is comprised of a sub-net consisting of PTNs.

Formally our Typed-Petri-Net (TPN) is defined as:

\[
TPN = (P, T, W, fn, m_0)
\]

where

1. \( P \) is a set of places
2. \( T \) is a set of transitions,
3. \( W \in \{P \times T \rightarrow \lambda \} \) is the incidence function
4. \( m_0 \in \{P \rightarrow \lambda \} \) is the initial marking

\( W \) is the incidence function consisting elements of \( \lambda \) (where \( \lambda \) is the set of all "types").

\( fn \) is a set of functions applied to \( T \). Where \( fn(t) \rightarrow \lambda \).

\( m_0 \) is the initial marking. This is a multi-set of \( P \), where \( m_0(p) \subseteq \lambda \).

Upon satisfying the firing condition, the input token of a given type are removed and the output token's type is given by \( fn \) which can be dependant upon a sub-TPN. In this manner, we can consume tokens of one type and generate a different typed token dependent on the consumed token.

Let us define some valid operations with types:

\( A, B, C, \Delta \in \lambda \)

where \( \Delta \) is a token of type "typless".

\( \Delta \) is a special token. It can combine with any other token to satisfy the firing condition. Analogous to the PTN incidence function we can represent token actions \( W(p,t) \) in an incidence matrix with the following notation:

i. \( W(p,t)=+A \) denotes token addition. This implies that a token of type \( A \) is added to an output place.

ii. \( W(p,t)=-A \) denotes token removal. This denotation represents the removal of token \( A \) from an input place.

iii. \( W(p,t)\triangleq A \) denotes the action of removing and adding a token \( A \) to the same place. To distinguish this from the subtraction operation in analysis, we will represent this with the short-
Situations of conflict can arise where two transitions are dependent upon a common input place. As shown in figure 6, transition t1 and t2 are contending for the same input token. We will define a rule to resolve the conflict based on our typed notation. We should note that transitions can either be sensitive to any token or (more frequently) to a set of typed tokens. The set of sensitive tokens will be all tokens that have a negation symbol in their incidence function and consequently in the incidence matrix. Hence if a typed token were to arrive at the input place, the sensitivity of the two transitions will first be checked to determine if the transitions require that type token to fire, the transition sensitive to the input type will be fired. If both transitions are sensitive to the token, then a random selection is made to determine the enabled transition.

![Figure 6: Example of conflict](image)

For analysis purposes we are concerned with the effects on markings. Hence, the following rules of operation are valid:

i. \( A \mapsto 0 \) (The same token being added and removed is equivalent to no change in the token count.)

ii. \( \Delta - A \equiv A - \Delta \)
    \( \equiv \Delta - \Delta \)
    \( \equiv A - A \)
    \( \equiv 0 \)

This rule implies that two identical types or a typless token will combine with a token of any other type.

iii. \( A \pm A \equiv A \) (this follows from rule i.)

iv. \( \Delta + \Delta \equiv 2\Delta \)

v. \( A - B \notin \lambda \). Two different token types cannot combine according to the firing rule.

vi. \( A + A \equiv 2A \). Two identical tokens can reside in a place, however net safeness is lost. Note that \( A + B \in \lambda \). This identifies the occupancy of tokens A and B in a particular place.

The matrix analysis used in PTNs extends to the typed notation. However, there are some denotation differences due to typing of tokens and the transform function \( f_n \). The existence of \( f_n \) implies a sub-net that can replicate that function. For example consider a net represented by the incidence matrices \( C, Proc_A \) and \( Proc_B \):
The three incidence matrices represents the control graphs for our dual processor example. Their respective TPNs are given in figure 7. The places p10, p1_out and p20, p2_out are superfluous places for analysis purposes. They are token holders for passing tokens to the main control TPN C. They are added in here to clarify the token operations.

As an example of the way the information can be analysed, let us look at the first two firing sequences. Initially, transition t1 is enabled. Using our matrix techniques, the next marking would be:

we can use $m' = m_0 + xM$ to find reachable markings from $m_0$. If t1 is fired, $x = [1, 1, 0]$, $m_0 = [new, new]$.

Now t2 and t3 can potentially fire. Let us consider the conditions for t3. From the matrix C, if t3 were to fire, the resulting marking would be:

To determine $fn(t3)$, we must examine matrix Proc_B since $fn(t3)$ is a function determined by this matrix. If $\Delta = new$, there is no possible firing sequence in the matrix Proc_B, since no transitions defined in Proc_B are sensitive to a "new" type token. Hence firing is impossible and this is an invalid marking for the TPN.

Let us now back track and assume t2 is to fire. The new marking would be:

f(t2) is determined by the incidence matrix Proc_A. Examination of the matrix indicates that transition t11 is sensitive to "new" and can fire. The internal marking of Proc_A will be modified from $m_0 = [\Delta, \Delta, 0, 0]$ to $m'_0 = [\Delta, 0, ack, mess_rdy]$ and $fn(t2)$ will be output to place p1_out. In this case $fn(t2) = mess_rdy$. Hence the new marking for the TPN C will be:
m'=[new, mess_rdy].

As seen from the example, the system control operations as represented by DSL is captured by the TPN. An important observation about the TPN models is that the control graphs for the modules (such as Proc_A and Proc_B) will be type-safe TPNs, whilst the overall system control mechanism (such as TPN C) may generally not be type-safe. This is due to the fact that the latter models interactions with the world external to the system model (such as traffic loads) and the message traffic type and rate is generally not controllable by the system.

As a result of the type-safe sub-petri-nets, we can formally analyse the module level control graphs for reachability and liveness. Then the graphs can be passed to the synthesis stage of the framework to be broken down into communicating FSMs.

Figure 7: TPN Representation of Dual Processor Example.

iii. The Design Synthesis Stage:

This stage will incorporate several steps in order to synthesize the final design. These steps have to be further studied to determine their relevance in the overall process. Libraries of existing components can possibly be used by the synthesis tool to complete the structures of the modules within the models. From the control graph, a communicating finite state machine representation can be obtained. With proper state encoding techniques the control portion of the model can be achieved.

Conclusions:

The System synthesis framework requires several stages to fulfil its requirements and goals. Currently, work has been concentrated on defining an appropriate Design Specification language (DSL) and petri-net extensions for control graph analysis. A rough version of the language was used to model a simple system, however further complex examples must be studied in order to ensure all the amiable features are present in the language.

The latter stages of the framework require innovative approaches in solving the current issues. A typed Petri-net partitioning algorithm will be necessary to generate a control graph.
The generation of communicating FSMs from the control and their state encoding will also be a unique contribution to the field.

The research opportunities are sufficiently rewarding to warrant further work to be pursued in this area. The end result can finally bridge the gap between system design to automatic hardware synthesis. This can dramatically shorten the design life-cycle and result in significant savings and profits to design teams.

References:


