AN INVESTIGATION OF COLOURED PETRI NETS: AUTOMATED PART CUTTING CASE STUDY

AN INVESTIGATION OF COLOURED PETRI NETS: AUTOMATED PART CUTTING CASE STUDY

By STEPHEN W. ADAMS, B. Sc.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Master of Science

McMaster University MASTER OF SCIENCE (2015) Hamilton, Ontario (Computer Science)

TITLE:An Investigation of Coloured Petri Nets: Automated Part Cutting Case Study

AUTHOR: Stephen W. Adams, B.Sc. (McMaster University)

SUPERVISOR: Professor Ryszard Janicki

NUMBER OF PAGES: xiv,84

Abstract

Petri nets are a graphical construction with clearly defined semantics which can model concurrent communicating systems in a formal manner similar to the way that automata theory can model formal language theory(Petri, 1962). As Dr. Carl Petri found the existing automata insufficient or too cumbersome for describing communicating systems others have found Petri Nets to be too cumbersome for effectively reasoning about sophisticated, real world systems. In some cases these difficulties were overcome by extending the theory of Petri Nets. Dr. Kurt Jensen developed the theory of Coloured Petri Nets (Jensen, 1981) for the purpose of generalizing and simplifying complex Petri Net models. This work incorporates Coloured Petri Nets and other theoretical extensions to describe a real world automated steel cutting system. During the course of this investigation the paper will formalize colours in the language of algebras and examine patterns related to timing conditions.

Acknowledgements

I dedicate this work to my sons Patrick and Edward; set your goals high and chase them, and let no one come between you and your own realization of success. If we disagree on the form of a successful life remember that you must walk the road that leads you to happiness as I have walked mine. Remind me of this often. To you and to Amanda: thank you for your love, patience and support during my return to the academic world. I have missed you during the long nights in front of the computer as much as I know you have missed me. This is the product of many hours of guidance, patience and support. It would not exist without the community that has helped shape me throughout my life and in particular through my education. To the teachers such as Mr. Michael Mancini who went beyond their jobs to learn about programming and introduce it to the children at St. Helen's Elementary School: loading a room with computers and teaching us BASIC gave my imagination a playground to explore that opened my eyes to the simple pleasure in the act of creation. Transforming that interest into a profession has been helped by many at McMaster University, which I may never have attended if not for the work of many who care about part time students. To Jackie Osterman and the staff at MAPS, thank you. For the professors and teachers who have guided me and provided the moral support I've needed, thank you, in particular to Dr. Jeffery Zucker whose faith in me has inspired me to investigate topics I believed beyond my horizon. Your sense of personal integrity serves as an ongoing inspiration to me. I extend my gratitude for the guidance of Dr. Ryszard Janicki who agreed to act as my supervisor and provided valuable input that has made this thesis possible.

Contents

List of Figures

List of Abbreviations and Symbols

$\mathbb{A}$ the set of printable ASCII characters

$\mathbb{B}$ the set of boolean values, e.g. {true, false}

CNC computer numerical control

CPC compile cycle program

CPN coloured petri net

HCPN hierarchical coloured petri net

HMI human-machine interface

P/T place/transition (net)

PLC programmable logic controller

SML standard metalanguage, an impure functional programming language

Declaration of Academic Achievement

The work of this thesis is solely that of the author, based on research performed by myself during the years 2012–2015. Editing and input has been provided by my supervisor, Dr. Ryszard Janicki and from my committee members. Any substantial contributions from other authors are indicated.

Chapter 1

Introduction

1.1 Petri Nets

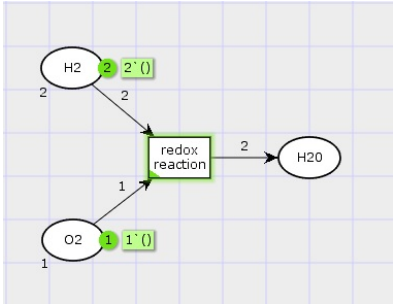

Figure 1.1: A Chemical Reaction Petri Net

Petri Nets are the creation of Dr. Carl Petri, invented in 1939 at age 13 as a method of describing chemical reactions (Reisig, 2010) but they were presented formally as part of Dr. Petri's Ph.D. dissertation as a method for describing communicating information systems(Petri, 1962). Dr. Petri argued that the existing theory of automata based on the work of Kleene and others was insufficient for describing the physical movement of information through a system and that a new type of construction was required(Petri, 1966). This net based construction was required to have the expressive

computational power of prior automata but also satisfy certain restrictions on information density and the speed of information transfer relevant to physical construction of such computing machines(Petri, 1966, pp.52-70,75-82). One argument put forth was that using global state changes, as some automata do, cannot be applied to large, distributed networks and that smaller, local state changes could(Reisig, 2010). Today such networks are a part of our daily computing life, it may be an impossibility to define the global state of the Internet yet perhaps useful models can be created that capture parts of it. The graphical form is simple to understand and manipulate yet the precise, formal mathematical description of the Petri net makes it amenable to analysis.

1.1.1 Graphical Description of a Petri Net

Informally, a Place-Transition Net (P/T Net) is a bipartite graph where indistinguishable tokens residing within the graph nodes are passed between nodes along directed, weighted edges. The graph of nodes and edges represents the static information about the system and the arrangement of the tokens represents the dynamic content. A token represents a single data point within the system and is associated with a particular node. A collection of tokens associated with a set of nodes is called a marking.

The nodes are divided into two groups which we call places and transitions. It is useful to think of places as states and transitions as function blocks which control the consumption and production of tokens. The edges of the graph are called arcs and connect places to transitions and vice versa. An arc which leads from a place to a transition is called an input arc of the transition. An arc which leads from a transition to a place is called an output arc of the transition. Each arc is associated with a natural number called the weighting which represents the number of tokens

consumed or produced when tokens are moved across the arc. A P/T Net in which all the weightings are 1 is called an Elementary Place/Transition Net. A Petri Net generally allows for any natural number to be used and it is not difficult to construct an informal proof of the equivalence between an Elementary P/T Net and a Petri Net.

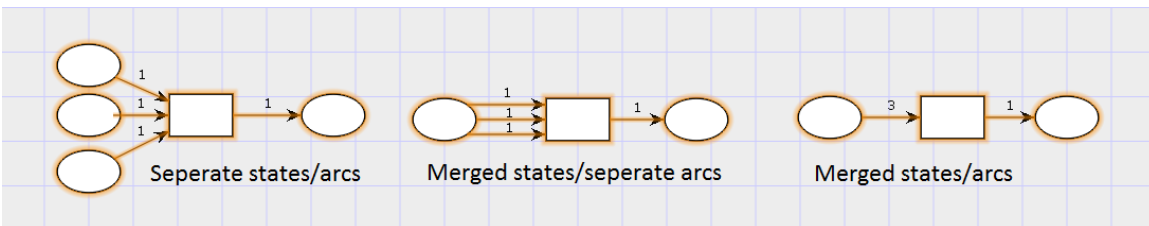

Figure 1.2: Demonstrating arc weight equivalences.

Begin by replacing any weighting with $n \in \mathbb{N}$ with 2 arcs: one with weight 1 and one with weight $n-1$ with the same start and end points as the original arc. Since no new nodes are created we have completed changing the Petri Net into a P/T Net. In the other direction we use the fact that tokens are semantically indistinguishable to assert that places which all provide tokens to a transition are also indistinguishable. If they are indistinguishable then we may merge them into a single state, connecting the initial or terminal points of the arcs of the original states to the new state. Once the states are collapsed the arcs connecting the new place and the original transition become indistinguishable and may also be merged. We merge 2 arcs by creating a new arc with weight $n + m$ where $n \in \mathbb{N}$ is the weight of one arc and the weight of the other is $m \in \mathbb{N}$. This process is visualized in 1.2. With this in mind we are not restricted to a particular formalism but rather can consider any P/T Net whose semantics are appropriate for our modelling requirements and could demonstrate how such semantics are valid with Petri Nets.

Figure 1.3 demonstrates a place-transition net with 5 places and 1 transition.

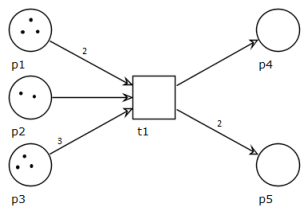

Figure 1.3: t1 Enabled

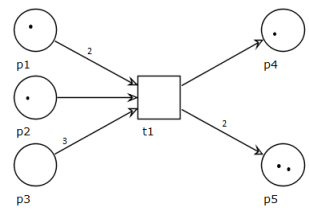

Figure 1.4: after firing

Places p1, p2 and p3 are the input places of t1 and p4 and p5 are the output places of t1. The arc $a_1 = (p1, t1)$ has a weight of 2, $a_2 = (p2, t1)$ has no marked weight and thus is assumed to be 1 and $a_3 = (p3, t1)$ has a weight of 3. A transition is said to be enabled and may be traversed, or fired, when all of the input places have a number of tokens equal to or greater than the weighting of the arcs which connect them to the transition. The input tokens are consumed and new tokens are instantaneously created in the output places corresponding to the weight of the output arcs which connect to them. It is notable that the tokens are created and destroyed but this is often obscured in Petri Nets because of the homogeneity among the tokens. It is an important characteristic and will be much more apparent during our discussion of Coloured Petri Nets. Transition t1 is enabled and can be fired. After firing the appropriate number of tokens are removed from the input places and a number of tokens equal to the weight of the arcs connecting the transition to the output places are created and placed in the corresponding nodes as in 1.4 Thus 1 token is created in p4 to correspond to $weight(t1, p4) = 1$ and similarly 2 tokens are created in p5 to correspond to $weight(t1, p5) = 2$.

Even this simple example is sufficient to raise questions of the more complicated aspects regarding the semantics of Petri Nets. Consider the scenario where the marking m above is replaced with the marking $2 * m$. This provides sufficient tokens such that t1 could fire twice; in what order does the firing occur or is it simultaneous?

Is t1 required to fire or is this a matter of choice? Unless the author of a net has specified a particular policy to answer these questions a reader should assume that the firing is non-deterministic. An enabled transition may fire but is not required to and if sufficient tokens exist it may fire multiple times simultaneously.

1.1.2 Formal Description of a Place/Transition Net

A Place/Transition Net is usually defined as $PTN = (P, T, F, W, M_0)$ (Murata, 1989). The elements of the tuple PTN are defined as:

$P = \{p_1, p_2, ..., p_m\}$ is a disjoint finite set of size m of places (states)

$T = \{t_1, t_2, ..., t_n\}$ is a disjoint finite set of size n of transitions

$F \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs

$W : F \rightarrow 1, 2, 3, ...$ is an arc weight function

$M_n : p \in P \rightarrow 0, 1, 2, 3, ...$ is a marking of tokens to places with M_0 being the initial marking

$P \cap T = \varnothing$ and $P \cup T \neq \varnothing$

If for a fixed j, $\forall i(p_i \in P, t_j \in T) \in F \wedge W(p_i, t_j) \geq M_n(p_i)$ then t_j is said to be *enabled*. Places for which $(p_i \in P, t_j \in T) \in F$ are called *input places* of t_j. Places for which $(t_j \in T, p_i \in P) \in F$ are called *output places* of t_j.

If a transition is enabled then tokens may "move" from the input places to the output place(s) (if they exist) in accordance to the weights specified on the input and output arcs. "Moving" a token from an input place to a transition means that sufficient tokens from p_i are removed and sufficient identical tokens are added to the output places of t_j simultaneously. It is important to note that the set of output places for a particular t_j may contain 0 or more elements, thus it is possible that the overall number of tokens can grow or shrink within the net. Also note that there is

no transitory marking where the token reside in t_j as our definition of a marking only allows for tokens to reside in places.

This process of transition firing occurs concurrently throughout the net and in fact may be concurrent to itself.

The example illustrated in figure 1.3 is defined as:

$P = \{p1, p2, p3, p4, p5\}$

$T = \{t1\}$

$F = \{(p1, t1), (p2, t1), (p3, t1)\} \cup \{(t1, p4), (t1, p5)\}$

$$W(F) = \begin{cases} W(p1, t1) = 2 \\ W(p2, t1) = 1 \\ W(p3, t1) = 3 \\ W(t1, p4) = 1 \\ W(t1, p5) = 2 \end{cases}$$

$M_0 = \{(p1, 3), (p2, 2), (p3, 3), (p4, 0), (p5, 0)\}$

The conditions $P \cap T = \varnothing$ and $P \cup T = \{p1, p2, p3, p4, p5, t1\} \neq \varnothing$ are satisfied. We observe that t1 is enabled as the input places of t1 (p1, p2 and p3) have sufficient tokens as described above. After t1 "fires" the resulting marking illustrated in figure 1.3 is $M_1 = (1, 1, 0, 1, 2)$.

It is worth nothing that a Place/Transition Net can also be equivalently formulated in terms of matrices and vectors which may be more convenient for certain types of analysis.

1.1.3 Computational Expressiveness of Petri Nets

Petri Nets have less expressive power than a Turing Machine in their standard definition (Popova-Zeugmann, 2013). There are several ways to extend the expressive power

to make Petri Nets Turing Complete such as allowing *inhibitor arcs*(Jones, Landweber, & Lien, 1977). Inhibitor arcs have the meaning "do not enable the transition if the condition is true". For example, if an arc connecting a place to a transition has an expression of "1", meaning any token value other than 0 will enable to transition, then this transition would only be enabled exactly when 0 tokens are present if the arc were an inhibitor arc. Indeed it seems that Petri nets fall just below Turing machines in expressive power normally and that many different significant extensions will allow Petri nets to model an undecidable decision problem(Peterson, 1981).

The primary reason that we do not automatically include extensions that increase the expressiveness of Petri nets is that problems such as reachability, which we desire to solve, can become undecidable(Bérard, Cassez, Haddad, Lime, & Roux, 2013).

1.1.4 Motivation for Coloured Petri Nets

Consider a Petri Net in which we want to represent some number of processes that share an arbitrary number of resources. For each process we can construct a net and through careful construction we structure each net's interaction with the resources as arcs connected to a series of places each representing each type of resource. Since tokens within a Petri Net are semantically indistinguishable from one another if they are to represented as distinct resources structure must be introduced to separate the differing types of resource. To demonstrate how typing can be accomplished through structure consider 2 processes labelled process a and process b as illustrated in figure 1.5.

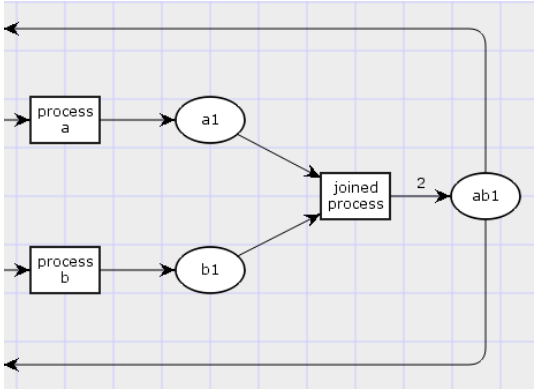

Figure 1.5: Token typing by structure.

Suppose that each of these processes independently control some finite resource and that the tokens in the process represent an available instance of that resource. Consider a process *ab* which requires one resource of each type to execute and which can release the resource back to its controller once it completes. Though simplified here it should be clear that processes a, b, and ab may be arbitrarily complex. This extra structure adds additional complexity to the system and may mask similarities between process or interactions between resources that we might otherwise observe. It is easy to imagine that the graphical representation of such a system could become quite cumbersome as either the number of resources or the number of processes increases. Tokens which contain sufficient semantic information to make them distinguishable from one another either by their type or by their contained values, and a mechanism to select among them, is needed to reduce the overall complexity of these types of Petri Net systems. The introduction of strongly typed, or coloured, tokens results in a reduction of both nodes and arcs.(Jensen, 1997, pp.11-12)

1.1.5 Graphical Description of Coloured Petri Nets

The motivation to provide strongly typed, or coloured, tokens has been to reduce the complexity of the construction of Petri Nets but simply adding a data type or data structure to be associated with the token is insufficient to achieve this goal. As indicated we must also have some mechanism that allows us to distinguish among the token types and to inspect the values assigned to them. Until now we have viewed places and transitions as containers which store and transform tokens. The tokens themselves were only defined as a mapping which referred to the quantity of tokens within a particular place. Likewise, the arc weighting function contained no information about data, it only indicted the quantity of tokens that should be created

or destroyed.

Extending these ideas to include data types requires a number of modifications. First we have to create a set of the available data types which we call the colour set. Next we extend the idea of the mapping function from $M_i = P \times \mathbb{N}$ to a function from P to multi-sets of tokens. We use multi-sets rather than sets as it is possible to create multiple tokens with the same value; the multiplicity of a particular token value is not restricted to 1. Rather than include colour information in the token itself, the places become associated with particular colours instead. Transitions remain uncoloured but gain optional guard expressions: predicate functions which can examine the values of tokens. While the arc expression continues to express the required quantity of tokens it may also examine to the value within token. The arc expression also gains the requirement that the colour of the tokens referred to in it must match the colours of the nodes it connects. Since places are explicitly coloured there is no ambiguity in regard to them. Transitions are uncoloured and so the valid token types must be inferred by an examination of the input and output places.

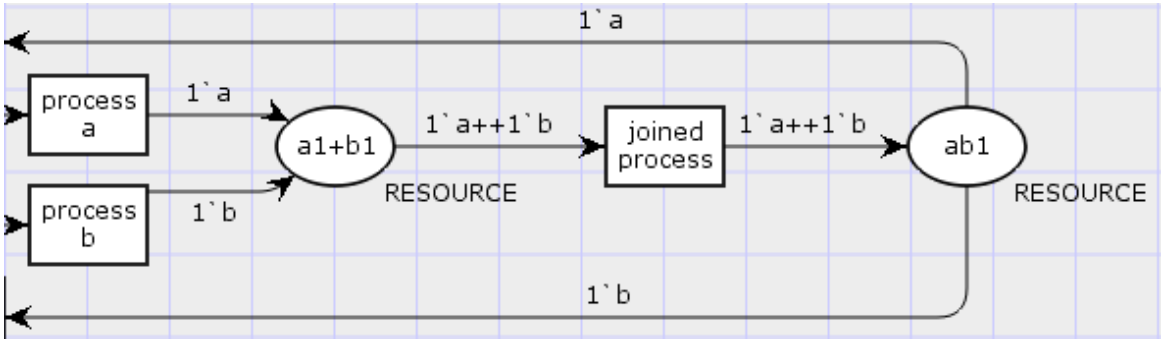

Figure 1.6: Figure 1.5 recast as a CPN

If we recast the example from Figure 1.5 as a CPN we can create a colour called $RESOURCE$ with values $a|b$. Figure 1.6 shows the merged places $a1$ and $b1$ which formed a new place $a1 + b1$. As before processes a, b, and ab1 may be arbitrarily

complex. Merging these two states illustrates that we can remove structure and replace it with semantically based selection. Should the structure of processes a and b be similar, there is potential that many more states could be merged. This CPN did not require the use of a guard function because we are only relying on the presence of a token with the value "a" and a token with the value "b" to become enabled. Multi-set notation (*quantity*`*value*) is used to denote a token. In this example we refer to tokens of colour RESOURCE as 1`a and 1`b to mean one token of resource a and one of b.

1.1.6 Formal Description of Coloured Petri Nets

A Coloured Petri Net is described by a 9 element tuple, where the expressions normally produce multiset values, defined by $CPN = (P, T, A, \Sigma, C, N, E, G, I)$ (Jensen, 1997, pp. 65-78).

$P = \{p_1, p_2, ..., p_m\}$ is a disjoint finite set of size m of places (states)

$T = \{t_1, t_2, ..., t_n\}$ is a disjoint finite set of size n of transitions

$A = \{a_1, a_2, ..., a_p\}$ is a disjoint finite set of size p of arcs

Σ is a set of colour sets, including operators and functions

$C \subseteq P \times \Sigma$ maps colours to places (colour function)

$N : A \rightarrow (p, t) \in ((P \times T) \cup (T \times P))$ maps arcs to nodes

$E : \{e_i \mid i \in \{a_j \in A \rightarrow \mathbb{N}\}\}$ maps arcs to arc expressions

$G : \{g_i \rightarrow \mathbb{B} \mid i \in \{t_j \in T \rightarrow \mathbb{N}\}\}$ maps transitions to predicate guard functions

$I : \{I_i \mid i \in \{p_j \in P \rightarrow \mathbb{N}\}$ maps places to initialization expressions

$P \cap T = \varnothing$ and $P \cup T \neq \varnothing$

Though there are many ways to represent a colour, colours and their associated operations are frequently expressed as SML data types and functions. Although

not required it is useful to think of colours and functions being related or grouped in some manner, akin to the way that classes implement the principle of encapsulation in object oriented programming. SML can provide encapsulation through abstract types, structures or through the modules language extension(Reppy, 2002). Though some feel that SML can represent the object oriented paradigm (Berthomieu, 2000) there is some debate over whether or not some aspects are appropriately modelled (Prof. Robert Harper (Carnegie Mellon), personal correspondence, August 03, 2015). Object orientation is not a requirement of the colour definition however and encapsulation alone suffices. The definitions within this thesis will be presented as SML code.

1.1.6.1 Colours as Algebras

It is common to see colours, as related to CPNs, described by SML data structures. One reason for this is that SML is the supported arc expression language in CPN Tools, a popular tool for creating, simulating and analyzing CPNs. CPN Tools originates from the CPN Group at Aarhus University with much of the work being completed between 2000-2010 by Kurt Jensen, Søren Christensen, Lars M. Kristensen, and Michael Westergaard (*CPN Tools*, 2015).

The formal description does not require us to adhere to SML code for the description of a colour. Since one of the virtues of Petri Nets is the well defined semantics and the applicability of robust mathematical theory to the models, it seems reasonable that colours also be represented in a more purely mathematical sense. For this purpose I propose we instead express colours as algebras. Motivations for doing so include abstracting away the implementation details of any particular computer system, tying together CPN system modelling theory with other correctness testing and specification language theories and also to suggest that techniques related to comparing and manipulating algebras and their properties may lead to meaningful insights

about complex data structures such as those used in industrial applications of CPNs. Redefining the system in a language other than SML raises questions about strict type evaluation at compile time and whether it can potentially be relaxed, for example by using Haskell or another language which more strongly incorporates lazy evaluation of types.

The idea of tying SML to algebras has been explored in the context of whether many-sorted algebras can provide a semantically correct representation of SML programs(Sannella & Tarlecki, 1996). What follows is the simpler task of demonstrating how to represent SML data types as many-sorted algebras for the purpose of specifying colours. A more complete proof is outside the scope of this work.

SML has a number of standard types built into its base libraries. These include bool, char, int, real and string. Each of these types is finite[1] and thus enumerable. Each of these types has a finite number of operators.(Reppy, 2002)

- bool : {*not, andalso, orelse*, =, <>}
- char : {*ord, chr*, =, <>, <=, <, >=, >}
- int : {+, -, *, *div, mod*, ˜[2], =, <>, <=, <, >=, >}
- real : {+, -, *, /, ˜, =, <>, <=, <, >=, >}
- string: {*size*, ˆ[3], =, <>, <=, <, >=, >}

For the interest of brevity I will omit the definitions of each operator and instead construct the algebra with the operator's signatures only as the definitions are neither

[1]SML booleans are valued true or false, numeric types are limited by word sizes specific to machines, characters are 1 byte ASCII encoded values, strings are finite length concatenations of characters. To see that SML strings are finite, let n be the number of distinct values in the alphabet and r be the length of the string, then the total number of strings is $\sum_0^r n^i$.

[2]The ˜operator means negation, e.g., ˜1 means -1

[3]The ˆoperator means concatenation, e.g., "this"ˆ"that" = "thisthat".

sophisticated nor novel. For the choice of carrier sets we could specify that the SML type *real* is carried by the set specified by IEEE 754-1985 or IEEE 754-2008 but it is precisely because these standards change that the carrier was chosen to be the set of real numbers. Similarly for the SML type *int*, the bounds imposed by implementations evolve with word size or on other factors unrelated to the set of integers which are being represented. While these bounds are relevant to some forms of reasoning about our systems, they are artificial and may not be representative of our problem. One indication that this may be true is the addition of the large integer colour set to CPN Tools[4] which is described as having no upper limit(*Large Integer color sets*, n.d.).

Since many of the operators are overloaded so that the same symbol works with a variety of data types the algebra will use a subscripted symbol of the appropriate carrier to differentiate the operators. The use of the Kleene star below should be understood to mean strings to an arbitrary but bounded length and does not describe an infinite set.

With the basic types defined it is now necessary to address tuples as supported by SML. Each element of the tuple is a SML_Base_Type. We can infer the signature of the algebra straightforwardly and thus rely on the definition of product types over a many-sorted signature as provided in "Product types over Σ" (Tucker & Zucker, 1998) to accomplish any necessary tuple definitions. We also gain a mathematical foundation for streams, such as we use when we have an input transition that selects values from a particular sort. This foundation is explained in "Adding streams: Algebras $\bar{A}$ of signature $\overline{\Sigma}$" (Tucker & Zucker, 1998).

With the base types defined as well as tuples and a clear understanding that enumerations are easily representable as a set of constant values and perhaps no

[4]Added in version 3.5.5 which was released on March 07, 2013.

[5]It might be more appropriate to refer to a true language alphabet rather than refer to ASCII but since all the printable characters are valid SML characters this notation seems justified.

algebra	SML_Base_Type
carriers	$\mathbb{B}, \mathbb{A}^5, \mathbb{Z}, \mathbb{R}$
functions	not: $\mathbb{B} \rightarrow \mathbb{B}$
	$andalso,\ orelse$: $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
	ord: $\mathbb{A} \rightarrow \mathbb{Z}$
	chr: $\mathbb{Z} \rightarrow \mathbb{A}$
	$<>_{\mathbb{A}}, =_{\mathbb{A}}, <=_{\mathbb{A}}, <_{\mathbb{A}}, >=_{\mathbb{A}}, >_{\mathbb{A}}$: $\mathbb{A} \times \mathbb{A} \rightarrow \mathbb{B}$
	$\tilde{}_{\mathbb{Z}}$: $\mathbb{Z} \rightarrow \mathbb{Z}$
	$+_{\mathbb{Z}}, -_{\mathbb{Z}}, *_{\mathbb{Z}}, div_{\mathbb{Z}}, mod_{\mathbb{Z}} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
	$<>_{\mathbb{Z}}, =_{\mathbb{Z}}, <=_{\mathbb{Z}}, <_{\mathbb{Z}}, >=_{\mathbb{Z}}, >_{\mathbb{Z}}$: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$
	$\tilde{}_{\mathbb{R}}$: $\mathbb{R} \rightarrow \mathbb{R}$
	$+_{\mathbb{R}}, -_{\mathbb{R}}, *_{\mathbb{R}}, /_{\mathbb{R}}$
	$=_{\mathbb{R}}, <>_{\mathbb{R}}, <=_{\mathbb{R}}, <_{\mathbb{R}}, >=_{\mathbb{R}}, >_{\mathbb{R}}$
	$<>_{\mathbb{Z}}, =_{\mathbb{Z}}, <=_{\mathbb{Z}}, <_{\mathbb{Z}}, >=_{\mathbb{Z}}, >_{\mathbb{Z}}$: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$
	$size$: $\mathbb{A}^* \rightarrow \mathbb{Z}$
	$\hat{}$: $\mathbb{A}^* \times \mathbb{A}^* \rightarrow \mathbb{A}^*$
	$=_{\mathbb{A}^*}, <>_{\mathbb{A}^*}, <=_{\mathbb{A}^*}, <_{\mathbb{A}^*}, >=_{\mathbb{A}^*}, >_{\mathbb{A}^*}$: $\mathbb{A}^* \times \mathbb{A}^* \rightarrow \mathbb{B}$
end	

Table 1.1: An algebra for the basic SML data types.

functions we are ready to formally define a CPN.

This is the formal definition of the CPN presented in figure 1.6.

$P = \{a1 + b1, ab1\}$

$T = \{process\ a, process\ b, joined\ process\}$

$A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$

$\Sigma =$

algebra	RESOURCE
carriers	$\{a, b\}$
functions	none

$C = \{(a1 + b1, RESOURCE), (ab1, RESOURCE)\}$

$$N = \begin{cases} a_1 = (processa, a1 + b1), \\ a_2 = (processb, a1 + b1), \\ a_3 = (a1 + b1, joinedprocess), \\ a_4 = (joinedprocess, ab1), \\ a_5 = (ab1, undefined), \\ a_6 = (undefined, processa), \\ a_6 = (undefined, processb) \end{cases}$$

$$E = \{a_1 = 1`a, a_2 = 1`b, a_3 = 1`a + +1`b, a_4 = 1`a + +1`b, a_5 = 1`a, a_6 = 1`b\}$$

$$G = \{\}$$

$$I = \{\}$$

1.1.7 Equivalence to Petri nets

It is clear that every Petri net can be represented by a Coloured Petri Net by creating a colour that represents all tokens identically and adding the colour information to places and transitions. Any structural semantics are preserved thus no loss nor gain of meaning occurs. We can call such valueless coloured tokens "unit tokens". It is somewhat more surprising that every Coloured Petri Net has an equivalent Petri Net representation(Jensen, 1997).

To sketch the proof of this fact suppose we have some CPN C. For each place in C replace it with one place for each colour in C. Create a new transition for each colour in C. Connecting these new places and transitions to one another by arcs we are left only to join the new structure to the original one using the original arcs. Arc, guard, and initializations expressions translated to the language of P/T nets. For a precise treatment see (Jensen, 1997)[pp.78-85]

A simple CPN P/T net equivalence is demonstrated in figure 1.7. All of the values from the CPN are preserved, with the places being labelled for the reader's convenience. Not all of the places are connected to the transition as those tokens did not satisfy the guard condition which has been rewritten without the colour information as structure. The CPN arc expressions have been replaced by colour free P/T net arc expressions.

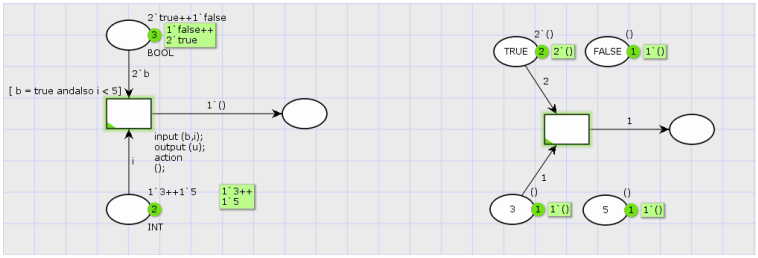

Figure 1.7: CPN-P/TN Equivalence Example.

The conclusion then is that Coloured Petri Nets provide no additional expressive power but instead simplify the structure of Petri Nets. However we can also use the simpler formalism of P/T or Petri Nets for proving that theoretical extensions to Coloured Petri Nets are valid. For example it may be simpler to write proofs regarding the properties of CPNs by proving the property instead about P/T nets and relying on this equivalence to prove the property about the CPN.

1.1.8 Hierarchical Coloured Petri Nets

The motivation to extend CPNs to include some form of modularity comes about from a long standing complaint that P/T nets do not support composition(Jensen, 1997). It is no surprise that particular patterns repeat within the construction of a net just as software design patterns emerge in encoding programming logic. A Hierarchical CPN is a CPN that supports such modularity.

Although formally defined (Jensen, 1997, pp. 107) so as to allow for mathematical analysis, here is presented a sketch of the construction sufficient to understand the

model presented.

A HCPN is a disjoint set of subnets, also called pages. Each element of the set is a non-hierarchical CPN as previously defined but shares no net elements such as nodes or arcs with any other subnet.

To move tokens between the pages there are substitution transitions and fusion places. A *substitution transition* is a transition that transfers a token from a *super page* to an instance of a *sub page*. A *fusion place* is a set of places which are "*fused*"; if a transition would add or remove a token from a fusion place it simultaneously performs the same action on all the places of the fusion set. A place may be an input, output or input/output place as required by its utility. If the fusion place exists on a sub page it is called a *port* and if it exists on a super page it is called a *socket*.

The final important note is that individual pages are instanced: unless intentionally designed to use the same page a substitution transition copies a token to an instance of a page (Kristensen, Christensen, & Jensen, 1998). This is easily visualized in a brief example. Consider the HCPN illustrated in figures 1.8, 1.9, and 1.10. The subnet "Group 2" has 2 instances in this HCPN. The first instance's interactions are described on the upper sequence of arcs and the second instance is described on the lower. The HCPN contains 2 substitution transitions labelled "Group 2 1" and "Group 2 2" that both specify transitions to page "Group 2". Each of these transitions has been surrounded by fusion places: one out socket and one in socket. The out sockets are fused with the in ports of the corresponding instance of "Group 2" shown below. Similarly the in sockets are fused with the out ports. Thus the entire net contains 6 distinct tokens, some of which are duplicated within fusion sets. Note that the transition inside "Group 2" does not correspond to the substitution transition, rather that transition can be expanded out to an arbitrarily complex CPN.

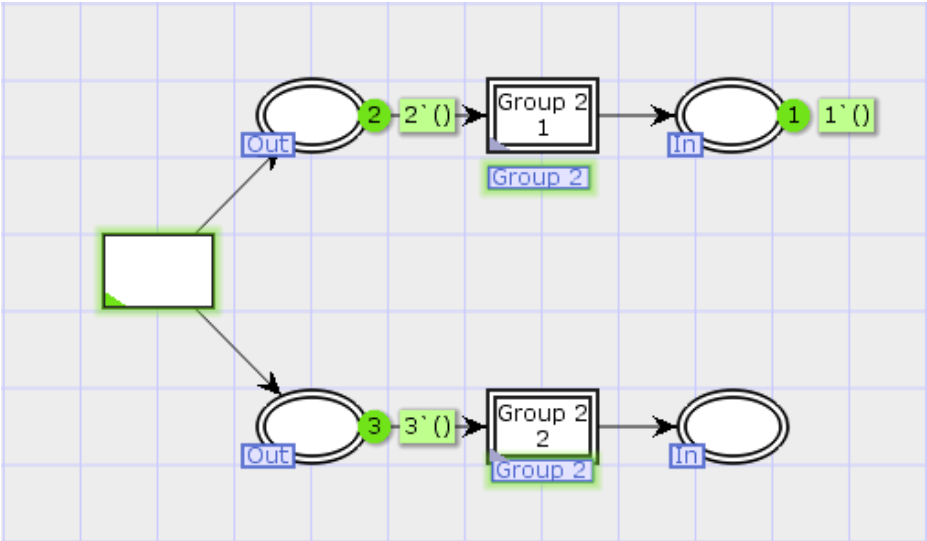

Figure 1.8: The superpage of a simple CPN.

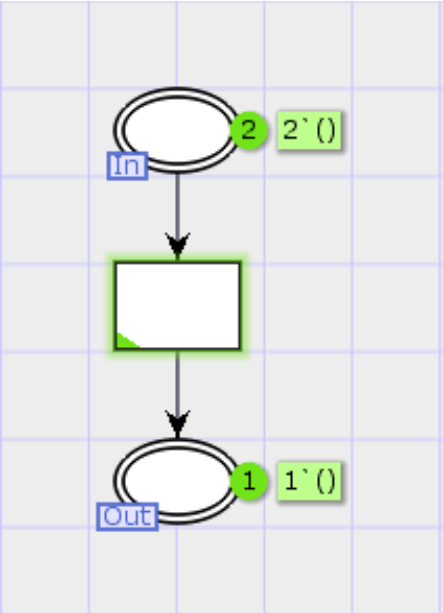

Figure 1.9: Instance 1 of "Group 2".

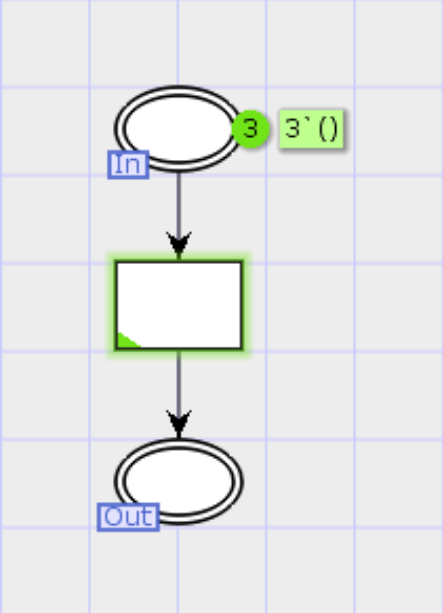

Figure 1.10: Instance 2 of "Group 2".

1.1.8.1 Equivalence to Coloured Petri Nets

Hierarchical Coloured Petri Nets can be shown to be equivalent to CPNs and thus to P/T nets. Thus they provide no new expressive power and instead are a convenience to a net designer.

Informally, for each subnet of a HCPN replace the substitution transition with the instance of the particular page in question in a simplistic way akin to cutting and pasting. Further for each fusion place we can create a single place that each of the places within the fusion set will now point to. This may come at the expense of legibility but it is clear that the meaning will be precisely the same. For a more formal treatment see (Jensen, 1997, p.115)

1.1.9 Timed Coloured Petri Nets

Nothing in the semantics of CP nets allows for introducing a delay on when a token may be available to the net. Once it is created it can immediately be used to enable a transition or participate in a firing sequence. Thus we cannot simulate a delay simply by introducing a series of places and transitions for the token to move through before arriving at the desired place because we can fire enabled transitions in a non-deterministic manner and advance through all of our intermediate states before processing the final, desired transition.

There have been several different theoretical approaches to introducing timing to Petri Nets but we consider the one selected for use in CPN tools(W.M.P. van der Aalst, 2013). This formalism adds a time value, called a *time stamp*, to each token. This time stamp exists in addition to the normal colour information of the token and is denoted by $@n$ or $@r$ where $n \in \mathbb{N}$ and $r \in \mathbb{R}$(Alur & Dill, 1994). Thus if we use the multiset notation previously introduced, we can represent a token by *quantity`value@time stamp*. One useful extension not supported directly by CPN Tools is the idea of Interval Time in which a token has both an availability time and an expiry time(van der Aalst, 1993). This would have been a useful extension in our work as we could more directly model time out conditions.

Time can be defined in other ways and we can denote any arbitrary time set as $\mathbb{T}$. The properties of the time set will have semantic implications within our timed net. Since our model uses a simple view of time, the main properties we require are predecessor and successor functions and a basic set of relational operators such as $\{=, >, <\}$ and a minimal set of arithmetic operators such as $\{+, -\}$. We allow for arithmetic to be performed on the values or for time values so that a notation such as $@n+1$ will mean *add one to the value of the time stamp n* and $@succ(n)$ will mean *one time increment more than the time stamp n*. If we use non-integer time stamps the distinction becomes clearer. We also wish to include the idea of generating random or pseudo-random values from a bounded range. Nets which derive time values from such functions are called Stochastic Timed Coloured Petri nets. A net which contains timed tokens may also contain untimed tokens. These untimed tokens behave immediately, that is without delay, as previously defined. We call such nets Generalized Stochastic Petri nets(W.M.P. van der Aalst, 2013).

Once we determine a time carrier and define $\mathbb{T}$ we need a global clock which will indicate the current time step. This is independent of the wall clock and relates only to the CPN for which is resides in, thus it is often called *model time*. A token is said to be unavailable if the value of its time stamp is less than the global clock and available otherwise. The global clock advances if there are no enabled transitions in a current marking. Care must be exercised when including untimed tokens as such a token in an infinite loop can prevent the global clock from advancing depending on the precise semantics used. For example the clock can be restricted to advance only if there are timed tokens remaining in the system which are not yet available. This restriction preserves the ability to detect deadlock instead of infinitely increasing the time index. A token may be used in the consideration of enabling a transition if it is available. If it is unavailable it is not used in determining whether a transition can

be fired.

1.1.9.1 Computational Expressiveness of Timed CPNs

It is believed that adding time to a CPN is insufficient to generally increase the computational expressiveness of CPNs to that of Turing Machines(van der Aalst, n.d.). It is clear however that some time formalisms do, if they incorporate the idea of urgency(Srba, 2008) defined as the maximum amount of time a transition which is enabled must wait to fire(Akshay, Genest, & Hélouët, 2014). What is not always clear is that this additional expressiveness may be undesirable as it may cause some analytical questions to become undecidable.

Even when the questions are decidable, it is often the case that the model may suffer from state space explosion unless some restricting techniques are applied(Kristensen et al., 1998). That is, CPNs, especially Stochastic CPNS are in general not amenable to analysis though GSCPNs can be converted to Markov Chains but suffers from state space explosion unless a technique like interval bounds are used(W.M.P. van der Aalst, 2013).

Thus simulation becomes the primary method for exploring the behviour of the nets(van der Aalst, 1993).

Chapter 2

Modelling A Steel Cutting Process

2.1 Overview

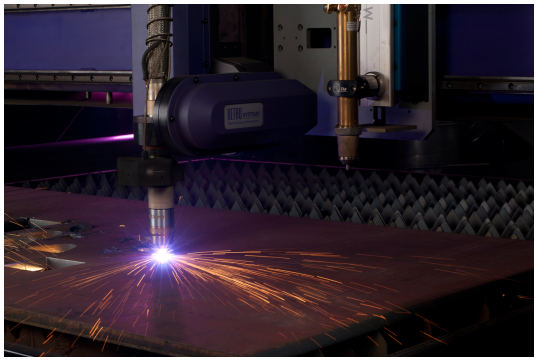

Figure 2.1: A CNC controlled plasma cutting torch.(Wikimedia, 2013)

The goal of a typical steel cutting process is to reduce the input materials, such as steel plate, to individual parts by some physical process. Since there are many physical processes available to accommodate diverse manufacturing requirements it is important that the description of the output part not be inexorably tied to the process that has been selected for its production. These part descriptions must be fulfilled by hardware or software capable of commanding any attached machinery designed for this purpose. Computerized machines of this type are collectively referred to as computerized numerical controls or CNCs[1].

[1]The steel cutting process described is based on my experiences working for Linatrol Systems, Inc. (Stoney Creek, Canada). The model presented represents the Infinity system which is an implementation of the PA8000 CNC software by Power Automation GmbH (Pleidelsheim, Germany).

There is a natural division of knowledge in the steel cutting process: part geometry and process information with a need to be able to communicate about each within the system. The geometry of the part should be considered separately from the information on how to manipulate the machinery to create it so that changes in production techniques, materials or processes will not require re-engineering of designs that have already been accepted as appropriate for production. There are many considerations why this is favourable including the possibility that acceptance tests may be expensive or time and labour intensive.

While the model does not attempt to simulate parsing, verification nor execution of specific G-Code programs it is clear that such an extension could be accomplished within the ML transition functions in the CPN. It is also apparent that if such a simulation were undertaken it would be possible to emulate the motion of the mechanical system, taking into account process knowledge and other interesting operations such as interpolation of movement for smooth mechanical motion. Though some of the functionality of these operations will be highlighted, it will only be to justify the existence of a particular communicating component as this work concentrates almost entirely on describing the communication of knowledge throughout the system rather than describing the knowledge itself.

2.1.1 Compile Cycle Programs

A compile cycle is a nonstandard term used here to refer to the master control loop of the CNC. This control loop must complete in deterministic time to avoid disruption in communication and to maintain responsiveness with the physical hardware. The total time is divided between all of the components which are under the direct control of the CNC, in particular the interpreter, interpolator, and the PLC all of which are

afforded limited time to complete their tasks.

Each of the interpreter and the interpolator can load up to 4 DLLs called compile cycle programs, or CPCs, which are user defined programs written in C. If these CPCs cause the total execution time of the component to swell too much a timing fault will be generated by the system. Thus operations which have non-deterministic timing conditions such as operating system calls or memory allocations are strictly forbidden.

There is one opportunity for the compile cycle programs to spawn a "user mode" child thread which is not constrained by the timing constraints of the control loop, precisely during initialization of the system prior to the establishment of the timing conditions. After this point if the child process becomes unresponsive there is no way to restart it and the parent must be prepared to propagate an error.

2.2 Considerations Around Part Geometry

2.2.1 Part Representation

Part geometry refers to any information that describes the physical measurements of the desired output part. The standard representation of part geometry is a flat text file that is formatted in either ISO-6582 format, commonly referred to as ESSI, or ISO-6983 (also known as RS-274) which are called EIA or "G-Codes". Table 2.1 demonstrates the difference between the formats.

EIA is probably the most commonly used format owing to its readability, which was of paramount importance prior to computer generated part files. Both of these code formats are still in use however, due to libraries of legacy part definitions. A CNC must have an interpreter capable of verifying the syntactic description of the part file in either format and converting it into instructions that can be passed to

hardware controllers to command motors, torches and the like.

Counter-clockwise cut from bottom-left:100mm rectangle with 20mm radius corners	
	'(ESSI CODES)
	21
	38
	39+0
'(EIA CODES)	40+0.0
%	8
N1 P1234	38
N2 G46 T1	5
N3 G97	++
N4 X20 Y0	6
N5 M04	20.0 0.0
N6 X80	9
N7 G03 X100 Y20 I0 J20	60.0 0.0
N8 Y80	20.020.00.020.0+
N9 G03 X80 Y100 I-20	0.0 60.0
N10 X20	-20.020.0-20.00.0+
N11 G03 X0 Y80 J-20	-60.0 0.0
N12 Y20	-20.0-20.00.0-20.0+
N13 G03 X20 Y0 I20	0.0 -60.0
N14 G98	20.0-20.020.00.0+
N15 M30	38
	5
	++
	6
	0

Table 2.1: EIA and ESSI Code Sample(Livingstone, 2000)

The geometry of the part is expressed as a path that a tool must pass over so that the resulting separation creates the requested part. Since these paths are typically travelled by a combination of linear motors, or rotary motors exerting force along a linear actuator such as a rack and pinion, and are only capable of producing straight line motion the commands in the part description program are typically vectors or arcs. Though arcs are not technically possible with linear travel, they are well emulated by commanding continuous motion in multiple axis simultaneously and are

therefore directly supported in the text representation.

The part files can also contain "M-Codes" which carry commands such as "activate the cutting tool" or "adjust a tool wear offset". They are usually considered to have immediate influence on the system and can adjust the physical path of the tool. These arcs, vectors and miscellaneous commands are translated into motions on particular axes by the CNC Interpreter discussed in section 2.2.2.

EIA and ESSI codes are well defined but manufacturers have extended the standards in proprietary ways since its formalization in 1980. As a simple example one manufacturer uses EIA code G21 for selecting metric units where another uses G71[2]. Thus it is often required to introduce a translation step available prior to the interpreter's involvement. In the model this facility is provided by a user defined compile cycle program, explained in detail in 2.2.3.

2.2.2 CNC Interpreter

The CNC Interpreter is responsible for verifying the syntax of a part file is correct, expanding macros, performing calculations contained in preprocessor blocks, converting part co-ordinates in the file into machine co-ordinates and generating a series of instructions correctly formatted to the internal language of the system that facilitates the operation of the physical process. The interpreter will normally support a variety of standards such as ESSI and EIA and perhaps others.

The interpreter typically does not analyze the part file as a whole, but rather only receives a buffered subset of the entire part. This is done to accommodate injection of motion corrections or other miscellaneous codes during run time. Thus the interpreter must remain responsive and active during the entire cutting process and must operate within defined timing parameters. To support this requirement, there is no facility

[2]G21 is used by Hypertherm Inc. while G21 is used by Linatrol Systems, Inc.

for user defined calculations.

The interpreter will have knowledge of the machine configuration and may inject simple process information into the code, often implemented by including one of a set of preprogrammed subprograms. In Table 2.1, *M04* is used to command the machine to *activate cutting process*. If the interpreter had no knowledge of the machine configuration it would not know how to correctly activate the process. Here a subprogram may inject process information to the code, for example instructions on how long to wait before deciding a torch has failed to ignite.

The output of the interpreter may be routed either to the kernel or directly to the CNC Interpolator depending on the desired operation of the machine.

In the CPN model the interpreter is a subpage of the CNC subpage. As with most pages in the model a fused place, here marked CNC, feeds tokens from the superpage to a substitution transition that has a guard function associated with it. The guard function will check the destination of the packet and compares it against a constant which represents the subpage. Overall the interpreter is a simple net except that it must enforce a FIFO processing of the incoming blocks as out of order execution of the code blocks would certainly and obviously corrupt the desired geometry. Since the block processing has not been modelled it is possible to exclude this detail even though buffering might produce an interesting pattern. Thus a proof of the concept of incorporating a FIFO structure within a CPN is discussed in section 3.3. Even though no simulation or emulation of code translation is taking place, the action is represented by the input of a code block from the CNC and the output of a different code block to the interpolator or the kernel. This is represented by incoming messages cDATA, cpDATA, pDATA and the outgoing message rDATA. The incoming packet is processed by *irTransition* and then passed to a transition that targets it either to the CNC or the CNCIP (interpolator).

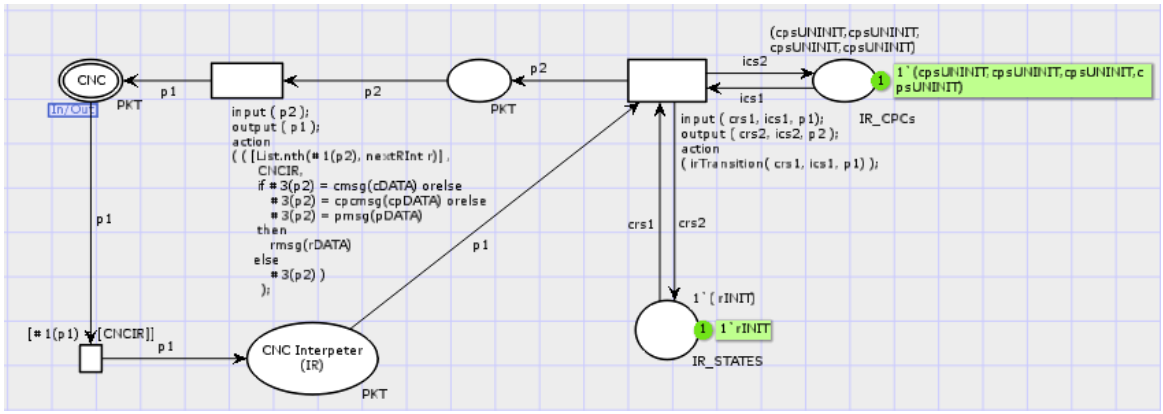

Figure 2.2: The CNC Interpreter Initial Marking

The Interpreter may have up to 4 compile cycle programs associated with it which are functions executed each time the interpreter runs.

Figure 2.3 describes the sequence of state transitions within the interpreter defined in listing 5.2. Once the interpreter starts execution it waits for each compile cycle to initialize individually with the state that the compile cycle reports being noted. Once the compile cycles are initialized the interpreter becomes ready. From a ready state the interpreter can move to a fault state in the event of a system error which halts all further processing or it processes incoming data from either the CNC, a CPC, or the PLC. For the precise definition of the different modelled states see appendix 5.2.

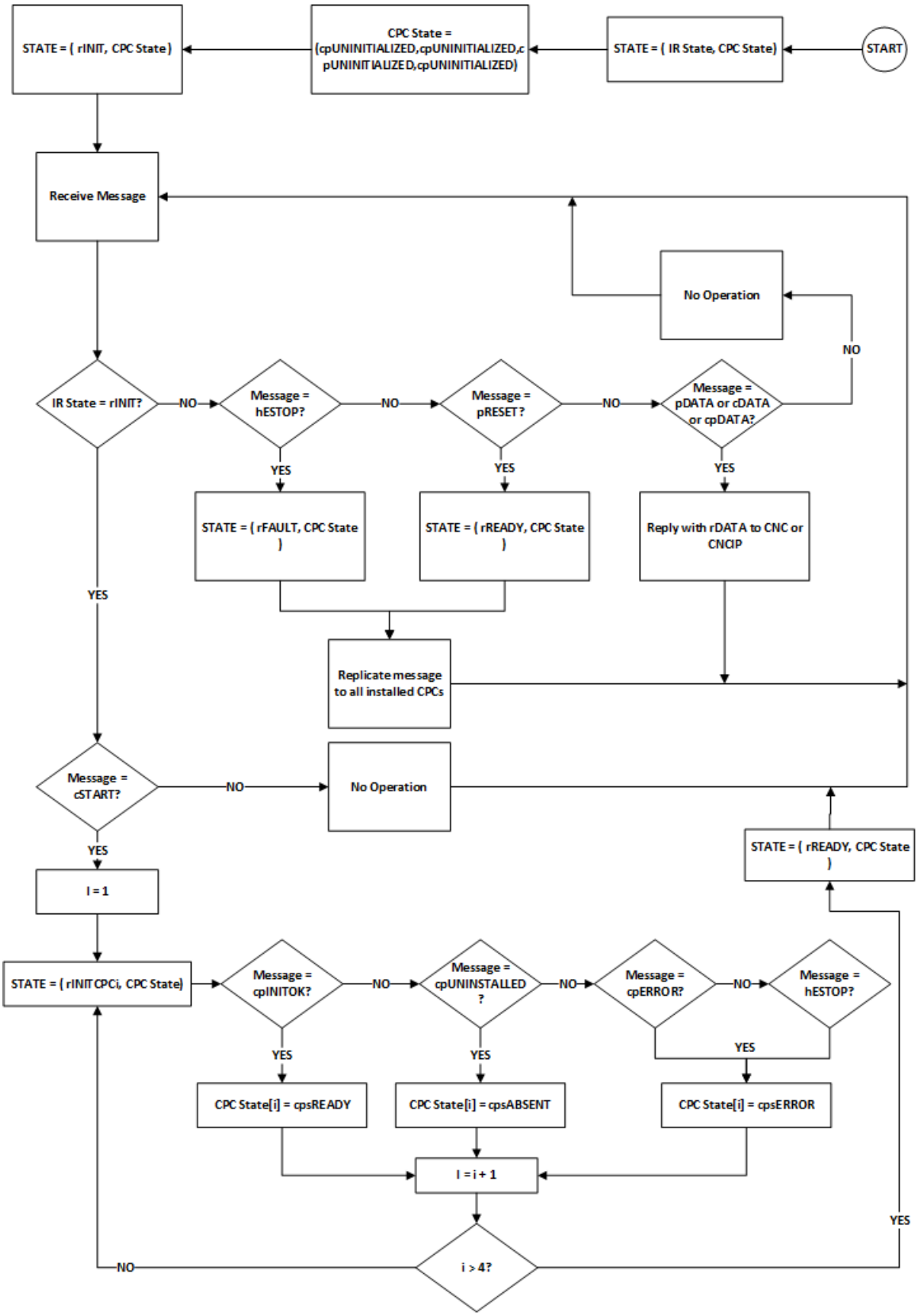

Figure 2.3: State Transitions Within the Interpreter

2.2.3 Load Time Translation Tool

The load time translation tool is a module loaded by the interpreter as a compile cycle program. This tool consists of 2 connected components: a real time control loop and a relaxed time program running in user mode. The components communicate through the use of shared memory and thus can maintain state information about each other. The real time portion of the translation tool must also communicate its liveliness to the PLC by way of a "heartbeat" signal. Since the control loop may not busy wait or block the execution of the interpreter, it may execute many times while waiting for the user mode component to complete the translation. Thus it is critical to model a timeout condition in this page. The proof of concept in section 3.4 demonstrates how a timeout can be incorporated into this net.

As the system is initialized the compile cycle is loaded in Windows User Mode, which allows the creation of a user mode thread which is not time sensitive. Once the CNC initialization transitions the system to Windows Kernel Mode this thread creation would generate a fault, thus the user mode program cannot be reset if the thread crashes in the background. It becomes necessary to provide a "still-alive" signal to the PLC to demonstrate responsiveness. This is modelled by sending a heartbeat packet to the PLC every time a packet is processed by the CPC. If the PLC does not receive a heartbeat packet within an arbitrary threshold it should generate a system fault.

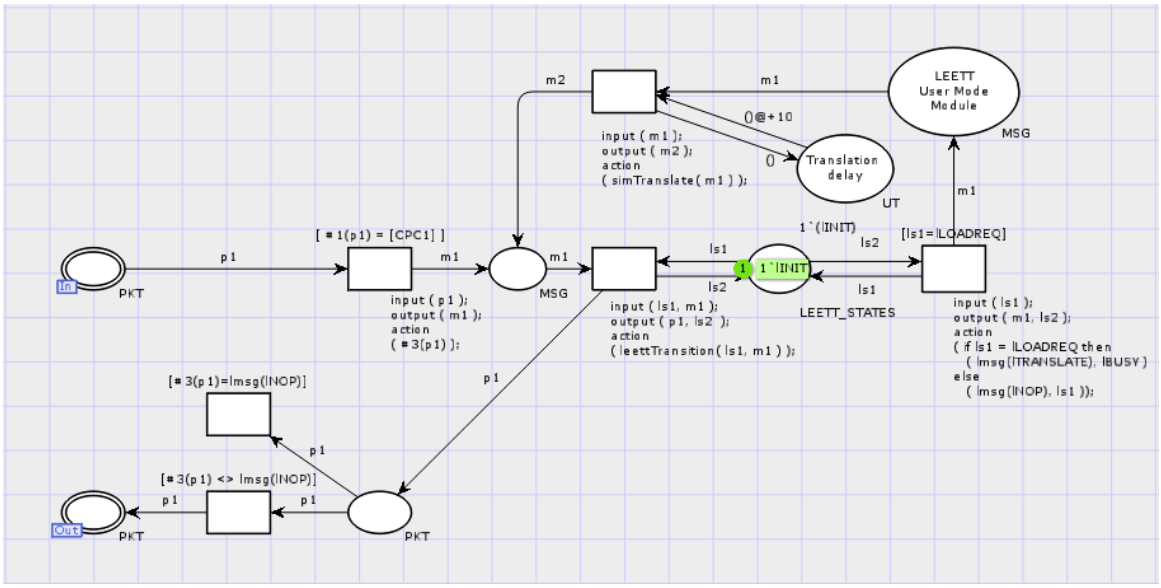

Figure 2.4: Translation Tool Cycle Intial Marking

The user mode and kernel mode cycles are represented on the same page by dividing them logically in the net. They can respond to communication separately through two transition functions representing the internal and external communication that is possible. Only one translation at a time is allowed and in the real system it is not possible to cancel the translation. A large geometry may only take a few seconds to translate so the PLC will simply queue commands until the translation is available.

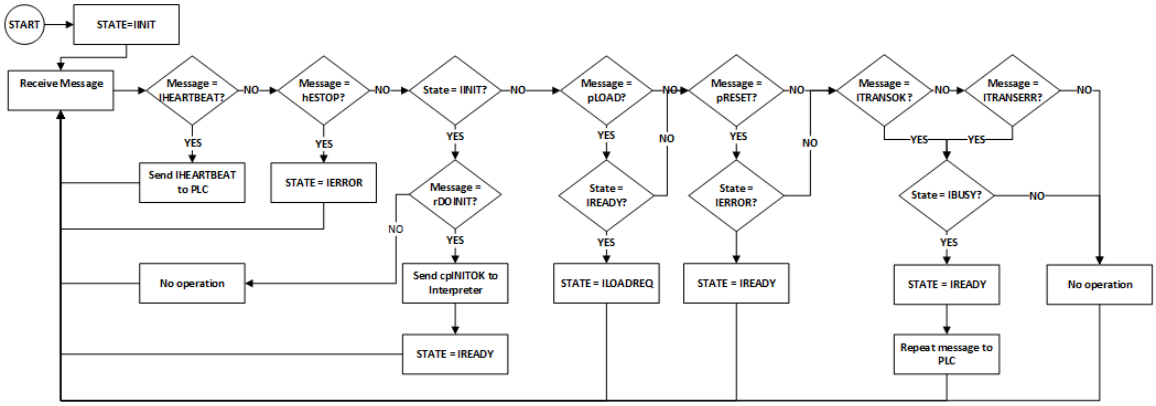

Figure 2.5: State Transitions Within the Translation Tool

Figure 2.5 represents the state transitions that occur between the compile cycle

and the external system. Once the compile cycle achieves a state of *lLOADREQ* it requires the internal facing transitions to progress. On a state of *lLOADREQ* the internal transition changes the state to *lBUSY*. This is done to avoid multiple requests to the translation thread preventing the code from attempting to start a new translation before the previous one has completed. An arbitrary time delay to represent the translation time is included by way a of a timed unit token. Using this technique a timed message colour is not required, though if this pattern emerged throughout the net it would be reasonable to create such a colour.

As with the real system if the user mode segment becomes unresponsive the entire system will become deadlocked. It is possible for the real system to continue to exhibit the heartbeat signal even when the user mode system is deadlocked, so although that has not been specifically modelled here, it is easily introduced. The function simTranslate allows for this translation CPN to produce the successful translation or the non-fatal translation error state changes as random outputs and could easily accommodate generating a fatal error by failing to ever change the state from *lBUSY*. The ML code representing the transitions, as well as the support function simTranslate are listed in appendix 5.3.5.

2.3 Considerations Around Process Information

Process information refers to knowledge of the machinery or the physical process that is to be used to create the output part. The process information can also be stored in the "G-Code" file, normally interspersed among the movement codes. This information is required due to the diversity in technology that can be employed to achieve the part creation: milling, drilling or router heads, oxyacetylene flame or plasma torches, water jet or laser cutting heads and laser sintering additive manufacturing to name a

few.

The CPN model does not rely on any particular configuration of equipment or process information, especially since we are not attempting to simulate any "G-Code" processing. One common process element that is represented here, however, is the CNC Interpolator in section 2.3.4. This is such a common quality component that the interaction with the rest of the system warrants inclusion.

Other components that rely on or generate process information are represented within the CPN such as the configuration tool, the gas controller though the communication aspects of these components is underscored and will be discussed in section 2.4.

There are other processes that while more specialized, introduce new communication requirements on the steel cutting system. These particular processes, height control, bevelled cutting, and pipe cutting, are not universal but are still relatively common. They require real time system responses such as axial transformations which we can represent in the model simply as packet generation rather than performing specific computations.

2.3.1 Height Control

One of the assumptions made in generating a part file is that the surface of the plate is uniform and level. Without any exotic extension to the steel cutting process it is not hard to imagine that this assumption is easily violated. Surface irregularities or a misalignment of a support can cause quality problems or worse, can cause the cutting tool to inappropriately impact the plate surface. This problem can also arise due to thermal rippling of a material, so it is not obvious that a higher grade material can protect against it. While this seems a very simple problem, it becomes much more

complex when the height controller moves in the Z axis but the cut is not restricted to the X-Y plane. In such cases more substantial calculations may be required to produce the correct movement.

To counteract the violation of the uniformity assumption a mechanical height controller is often introduced. There are several ways that the cutting tool distance to plate can be measured such as arc voltage measurement in plasma cutting or laser range finding. For our purposes it does not matter which is implemented. The CPN will only consider that the height controller may inject movements and thus generates communication.

2.3.2 Pipe Cutting

The next assumption which can be violated is that the surface we are cutting our part from is flat. When parts are cut from or into a pipe it is often the tool which is held fixed in at least one axis while the pipe rotates beneath it. If the pipe is square shaped, or otherwise has portions where the thickness varies, adustments may be required in the cutting speed. The model recognizes these adjustments as a real time axial transformation and a correction factor. Though there is no specific pipe cutting module within the CPN, there is a compile cycle discussed in section 2.4.4 which transforms linear co-ordinates,

Figure 2.6: An example of pipe cutting.(Wikimedia, 2007)

such as line drawn along the X-axis, into a rotational movement of the pipe.

2.3.3 Bevel Cutting

The final assumption violation that the CPN model will account for is the idea that the cutting head is fixed at 90°, or some other arbitrary angle, to the cutting surface. This problem arises in post production, when parts are prepared for welding and joining by creating a beveled surface that will butt against another part, forming a small gap which will be filled by molten metal. When performed manually bevelling can be labour intensive and can suffer from inconsistent results. This process brings together difficulties with height control, thickness variations and many other required corrections.

To create the geometry it is required to account for the motion of the head about its own axis as well as the relationship to the X-Y plane of the plate. Many problems of acceleration come into play during cornering where maintaining the angle around the corner requires a substantial repositioning of the head. Typically the custom process information may recognize an approaching corner and inject many statements to smoothly transition. This idea is akin to manual interpolation of the rotation of the head with the acceleration being used as the limiting factor. This type of axial transformation is also discussed in section 2.4.4.

2.3.4 CNC Interpolator

The CNC Interpolator is responsible for smoothing the mechanical motion of the machine operation as it travels along the geometry. There are a few different specific concerns that the Interpolator attempts to correct. The primary input stream for the Interpolator is the Interpreter, though it is possible to inject sequences manually.

The most basic need for interpolation arises from the desire for simultaneous movement in multiple axes. This arises in many motions but can be simply explained by a straight, diagonal line. If this line is to be traversed by 2 linear motors the motion needs to be carefully co-ordinated. This can be accomplished by altering the feed rate (speed) of one of the axes so that both motors stop at the end point simultaneously providing the diagonal path required. If the motors cannot be independently commanded in such a manner then the path has to be broken in a series of alternating steps, each step usually at the machine resolution, which approximates the diagonal straight line. Indeed machine resolutions are normally so small that for most purposes the interpolated nature of the motion is undetectable.

Another situation where interpolation creates a smoothed motion is stitching together consecutive code blocks. In order to prevent stuttering or stop/start motion while the next block is read the blocks are buffered to provide a continuous stream of motion commands. Moving from one block to the next it is possible to have a change in speed, direction or both. Since it is not possible to change speed or direction instantly due to the intertia of the physical system, the request must be processed over some short time interval. During this time interval the machine may still be in motion and then the change is spread over a short distance. These short paths must be created by the interpolator within the tolerances of the connected devices. Corners and arcs provide the most extreme examples of typical interpolated motion.

In the CPN the Interpolator should be represented as a FIFO net if any real modelling of the "G-Code" is to be accomplished for the same reason as the Interpreter: out of order execution of movement commands will results in a corrupted geometry. The output from the Interpolator is passed to the CNC Kernel for dispatch to the physical devices.

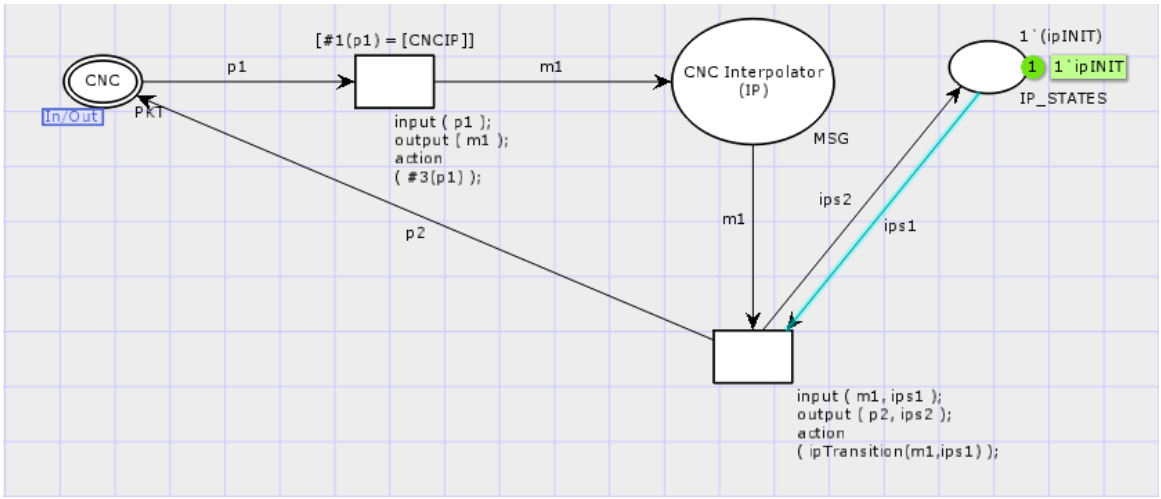

Figure 2.7: The CNC Interpolator Initial Marking

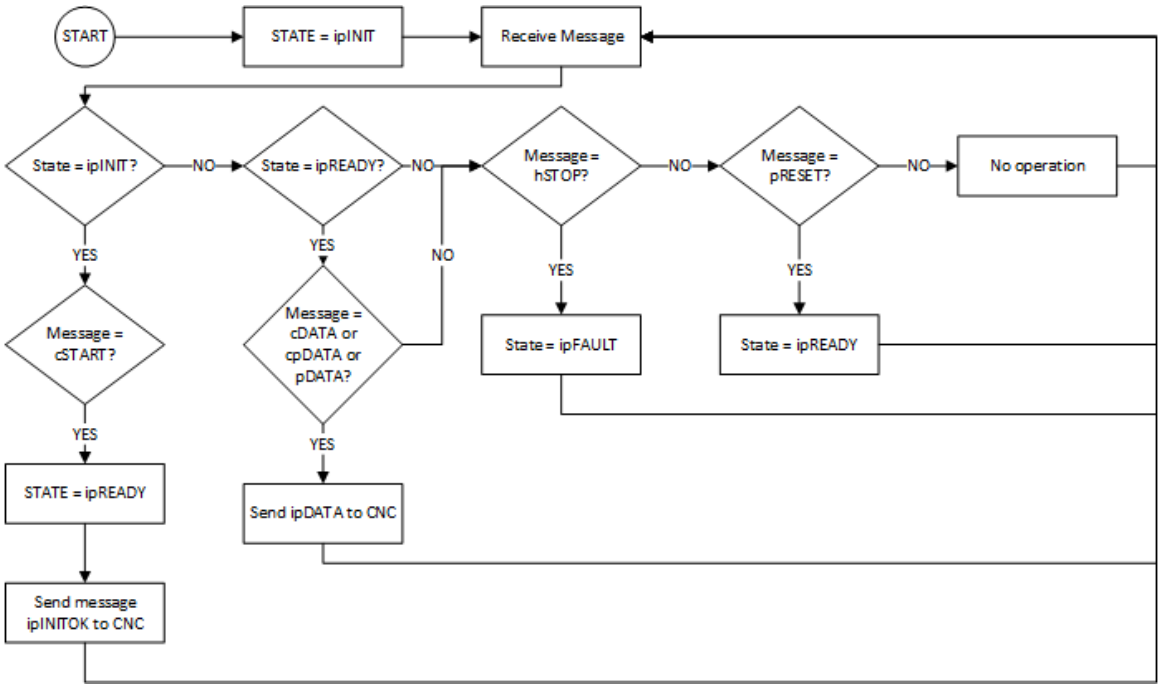

Figure 2.8: The CNC Interpolator State Transitions

2.3.5 Programmable Logic Controller

What differentiates the quality of one CNC from another is often found in the attention paid to process information. A substantial amount of this knowledge becomes

embedded in the custom software running in the Programmable Logic Controller, or PLC. The CNC Kernel itself is capable of commanding the hardware but it does not generally encapsulate the specific engineering knowledge that aids in high quality and efficient part creation. Although there are several types of languages available to program PLCs with such as relay ladder logic which resembles the historical electrical circuit control or Pascal-like programming languages, the CPN model does not emulate the parsing of any programs. Instead the effects of the state transitions are manipulated to simulate input from the HMI or simulating responses to communication from the system. The languages can all access a library of CNC functions provided by the manufacturer but due to strict timing constraints are not allowed access to operating system functions or memory allocation.

The PLC is capable of re-routing HMI requests like "Load File" to the compile cycle before presenting the final translation to the CNC for interpretation. Similarly most input and output which communicates through the communication adaptor compile cycle described in section 2.4.4.2 can be intercepted by the PLC, reformatted and acted upon before the CNC needs to be involved in the final processing if at all.

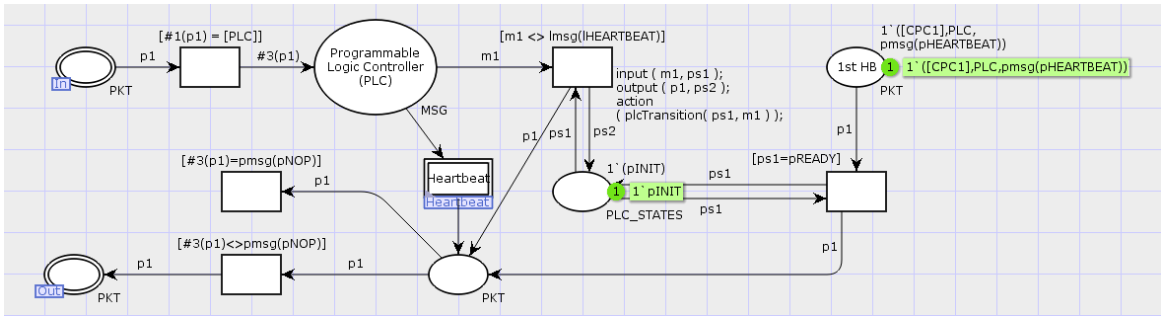

Figure 2.9: The PLC Initial Marking

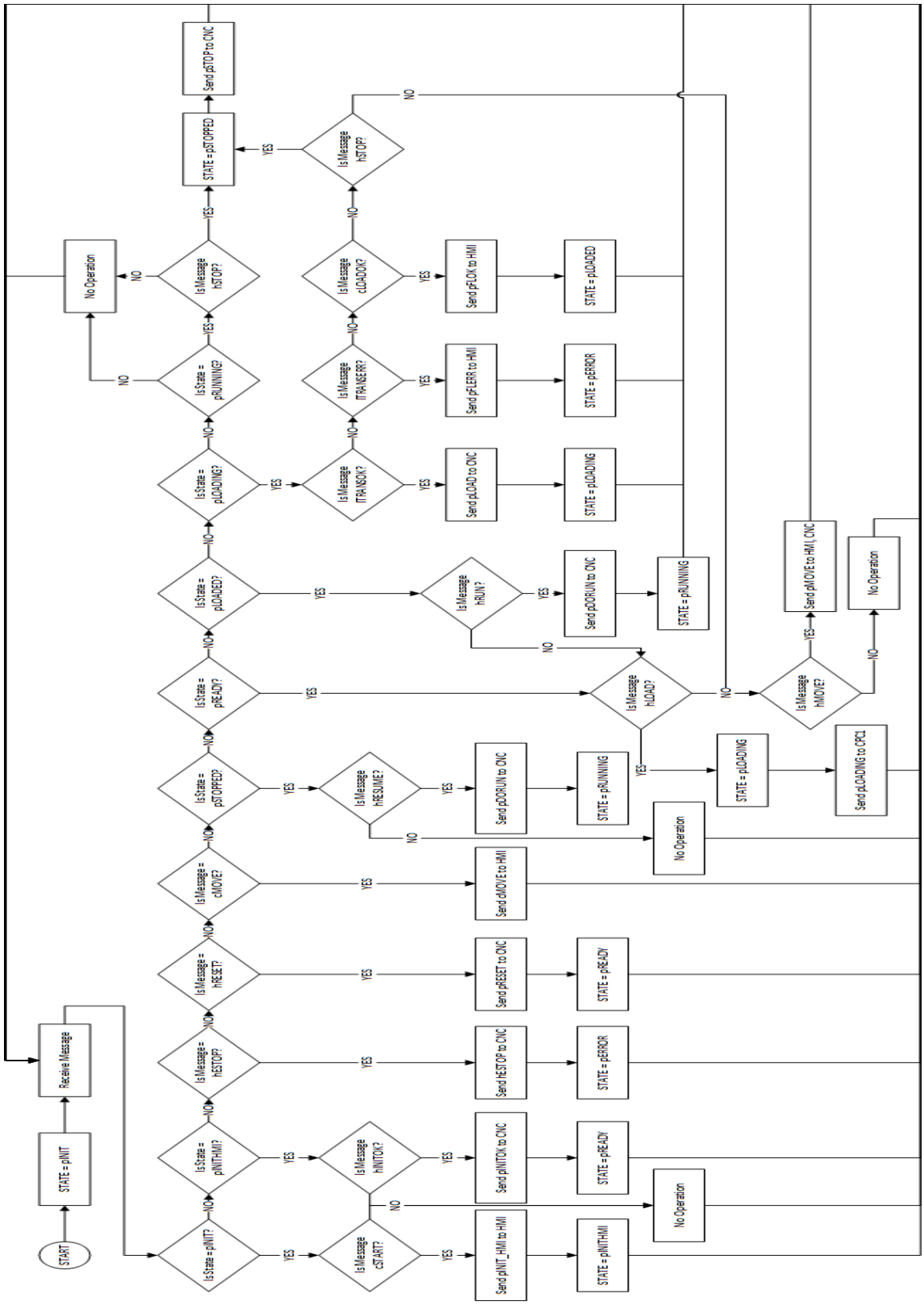

Figure 2.10: The PLC State Transitions

2.4 Considerations Around Communications

The system is composed of heterogeneous systems; PC based controllers, integrated systems and real time sensors all frequently communicate with each other. Communication in the physical system under consideration is carried on a TCP/IP interconnect network. The TCP/IP interconnect is represented as a packet passing state which connects all of the components which are modelled as pages. A more complete model could expand the single state into a communication subpage that addresses topics such as lossy communication which is common in some cutting applications due to high levels of radio frequency interference generated during plasma cutting, however modelling the TCP/IP protocol was not the focus of this paper. Indeed this has been accomplished in (Figueiredo & Kristensen, 1999). Thus lossy communication, retransmission and error correction on the interconnect are assumed.

One behaviour that seemed important to model was to allow the TCP/IP interconnect place to produce broadcast messages; a message such as an emergency stop should be transmitted to all connected subsystems as rapidly as possible. A detailed explanation of the broadcast CPN is presented in section 3.2. Another interesting characteristic is that the system must continue to be live while waiting for the physical movements to complete; a code to move 3 inches can be processed instantaneously but the machinery requires some time to complete the motion. This is another motivation for timing and buffering ideas that are represented within the CPN.

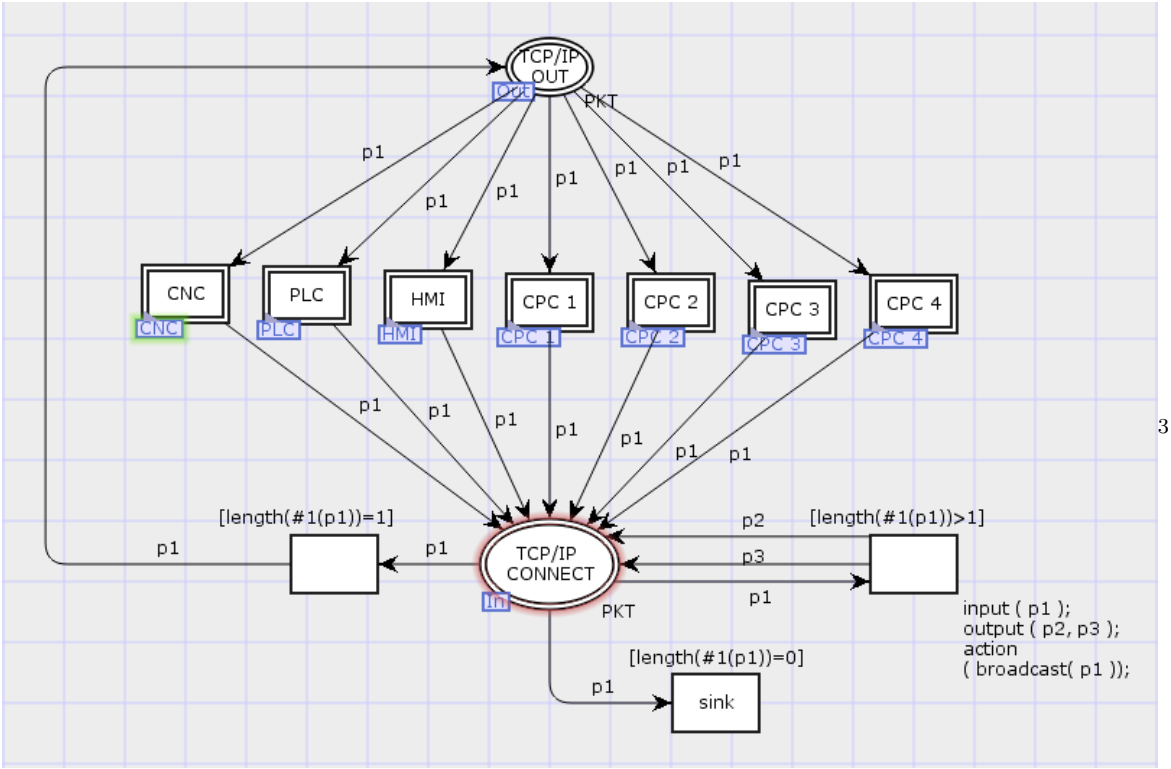

Figure 2.11: The top level net.

In the top level net there is 1 place with 3 connected transitions representing the TCP/IP interconnect and 7 transitions representing the CNC, PLC, HMI and 4 user defined compile cycle programs. Each transition is an I/O port to a page defining the specified component. Routing is accomplished by the guard functions of the connected transitions checking the first parameter of the PKT structure. Not that if not for the guard functions, a copy of the packet arriving in *TCP/IP OUT* would be replicated to all the connected transitions which is not a desirable characteristic.

2.4.1 Representation of Packets and Messages

Having earlier decided not to model the TCP/IP protocol all that remains is to represent the movement of TCP/IP packets in the model. Using the terminology

[3]The light pink aura on the TCP/IP place denotes that it is a fusion place.

packet to mean directed, encapsulated message, the packets are represented as the product set $PKT = COMPONENT \times COMPONENT \times MSG$. The meaning is described by the tuple $(destination, origin, message)$. This definition of packet allows for one uniform colour interface across all pages which is a satisfactory representation of the real system. The definitions of components and messages are listed in appendix 5.1.

The essential characteristic that the model captures is that there are varied messages produced by each subsystem which are perhaps only understandable by particular recipients. Neither the PLC nor the CNC may have any idea how to understand these messages and thus they should not process them, only route them. The TCP/IP interconnect place serves this purpose.

Creating a generalized message type that will allow for strict data typing required the message colour to be defined as a union of all the potential message types from each subsystem. Thus the colourset MSG is defined as:

MSG = union cmsg:CNC_MSGS + pmsg:PLC_MSGS + hmsg:HMI_MSGS + lmsg:LEETT_MSGS + rmsg:IR_MSGS + ipmsg:IP_MSGS

2.4.2 Real Time CNC Kernel

The real time kernel is the portion of the system which actually interacts with the signals to and from the hardware including movement, safety critical systems and other time sensitive operations. Calls to non-kernel mode operating system functions are not allowed; functions such as memory allocation or other library calls that have non-deterministic timing conditions could potentially make the system unstable and are thus restricted.

Since the "G-Code" was not modelled in this system no effort has been made

to track the states regarding the motors or tools though both of these seem to be interesting extensions to this work which lead down the path to operation simulation.

The CPN models the CNC Kernel as a message router that contains the interpolator, the interpreter and the motor controller, all of which may operate independently of one another. The compile cycle programs should exist as subpages to either the Interpreter or the Interpolator pages but since this is a non-standard feature they are modelled on the top page, with the communication being routed more directly as the primary means of communication is through TCP/IP. This simplification simply saves some packet readdressing and seems reasonable. It would also be reasonable to tighten the dependency to better emulate the true design of the CNC system if we were so inclined however, since packets may be processed without any time consumption this is perhaps trivial.

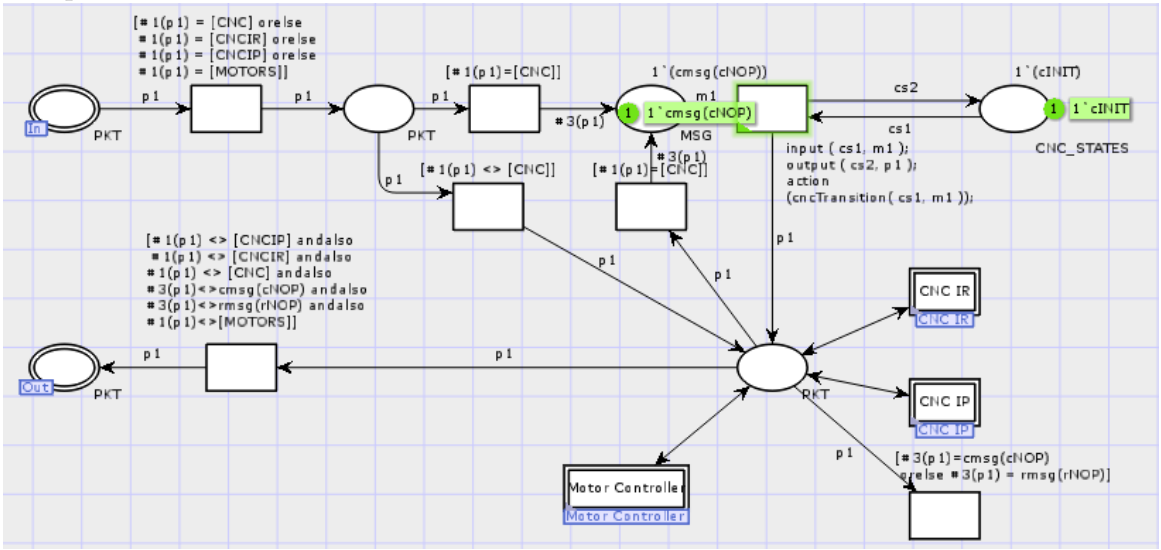

Figure 2.12: The CNC Kernel Initial Marking

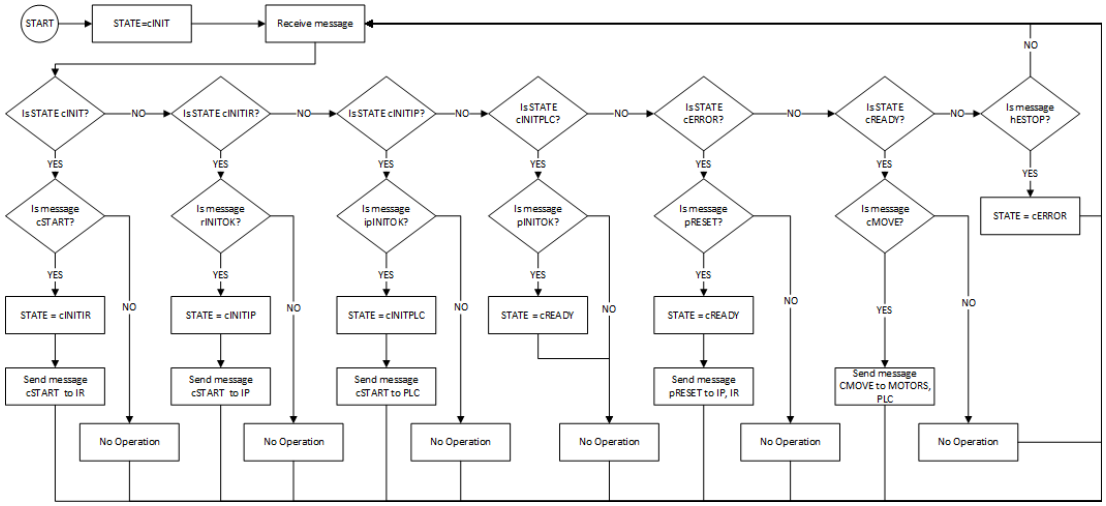

Figure 2.13: State Transitions Within the CNC Kernel

Although there is a defined sequence of initialization steps it is important to realize that all of these initializations occur within a single time step as the state token is not timed and any packets associated with it will be processed without delay. This is satisfactory because we are not interested in making any substantial analysis of the startup conditions. If we wished to better emulate the CNC initializing all its components in parallel we could introduce intermediary states to generate multiple packets, or use a broadcast initialize to the CNC components. One consideration would be to model the race condition that exists on the start up of a compile cycle and its attempt to establish a user mode thread before the CNC system restricts operations to strict timing in Kernel mode. There is no published information on this condition so it has been omitted from the model.

2.4.3 Human-Machine Interface

The Human-Machine Interface, or HMI, represents the primary method by which the machine operator is able to interact with the system. The operator may make requests of the system to perform particular actions or may manipulate some of the system's

environment variables. Any requests must be handled in a manner that is consistent with safety policies. The system must relay to the operator pertinent data as to the state of the system. Finally the HMI must remain responsive to send emergency stop requests.

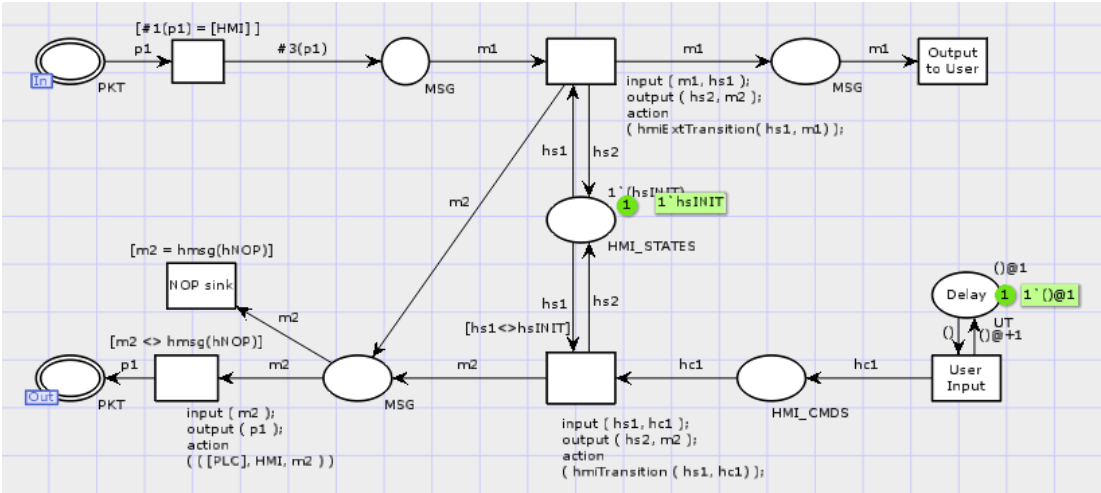

Figure 2.14: The HMI Initial Marking

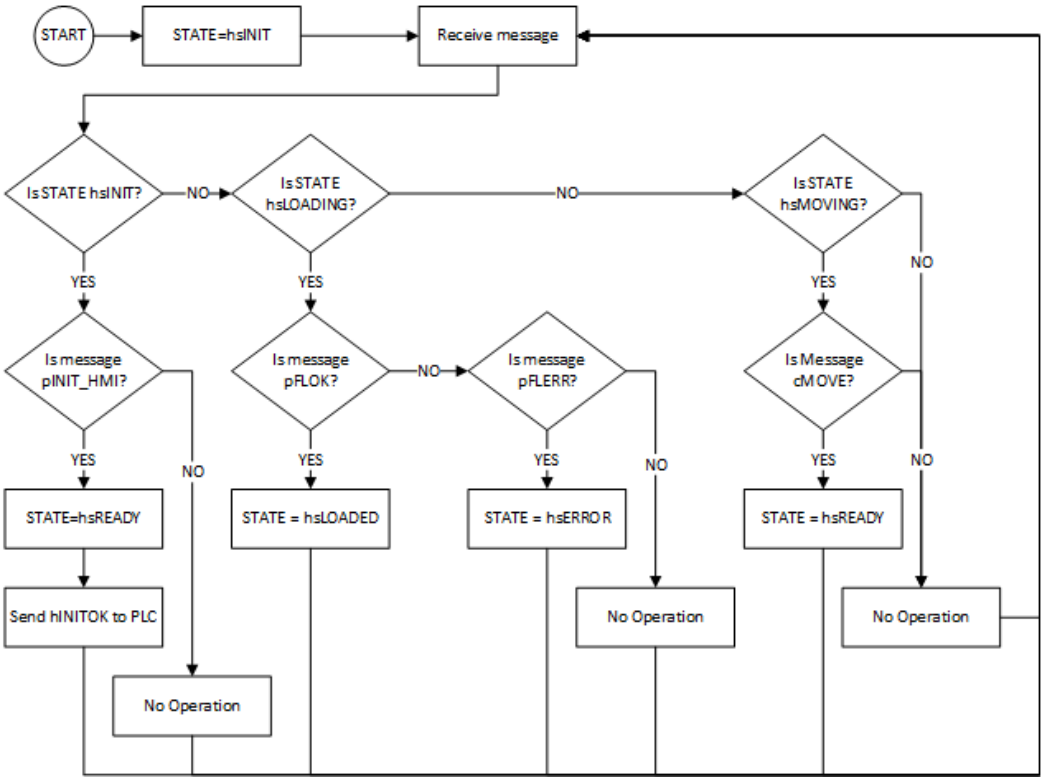

Figure 2.15: State Transitions Within the HMI (External)

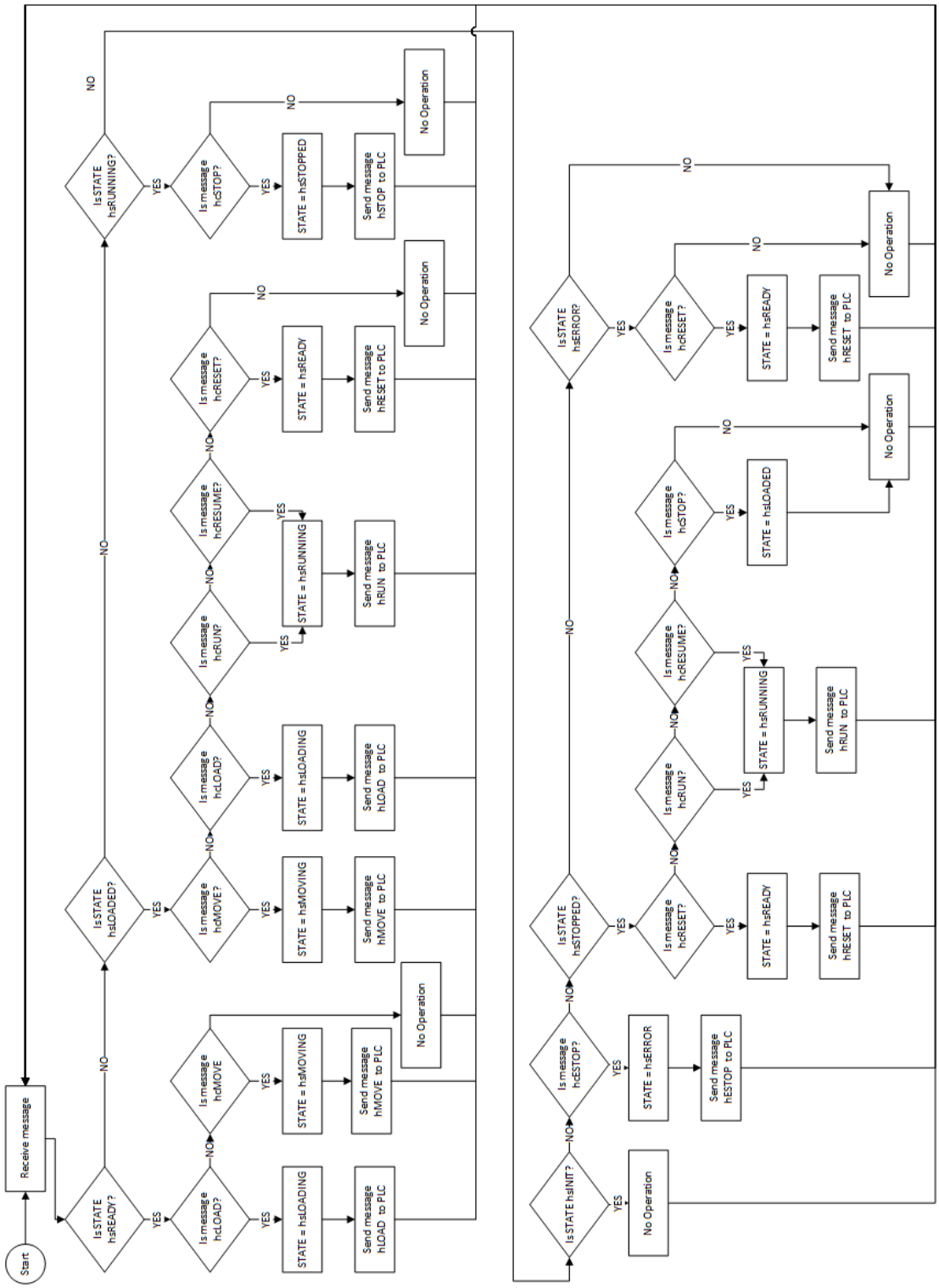

Figure 2.16: State Transitions Within the HMI (Internal)

Many common functions that are available on a CNC's HMI have been omitted. For example while machine homing is a standard operation for our purpose it can be generalized as part of the CONFIGURE function. State transitions are either triggered internally in response to the HMI input or externally in response to some system action.

In this CPN timed unit tokens have been used as a delay on the stream of randomly generated user input messages. The reason for this delay is that if this input was left unbounded it would become impossible to advance the global clock. Thus a delay of 1 time unit is introduced so that the *User Input* transition cannot fire infinitely often.

2.4.4 Compile Cycle Programs

2.4.4.1 Real Time Transformation Tool

This compile cycle transforms the G-Code description of the part into movements that are appropriate for the physical configuration of the cutting tool for geometries that are not standard. One such non standard geometry comes out of attempting to add bevel cutting on to a 2D part. Since the part is normally described by movements in the X-Y plane with an assumption of uniform thickness it is reasonable to consider the bevel as being described simply as an angle to the plane. In other words, once the machine has reached a specified X-Y position the machine rotates the cutting head by some mechanical means to the desired angle and continues cutting along its X-Y path at the new angle. This interpretation implies that the en-

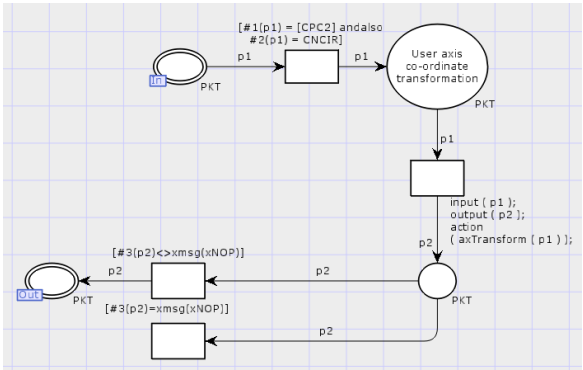

Figure 2.17: Transfom Initial Marking

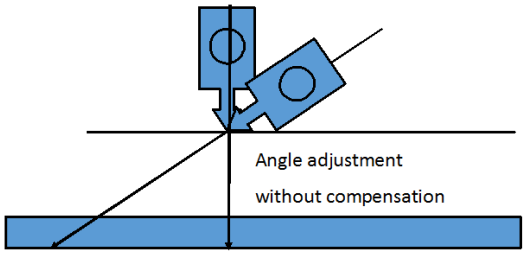

Figure 2.18: Cutting angle adjustment without compensation.

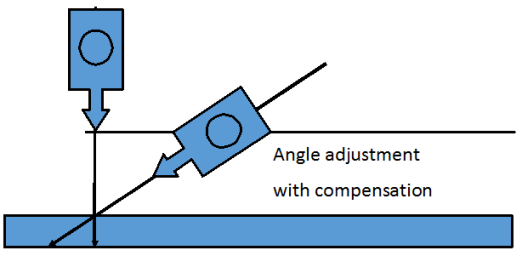

Figure 2.19: Cutting angle adjustment with compensation.

try cutting point should be retained at the specified X-Y co-ordinate as illustrated in figure 2.19 and not as in figure 2.18. It becomes clear that to satisfy this requirement some mechanical adjustment will be required. Another possible source of movement comes from the mechanism chosen to rotate the cutting head as the pivot point of the cutting tool is almost never located at the tip of the tool but normally lies within the body of the tool itself. This is represented in the aforementioned illustrations by a circle. Since the G-Code represents the geometry of the part and the CNC does not automatically compensate these non-standard configurations a manual kinematic transformation must be included as a processing step. As with other G-Code transformations within the model, this part of the model should maintain a FIFO structure. The transformation here is represented as the generation of a new output packet for each input packet. The output of this transformation can then be passed to the CNC kernel or to the interpolator depending on the requirements of the machine.

2.4.4.2 Communication Adaptor

The communication adaptor connects the system to components that do not communicate by TCP/IP. Samples of such systems include a Gas Control Console or a Height Controller which operate by a shielded serial connection in real time during the operation of the cutting process or a machine configuration tool which operates with

the CNC during system initialization to read values before the kernel mode transition and then is also able to make changes to the system while the cutting tool is off.

The gas control console has the ability to modulate any of the gas flow variables and such functionality can be useful if a system requires an additional method of varying the cutting power. The number of setup and configuration details in such a system is very daunting. The creation of a configuration tool which is able to expose some of the interdependencies between variables and simplify the setup parameters by interrogating the user is of inestimable value. No such standard tool exists, and can not be created within the framework provided by the PLC or other components due to the timing of the variables being available to being written to; machine parameters cannot be rewritten once they are loaded into the CNC and must be written to during initialization.

Thought it is atypical these types of systems can be developed by a capable OEM and provided to their customers by building a TCP/IP adaptor to interrogate the end user software and interact with the CNC. In the particular software modelled here the CNC's TCP/IP end point lies in a compile cycle program and is not otherwise intentionally exposed.

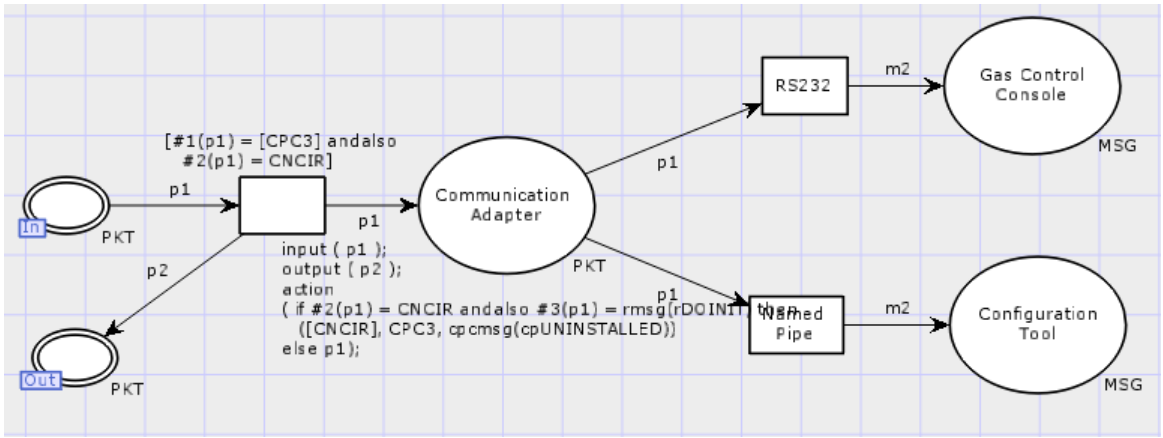

Figure 2.20: Communications Adaptor Initial Marking

Although this particular model does not generate any feedback to the rest of the system it is retained to demonstrate how adaptors between physical connections or between logical protocols could be introduced into the model.

Chapter 3

Observations

From the outset it was clear that there were aspects of the system which where it was not clear how to translate the behaviours into the CPN model. As the CPN grew it became apparent that there was more value in inspecting these elements than in attempting to create an industrial grade emulation of the cutting system. This is the reason that the current model is somewhat incomplete feeling in some regards.

3.1 Design

The initial design of the CPN mimicked the design of the hardware and software components of the part cutting machine. The major components were modelled in the net structure and the behaviour of the system was captured in the ML code attached to transition inscriptions. This came about naturally, beginning with representing the component states and interprocess messages as colours and changes to states being encoded in the inscriptions. The final model readily demonstrates this early decision. Following the design choices of the machine builders allowed an understanding of what the components were responsible for and how they communicated and interacted with

the entire system.

Many sub-systems followed common structural patterns but some care was applied to resist the temptation to merge the systems in an effort to reduce the net structure of the model. Keeping the components separated retained the visual understanding of the net and did not muddle the understanding in arc inscriptions, since that is where the complexity would have most likely been transferred. Kurt Jensen notes in his introduction to CPNs that "it is [usually] a good idea to try to distribute the information between [...] the net structure, the declarations and the net inscriptions"(Jensen, 1997, p.19). Following this advice little effort was made to reduce the net structure to anything resembling a minimal state. Ultimately as the model matured this proved to be a reasonable approach as minor early differences became more significant between components.

Initially, there was no clear understanding of how to model the critical timing loop that the CNC maintains. Indeed if the model were created today this idea would be accounted for in the earliest iterations by using a timed control token that was passed from component to component and allowed execution of the sub-system only when it was present. If the control token expired prior to being passed back to the CNC the timeout condition could be detected and the error propagated through the system. While this has not be factored into the model, what came from the investigation of this question is a model of a time out condition which is detailed in section 3.4.

The next aspect of the system that was only partially accounted for is the idea of in-order execution of some aspects of the system such as G-Code processing. The current model fails to correctly implement this logic, however this is not catastrophic as no attempt is made to correctly simulate the mechanical movement of the system or to verify that the output part matched the requested one. This question motivated an investigation of how to construct a FIFO buffer within the CPN, the results of

which is discussed in section 3.3.

In order to model the idea of an emergency stop message being propagated to all components as quickly as possible it became desirable to represent a TCP/IP broadcast message, the result of which is described in section 3.2.

Finally, the initial approach of retaining a single state for each component proved problematic in some cases. Although the following insight was not incorporated into the model in a wholesale fashion, but does appear to a limited degree within the CNC in the tuple of states representing the states of the compile cycles. After the investigation of TCP/IP broadcast messages in a CPN it became apparent that a tuple of states was just as easy to work with as a single state was and posed no significant challenges beyond refactoring the model. This would be useful in cases where commands are dispatched and the component needs to await a reply, such as when the CNC is initializing the interpreter and the interpolator. There is no reason why the real machine would wait for verification of successful initialization of a component as modelled here. Instead several such instructions could be dispatched and the system can easily be modified to retain all the different interim states until the components report ready or error. The current model attempts to address this by making all of the initialization occur in simultaneous untimed steps which imposes an artificial sequencing that is functional if awkward.

3.2 Broadcast Message Proof of Concept

Broadcast messages were incorporated into the model late in the CPN model creation. The simplest way to implement this idea and avoid undertaking substantial effort to re-colour all the associated nodes, arc-guard inscriptions and ML programs seemed to be to use lists instead of singular destinations in the destination field of the PKT

colour. Once this change was completed, all that was required was to implement the proof of concept below to the top level net and adjust the packet consuming transitions to accept lists of single values in place of direct values. It is clear that this can be extended trivially to timed tokens where the implementer can determine if they wish the original time stamp to be preserved or a new one applied as the net will proceed with the broadcast only once the global time step advances, if required.

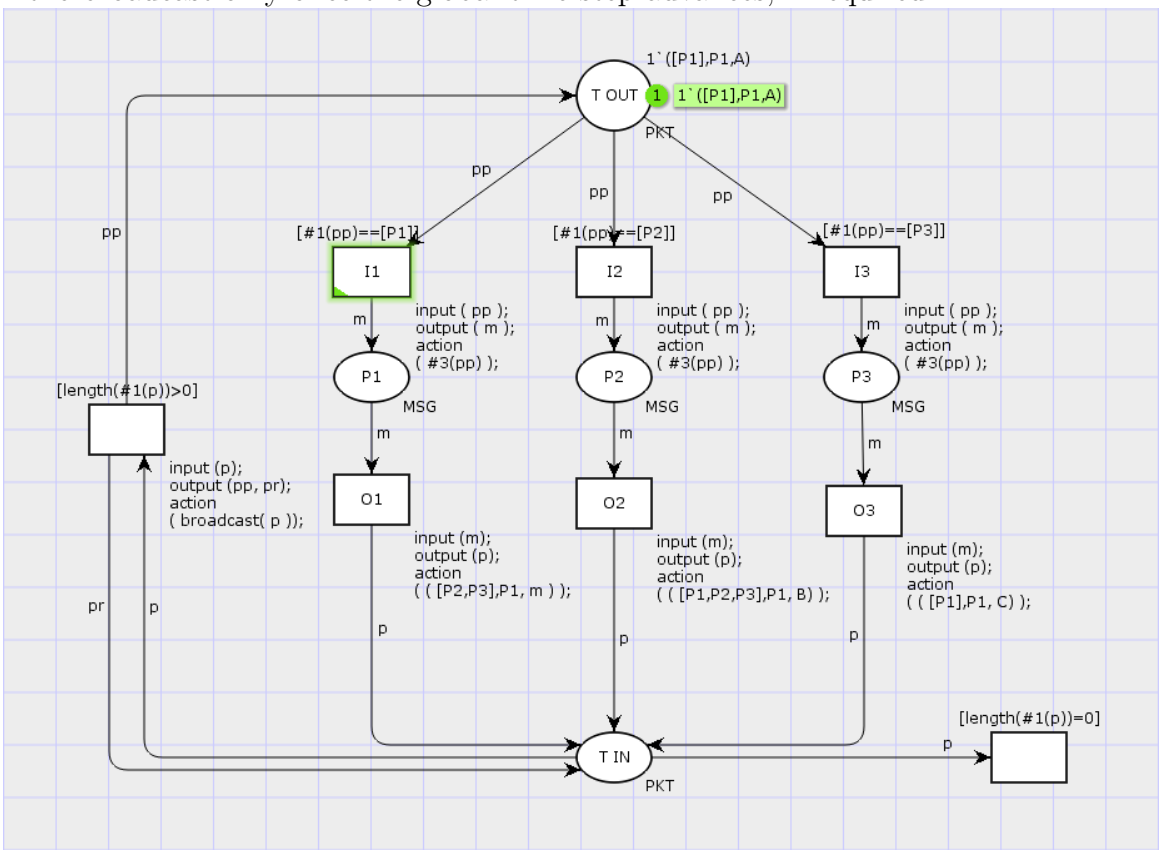

Figure 3.1: A CPN with broadcast messages

For this the CPN in figure 3.1 define the colours:

- messages : MSG = $\{A|B|C\}$
- packets : PKT = $[] :: PL^n \times PL \times MSG$

where [] :: is list concatenation and $0 <= n <= 3$. Also consider initial marking $M_0 = \{T\ IN = 1`([P1], P1, A)\}$, that is, a packet in place $T\ IN$ whose destination is

a list containing only place P1, origin is P1 and message is A.

The broadcast function shown in listing 3.1 accepts one PKT token as input and produces two PKT tokens as output. The first output token contains the head of the list of destination systems from the place list of the input token. The second token contains the tail of the list, or an empty list, as its first element. Both tokens preserve copies of the 2nd and 3rd elements of the input token. It is clear that if we so desired we could avoid updating the arc guard transitions of may other nodes by simply omitting the "::[]" in the generation of the first packet, however we would then require a different colour for incoming messages and outgoing messages to retain type correctness in the net.

Listing 3.1: broadcast function

```
fun broadcast( p:PKT ) =
   if length(#1(p)) = 1 then
      (p,([],#2(p),#3(p)))
   else
      ( (hd(#1(p))::[],#2(p),#3(p)),
        (tl(#1(p)),#2(p),#3(p)) )
```

The initial marking satisfies only the guard condition of transition I1, thus enabling it, which when fired unpacks message m and produces token m in place P1. The outgoing message is packaged into a PKT by firing of transition O1 to have destinations of places P2 and P3. In this illustration the message is simply being retransmitted to the other nodes but it is clear that a net of arbitrary complexity could replace P1, P2 or P3 from the earlier discussion of Hierarchical nets. P1 broadcasts the message to P2 and P3, and so forth. Place T IN represents a gathering place for packets which may need to be broadcast. In the Steel Cutting system CPN the

functionality of T IN and the connected unlabelled transitions would be part of the TCP/IP interconnect. We could split the TCP/IP interconnect into send and receive components, where the receive component would handle the broadcast strategy. The other side of place T IN is a transition whose guard accepts packets with 1 or more destinations. This transition produces 2 packets labelled *pp* and *pr*. Packet *pp* is the first output of *broadcast* that contains a single destination and the packet *pr* contains an element that contains either an empty list or the remaining destination nodes. The guard functions on the connected transitions ensure that packets whose first element is a length 0 list are consumed by a sink transition. The reason for generating the "sink" token is that SML requires all exit paths from a function to be of the same type and so we must produce a packet to satisfy strict type correctness.

In this example packet p will be processed into packets $pp = ([P2], P1, m)$ and $pr = ([P3], P1, m)$. Subsequently pr is passed to broadcast as p and produces $pp = ([P3], P1, m)$ and $pr = ([], P1, m)$. Token pr is consumed and no further token is created.

While it appears our message has been broadcast to a list of destinations it should be noted that unless "must fire" semantics are enforced, such as by the use of timed tokens, there is no guarantee that the tokens in T IN will be consumed prior those in T OUT or at all.

3.3 FIFO Places Proof of Concept

The model fails to achieve the goal of correctly modelling sequential execution within the system. In some cases the state transition functions are guarding the execution to prevent unintended state transitions in a way that does not exist in the real system. As demonstrated here it is possible to enable and maintain a FIFO structure but it

is not clear if new complications will arise that will result in a major change to the CPN.

To illustrate the difficulty, consider if we choose to model the flow of G-Code within the system and require that it maintain a strict ordering through the various components it initially seems that the TCP/IP interconnect would have to be modelled as a FIFO buffer. If the TCP/IP link is modelled as a FIFO state that contains timed values it is semantically clear that the time index should override the order of the values in the FIFO structure if they have an unrealized time index but it is not clear if this can easily be achieved or if this becomes quickly unmanageable. The particular semantics of a mixed timing FIFO list are unclear and would need to be properly defined before sequential execution could be fully implemented in this system. It may be possible to create such a transmission without resorting to a FIFO buffer in the TCP/IP interconnect, but it is clear from the order of magnitude time difference in issuing movement commands to their resolution that a buffer will be required somewhere within the system.

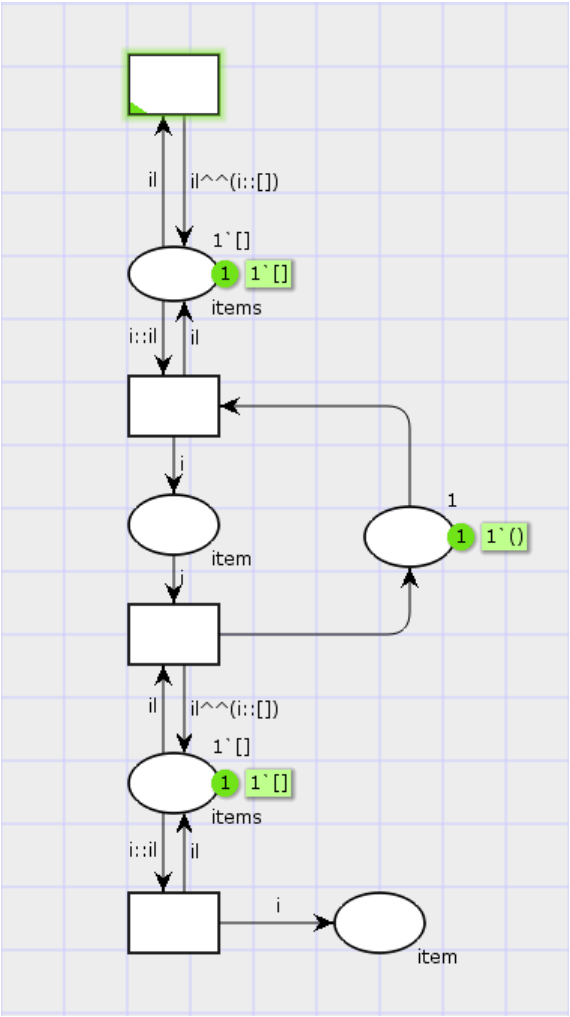

Figure 3.2: A net that has FIFO processing

The concurrent aspects of CPNs inherently removes the of ordering of tokens. Consider a place whose input arc indicates that the tokens are values from a list;

each token is created by popping the top element off a stack of values. Concurrent execution of these steps allows for some or all of the tokens to be created simultaneously and after creation they contain no semantic information about ordering, thus they may be used by a subsequent transition in any order. In order to achieve any ordering, a structure has to be imposed on a solution that makes clear how the data is ordered. The net presented in figure 3.2 demonstrates one method by which to impose a FIFO structure on the data. The pattern derives from a fairly standard method of developing a FIFO queue in SML and introduces a bottle neck where processing occurs on individual items. It differs from the net presented in (Mulyar & van der Aalst, 2005) by demonstrating the singleton path explicitly.

The initial marking shows 3 tokens: 2 empty lists (1‘[]) and an unit token (1‘()). The enabled transition at the top randomly generates an item from the colour set *item* and the arc expression *il^^(i::[])* prepends *i* to an empty list and then concatenates it to the tail of *il*. Once the list in the upper place is populated with at least 1 element the lower transition may fire because the arc expression *i::il* is fulfilled; i is the first element in list il and is only satisfied for lists of at least 1 element. The tail of the list, the list minus the first element, is returned to the upper place. The unit token is consumed and no further elements from the list can be processed. A token with the value of *i* is created in the central place. Once the token is passed from the central place to the lower one the unit token is recreated and more data can be passed into the FIFO critical portion. The list is reassembled in order in the lower place and the rest of the net can resume.

3.4 Timeout Condition Proof of Concept

As described in section 2.4.4, the CNC operates under strict timing conditions. The process must complete within a requisite period and relinquish control back to the CNC and or else generate a timing fault within the system. The present system captures detecting the timeout in a manner that allows each module to establish its own timing criteria. The most difficult aspect was avoiding scenarios where untimed tokens prevent the global clock from advancing such as a loop that always allowed a transition to be enabled. Conditions like this may be detectable by introducing a temporal logic such as Kripke semantics to assert undesirable conditions. This particular scenario may be described as *it is never the case that in the future we can always enable a transition without advancing the global clock* or some similar proposition. Whether this opens a new analysis route or if this can be proven using current CPN analysis techniques is not addressed here.

A proof of concept modelling a timeout condition is presented here. It uses a mix of timed and untimed tokens though it is clear that all the tokens could be timed if so desired. It is not obvious if this can be somehow simulated by untimed tokens alone but I conjecture that it may be possible for this particular net, but there will be a severe penalty in complexity to attempt to enforce a countdown to mimic time steps in tokens alone.

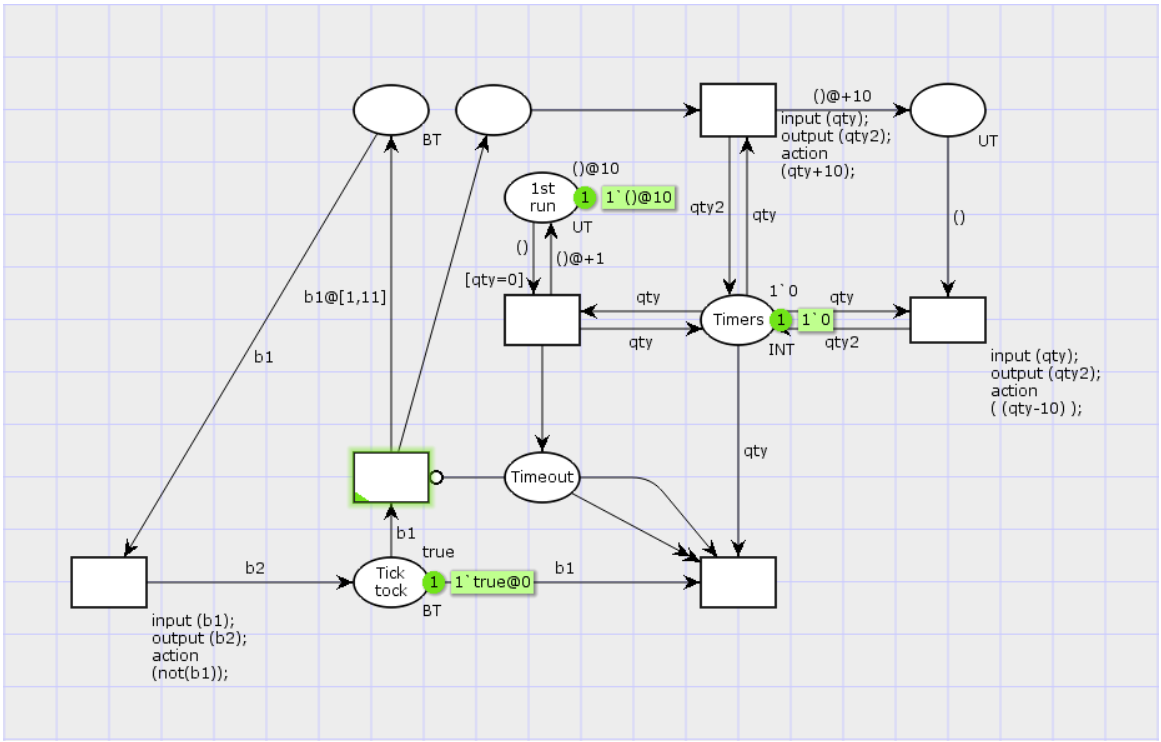

Figure 3.3: A net implementing a timeout condition

The initial marking provides a value in place *Tick Tock* which may be modified; in this net it will simple toggle back and forth between true and false but any more sophisticated processing loop or colour could replace it at will. Where the left side of the net is free to advance arbitrarily, the right hand side of the net receives a unit token when a boolean token passes through the transition connected to *Tick Tock* and represents a countdown to a time out. The transition that passes the unit token also requires the token from the *Timers* place in order to fire. This token is simply returned to the Timers place with its value incremented. The unit token is timed 10 steps in the future, which is our arbitrarily chosen timeout value. When the timer matures and this token becomes available and the *Timers* token is decremented. If the value of the token in Timers becomes 0 it will pass the guard condition and a token will appear in the *Timeout* place. The token in 1st run ensures that this does not happen on the first time step (false positive). The inhibitor arc on the Timeout

place prevents it from firing if a unit token were about to be provided to the Timeout mechanism and the reset arc clears all tokens from the *Timeout* place.

This scenario requires *must fire* semantics on transitions or for the *Timer* token to be timed in such a way as to prevent the rest of the net from advancing until it is moved to the *Timeout* state. This can be done by using integer values for timing except on the arc signalling a failure where you add a fractional time step. The theory allows for real valued time steps (W.M.P. van der Aalst, 2013) but CPN Tools does not[1]. In order to side step this issue in simulation the net takes advantage of the semantic that if a transition is possible with an untimed token, it must occur before the time step will increment. Thus by leaving the *Timers* token untimed it must progress to a *Timeout* state before any other timed tokens can be enabled.

[1]This has been changed in version 4.0 of CPN Tools.

Chapter 4

Conclusions

Using the graphical approach to constructing Coloured Petri Nets I was able to produce a partial model of a real world concurrent steel cutting system. The exploration of the design of the system led to the incorporation of several other theoretical extensions to CPNs such as hierarchical nets, FIFO structures and timed nets. These extensions contributed to a more complete picture of the real world system and raised some new questions. One such question is to define what a timed or mixed timed/untimed token FIFO structure may semantically mean and whether such a structure can be designed. If we approach this to mean untimed and enabled timed tokens in the order they appear ahead of unenabled timed tokens then a simple recursive definition may solve the problem of searching the list for the next available token though what the impact of this on analysis of the system might be is unclear.

Although there exist many techniques for analyzing the formal structure of the CPN model the introduction of timing to the model suggests that certain questions may be formally uncomputable. Additionally as some sections of the CPN are somewhat underdeveloped it is clear that some unintended behaviour may be discovered. Be that as it may, it is equally clear that some goals can be demonstrated to have

been met by simulation. For example, user control is simulated by an input stream of messages from the HMI to the PLC. Emergency stops are distributed through the system from the CNC to all components by a combination of broadcast messages and net inscriptions.

Investigating the mathematical definition of Coloured Petri Nets I cast colours as algebras to further formalize the semantics of Coloured Petri Nets and uncouple them from any dependence upon SML. This also opens some interesting questions. Does the implementation of CPN Tools rely upon any side effects that are present in SML but that would be disallowed by a pure functional language like Haskell? If the implementation were changed to a language which supported lazy evaluation, is there any gain to be made in allowing places to be typed in a lazy fashion or is all the desirable functionality achieved by declaring the type to be a union of colours? From a theoretical point of view does the work on the computability of alegbras open any new avenues for the analysis and classification of CPNs?

Although many aspects of the real world system are omitted to make the scope of the project more manageable these findings support the concept that Coloured Petri nets are viable mechanisms for modelling modern systems.

Chapter 5

Appendix

5.1 Colour Definition of Component Messages

The first colour definition is an enumeration of the components in the system:

$COMPONENT = with\ CNC|CNCIR|CNCIP|PLC|HMI|CPC1|CPC2|CPC3$

$|CPC4|CFG|GAS$, where

- CNC is the CNC Kernel
- CNCIR is the CNC interpreter
- CNCIP is the CNC interpolator
- PLC is the programmable logic controller
- HMI is the human-machine interface
- CPC1, CPC2, CPC3, CPC4 are generic names for compile cycles 1-4
- CFG is the configuration program communicated with from CPC3
- GAS is the gas controller communicated with from CPC3

The color sets for the messages of each component are defined by:

- CNC_MSGS = $cSTART\|cSTOP\|cNOP\|cDATA$
- IP_MSGS = $ipNOP\|ipINITOK\|ipDATA\|ipERROR$

- IR_MSGS = $rNOP \| rDOINIT \| rINITOK \| rERROR \| rDATA$
- PLC_MSGS = $pNOP \| pINIT\_HMI \| pINITOK \| pMOVE \| pLOAD \| pFLERR \| pFLOK \| pDORUN \| pSTOP \| pRESET \| pHEARTBEAT \| pDATA$
- HMI_MSGS = $hINITOK \| hERROR \| hNOP \| hRUN \| hSTOP \| hRESUME \| hRESET \| hLOAD \| hMOVE \| hESTOP$
- LEETT_MSGS = $lHEARTBEAT \| lTRANSLATE \| lTRANSERR \| lTRANSOK \| lNOP$

5.2 ML Representations of System States

- CNC_STATES = $cINIT \| cINITIR \| cINITIP \| cINITPLC \| cLOADED \| cREADY \| cERROR$
- CPC_STATES = $cpsUNINIT \| cpsREADY \| cpsABSENT \| cpsERROR$
- IP_STATES = $ipINIT \| ipREADY \| ipFAULT$
- IR_STATES = $rINIT \| rINITCPCR1 \| rINITCPCR2 \| rINITCPCR3 \| rINITCPCR4 \| rREADY | rFAULT$
- PLC_STATES = $pINIT \| pINITHMI \| pREADY \| pERROR \| pLOADING \| pLOADED \| pRUNNING \| pSTOPPED$
- HMI_STATES = $hsINIT \| hsREADY \| hsLOADING \| hsLOADED \| hsMOVING \| hsRUNNING \| hsSTOPPING \| hsSTOPPED \| hsRESETTING \| hsABORTING \| hsERROR$
- CPC1_STATES = $lINIT \| lFIRSTHB \| lREADY \| lLOADREQ \| lBUSY \| lDONE \| lERROR$

5.3 ML Transition Functions

5.3.1 Initial States

The model takes into account the idea that the physical machine has a power up phase during which the timing conditions of the software are not enforced. It is during this phase that certain implementations will allow the CNC to communicate with the operating system for non-deterministically timed functions such as memory allocation. Once the start up phase is completed the system enters into a state where it is under strict timing conditions and must remain responsive. A loss of responsiveness could prevent emergency shut down procedures from activating and could represent a serious safety concern.

This is represented in the system as the set of states $\{cINIT, rINIT, pINIT, hsINIT, lINIT\}$. Each transition function is passed the initial state token and a sequence of start up steps is performed that will be described in the appropriate subsection.

5.3.2 Understanding the Code Segments

Many of the transition functions listed below will generate either a message or a packet. The variable names listed on the arcs are a visual hint with names like *p1* indicating a packet token and *m1* signifying a message token. As defined in section 2.4.1 syntactic elements like *cmsg* appear to provide the syntactic link to the original type that was included in the union definition of messages.

It is also necessary to observe the output notation associated with the transition. The notation states what type of tokens must be created by the function. The return is a tuple of the required tokens, thus the notation (token1, token2) should not be

confused as a single token with a product set of token1 and token2's data types.

5.3.3 CNC Kernel Transitions

Listing 5.1: cncTransition function

```
fun cncTransition( cs : CNC_STATES, m:MSG ) =
    if m = hmsg(hESTOP) then
        ( cERROR, ([CNCIR, CNCIP], CNC, hmsg( hESTOP )) )
    else if cs = cINIT then
        ( cINITIR , ([CNCIR], CNC, cmsg( cSTART )) )
    else if cs = cINITIR andalso m = rmsg(rINITOK) then
        ( cINITIP, ([CNCIP], CNC, cmsg( cSTART )) )
    else if cs = cINITIP andalso m = ipmsg(ipINITOK) then
        ( cINITPLC, ([PLC], CNC, cmsg( cSTART )) )
    else if cs = cINITPLC andalso m = pmsg(pINITOK) then
        ( cREADY, ([CNC], CNC, cmsg( cNOP)) )
    else if cs = cERROR andalso m = pmsg(pRESET) then
        ( cREADY, ([CNCIR, CNCIP], CNC, pmsg(pRESET)) )
    else if cs = cREADY andalso m = pmsg(pMOVE) then
        ( cREADY, ([MOTORS, PLC], CNC, cmsg(cMOVE)) )
    else
        ( cs, ([], CNC, cmsg( cNOP )) );
```

5.3.4 CNC Interpreter Transitions

Listing 5.2: irTransition function

```
fun irTransition ( s:IR_STATES, cps:IR_CPCs, p:PKT) =
```

```
2   if s = rINIT then
3      if #3(p) = cmsg( cSTART ) then
4         ( rINITCPCR1, icsTransition(#3(p), #2(p), cps), ( [CPC1], CNCIR, rmsg(
             ↪ rDOINIT ) ) )
5      else
6         ( s, icsTransition(#3(p), #2(p), cps), ([], CNCIR, rmsg(rNOP)) )
7   else
8      if #3(p) = hmsg( hESTOP ) then
9      (* Figure out which of the CPCs is installed so we can repeat the ESTOP message *)
10     (* IE: the output packet will look like (rFAULT, icsTransition(hmsg(hESTOP), #2(p),
          ↪ cps), (([CPC1,CPC2,CPC3,CPC4],CNCIR, hmsg( hESTOP)))) *)
11        ( rFAULT, icsTransition(#3(p), #2(p), cps),((if #1(cps)=cpsABSENT then [] else
             ↪  [CPC1])^^(if #2(cps)=cpsABSENT then [] else [CPC2])^^(if #3(cps)=
             ↪ cpsABSENT then [] else [CPC3])^^(if #4(cps)=cpsABSENT then [] else
             ↪ [CPC4]),CNCIR, #3(p) ))
12     else if #3(p) = pmsg( pRESET ) then
13        ( rREADY, icsTransition(#3(p), #2(p), cps),((if #1(cps)=cpsABSENT then []
             ↪ else [CPC1])^^(if #2(cps)=cpsABSENT then [] else [CPC2])^^(if #3(cps
             ↪ )=cpsABSENT then [] else [CPC3])^^(if #4(cps)=cpsABSENT then []
             ↪ else [CPC4]) ,CNCIR, #3(p) ))
14     else if #3(p) = cpcmsg( cpINITOK ) orelse #3(p) = cpcmsg( cpUNINSTALLED )
          ↪ then
15        if s=rINITCPCR1 then
16           ( rINITCPCR2, icsTransition(#3(p), #2(p), cps), ( [CPC2], CNCIR, rmsg(
                ↪ rDOINIT ) ) )
17        else if s = rINITCPCR2 then
```

```
18              ( rINITCPCR3, icsTransition(#3(p), #2(p), cps), ( [CPC3], CNCIR, rmsg(
                    ↪ rDOINIT ) ) )
19          else if s = rINITCPCR3 then
20              ( rINITCPCR4, icsTransition(#3(p), #2(p), cps), ( [CPC4], CNCIR, rmsg(
                    ↪ rDOINIT ) ) )
21          else if s = rINITCPCR4 then
22              ( rREADY, icsTransition(#3(p), #2(p), cps), ([CNC], CNCIR, rmsg(rINITOK)))
23          else
24              ( s, icsTransition(#3(p), #2(p), cps), ([CNC], CNCIR, rmsg(rNOP)) )
25      else if #3(p)=cmsg(cDATA) orelse #3(p) = cpcmsg(cpDATA) orelse #3(p) = pmsg
            ↪ (pDATA) then
26          (s, icsTransition(#3(p),#2(p),cps),([CNC,CNCIP],CNCIR,#3(p)) )
27      else
28          ( s, icsTransition(#3(p), #2(p), cps), ([], CNCIR, rmsg(rNOP)) );
```

5.3.5 Translation Tool Compile Cycle Transitions

Listing 5.3: leettTransition function

```
1  fun leettTransition( ls:LEETT_STATES, m:MSG) =
2      if m = pmsg(pHEARTBEAT) then
3          ( ([PLC], CPC1, lmsg(lHEARTBEAT)), ls )
4      else if m = hmsg(hESTOP) then
5          ( ([],CPC1,lmsg(lNOP)), lERROR )
6      else if ls = lERROR andalso m = pmsg(pRESET) then
7          ( ([], CPC1, lmsg(lNOP)), lREADY )
8      else if ls = lINIT andalso m = rmsg(rDOINIT) then
9          ( ([CNCIR], CPC1, cpcmsg( cpINITOK )), lREADY)
```

```
    else if ls = IREADY andalso m = pmsg(pLOAD) then
        ( ([CPC1], CPC1, lmsg(INOP)), ILOADREQ)
    else if ls = IBUSY andalso m = lmsg(ITRANSOK) then
        ( ([PLC], CPC1, lmsg(ITRANSOK)), IREADY)
    else if ls = IBUSY andalso m = lmsg(ITRANSERR ) then
        ( ([PLC], CPC1, lmsg(ITRANSERR)), IREADY)
    else
        ( ([], CPC1, lmsg(INOP)), ls)
```

Listing 5.4: Execution Simulation function

```
fun simTranslate( m:MSG ) =
    if m = lmsg( ITRANSLATE ) then
        (* There are two common sources of errors to consider here:
        -- The file is not parasable due to incorrect formatting
        -- There is an error retrieving the file such as disk failure

        I think it's sufficient to show how an error can be signalled, thus I will not distinguish.
        *)

        if Random.randRange(1,100)(Random.rand(1,100)) < 5 then
        (* Simulate load error due to format or hardware error *)
            lmsg(ITRANSERR)
        else
            lmsg(ITRANSOK)
    else
        lmsg(INOP)
```

5.3.6 CNC Interpolator Transitions

Listing 5.5: Interpolator Transition function

```
fun ipTransition( m:MSG, s:IP_STATES ) =
    if s = ipINIT then
        if m = cmsg( cSTART ) then
            ( ([CNC], CNCIP, ipmsg( ipINITOK ) ), ipREADY )
        else
            ( ([], CNCIP, ipmsg(ipNOP)) , s)
    else
        if s = ipREADY andalso ( m = cmsg( cDATA ) orelse m = cpcmsg(cpDATA) orelse
            ↪ m = pmsg(pDATA) ) then
            ( ([CNC], CNCIP, ipmsg( ipDATA ) ), ipREADY )
        else if m = hmsg(hESTOP) then
            ( ([], CNCIP, ipmsg( ipNOP )), ipFAULT)
        else if m = pmsg(pRESET) then
            ( ([], CNCIP, ipmsg( ipNOP )), ipREADY )
        else
            ( ([], CNCIP, ipmsg(ipNOP)) , s);
```

5.3.7 PLC Transitions

Listing 5.6: plcTransition function

```
(*
inputs : current state, message
output: output message in packet, new state

pINIT – Initial state, does not respond to messages until the CNC initializes with a message
pINITHMI – The PLC has requested the HMI to initialize.
pREADY – The PLC is not engaged in any particular task and is ready to respond to requests
pERROR – The PLC is in an error state
pLOADING – The PLC has processed a request to load a file from the hard drive and
    ↪ requested file translation
```

```
10 pLOADED – The PLC has succeeded in loading the translated file and is ready to execute the
     ↪ program
11 pRUNNING
12 pSTOPPED
13 *)
14
15 fun plcTransition ( ps:PLC_STATES, m:MSG ) =
16     if ps = pINIT then
17         if m = cmsg( cSTART ) then
18             ( ([HMI], PLC, pmsg(pINIT_HMI)), pINITHMI )
19         else
20             ( ([PLC], PLC, pmsg(pNOP)), ps )
21     else if ps = pINITHMI then
22         if m = hmsg( hINITOK ) then
23             ( ([CNC], PLC, pmsg( pINITOK )), pREADY )
24         else
25             ( ([PLC], PLC, pmsg(pNOP)), ps )
26     else if m = hmsg( hESTOP ) then
27         ( ([CNC],HMI,hmsg(hESTOP) ), pERROR )
28     else if m = hmsg(hRESET) then
29        ( ([CNC], PLC, pmsg( pRESET )), pREADY )
30     else if m = cmsg( cMOVE ) then
31        ( ([HMI],PLC,cmsg(cMOVE)), ps)
32     else if (ps = pREADY orelse ps = pLOADING orelse ps = pLOADED) andalso m =
         ↪ hmsg(hMOVE) then
33         ( ([HMI,CNC], PLC, pmsg( pMOVE )), ps )
34     else if ( ps = pREADY orelse ps = pLOADED ) andalso m = hmsg(hLOAD) then
```

```
        ( ([CPC1], PLC, pmsg( pLOAD )), pLOADING)
    else if ps = pLOADING andalso m = lmsg( ITRANSOK ) then
        ( ([CNC], PLC, pmsg( pLOAD )), pLOADING)
    else if ps = pLOADING andalso m = lmsg( ITRANSERR ) then
        ( ([HMI], PLC, pmsg( pFLERR )), pERROR)
    else if ps = pLOADING andalso m = cmsg( cLOADOK ) then
        ( ([HMI], PLC, pmsg( pFLOK)), pLOADED)
    else if ( ps = pLOADED ) andalso m = hmsg(hRUN) then
        ( ([CNC], PLC, pmsg( pDORUN )), pRUNNING )
    else if ( ps = pLOADING orelse ps = pRUNNING ) andalso m = hmsg(hSTOP) then
        ( ([CNC], PLC, pmsg( pSTOP)), pSTOPPED )
    else if ps = pSTOPPED andalso m = hmsg(hRESUME) then
        ( ([CNC], PLC, pmsg( pDORUN )), pRUNNING )
    else
        ( ([PLC], PLC, pmsg(pNOP)), ps )
```

5.3.8 HMI Transitions

Listing 5.7: HMI Internal Transition function

```
(* Process USER GENERATED commands *)

fun hmiTransition( hs:HMI_STATES, hc:HMI_CMDS) =
(* If the command has to be relayed to the PLC, pass it along, otherwise hcNOP *)
if hs <> hsINIT andalso hc = hcESTOP then
(hsERROR, hmsg(hESTOP))
else if hs = hsINIT then
(* No user commands are permitted while the system is initializing *)
```

```
9  (hsINIT, hmsg(hNOP))
10 else
11 if hs = hsREADY then
12 (* The user may request a LOAD or MOVE, ignore other inputs *)
13 if hc = hcLOAD then (hsLOADING, hmsg(hLOAD))
14 else if hc = hcMOVE then (hsMOVING, hmsg(hMOVE))
15 else (hsREADY, hmsg(hNOP))
16 else
17 if hs = hsLOADED then
18 (* Once a file is loaded (in the CNC) it may be RUN, RESUMEd or the system can be
       ↪ RESET or a manual MOVE may be performed. *)
19 if hc = hcMOVE then (hsMOVING, hmsg(hMOVE))
20 else if hc = hcLOAD then (hsLOADING,hmsg(hLOAD))
21 (* Although not accounted for here, it is implicit that a new LOAD will free resources
       ↪ allocated by the old file before performing the operation *)
22 else if hc = hcRUN orelse hc = hcRESUME then (hsRUNNING, hmsg(hRUN))
23 (* RESUMEing from the start will have the same meaning as RUNning *)
24 else if hc = hcRESET then (hsREADY,hmsg(hRESET))
25 (* Although not modelled here, resources used by loading are freed during this operation. *)
26 else (hsLOADED, hmsg(hNOP))
27 else
28 if hs = hsRUNNING then
29 (* If the machine is in automatic mode the HMI operator may only STOP the job as other
       ↪ functions may be unsafe *)
30 if hc = hcSTOP then (hsSTOPPED, hmsg(hSTOP))
31 else (hsRUNNING, hmsg(hNOP))
32 else
```

```
33 if hs = hsSTOPPED then
34 (* If the machine is STOPPED it may be RESET or RESUMED. Here RUN is assumed to
       ↪ mean RESUME. A second STOP terminates the job, but not unload the program, as
       ↪ opposed to RESET which will clear the program from memory *)
35 if hc = hcRUN orelse hc = hcRESUME then (hsRUNNING, hmsg(hRESUME))
36 else if hc = hcRESET then (hsREADY, hmsg(hRESET))
37 else if hc = hcSTOP then (hsLOADED, hmsg(hNOP))
38 else (hsSTOPPED, hmsg(hNOP))
39 else
40 if hs = hsERROR then
41 if hc = hcRESET then (hsREADY, hmsg(hRESET))
42 else (hsERROR, hmsg(hNOP))
43 else
44 (hs, hmsg(hNOP))
```

Listing 5.8: HMI External Transition function

```
1  fun hmiExtTransition ( hs:HMI_STATES, m:MSG) =
2      if hs = hsINIT andalso m = pmsg( pINIT_HMI) then
3          ( hsREADY, hmsg( hINITOK ) )
4      else if hs = hsLOADING andalso m = pmsg( pFLOK ) then
5          ( hsLOADED, hmsg( hNOP ) )
6      else if hs = hsLOADING andalso m = pmsg( pFLERR ) then
7          ( hsERROR, hmsg( hNOP ) )
8      else if hs = hsMOVING andalso m = cmsg( cMOVE ) then
9          ( hsREADY, hmsg( hNOP ) )
10     else
11         ( hs, hmsg( hNOP) )
```

Listing 5.9: acXformTransition function

```
(* The axis transformation routine rewrites individual instructions
on the fly, and thus can be considered stateless *)

fun axTransform ( p : PKT ) =
if ( #2(p) = CNCIR ) then
if #3(p) = rmsg( rDOINIT ) then
( [CNCIR], CPC2, cpcmsg( cpINITOK ) )
else
( [CPC2], CPC2, xmsg( xNOP ) )
(* xNOP to xDATA? *)
else
p
```

Listing 5.10: Broadcast function

```
fun broadcast( p:PKT ) =
if length(#1(p)) = 1 then
(p,([],#2(p),#3(p)))
else
((hd(#1(p))::[],#2(p),#3(p)), (tl(#1(p)),#2(p),#3(p)))
```

References

Akshay, S., Genest, B., & Hélouët, L. (2014, November). *Timed Petri Nets with (restricted) Urgency.* Retrieved from https://hal.inria.fr/hal-01088997

Alur, R., & Dill, D. L. (1994, April). A theory of timed automata. *Theor. Comput. Sci.*, *126*(2), 183–235. Retrieved from http://dx.doi.org/10.1016/0304-3975(94)90010-8 doi: 10.1016/0304-3975(94)90010-8

Bérard, B., Cassez, F., Haddad, S., Lime, D., & Roux, O. H. (2013). The expressive power of time petri nets. *Theor. Comput. Sci.*, *474*, 1–20. Retrieved from http://dx.doi.org/10.1016/j.tcs.2012.12.005 doi: 10.1016/j.tcs.2012.12.005

Berthomieu, B. (2000, March). Oo programming styles in ml. *LAAS Report #2000111*. Retrieved 2015-08-03, from http://homepages.laas.fr/bernard/oo/ooml.html

Cpn tools. (2015). Retrieved 08-27-2015, from https://en.wikipedia.org/wiki/CPN_Tools

Figueiredo, J. C. A. D., & Kristensen, L. M. (1999). *Using coloured petri nets to investigate behavioural and performance issues of tcp protocols* (Tech. Rep.). Department of Computer Science, Aarhus University.

Jensen, K. (1981). Coloured petri nets and the invariant-method. *Theoretical Computer Science*, *14*, 317-336. Retrieved 2014-03-15, from http://dx.doi.org/

10.1016/0304-3975(81)90049-9

Jensen, K. (1997). *Coloured petri nets - basic concepts, analysis methods and practical use* (Second ed., Vol. 1). Berlin, Germany: Springer.

Jones, N. D., Landweber, L. H., & Lien, Y. E. (1977). Complexity of some problems in petri nets. *Theor. Comput. Sci.*, *4*(3), 277-299. Retrieved from http://dblp.uni-trier.de/db/journals/tcs/tcs4.html#JonesLL77

Kristensen, L. M., Christensen, S., & Jensen, K. (1998). The practitioners guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, *2*(2), 98-132. doi: 10.1007/s100090050021

Large integer color sets. (n.d.). Retrieved 08-28-2015, from http://cpntools.org/documentation/concepts/colors/declarations/colorsets/intinf_colour_sets

Livingstone, S. (2000). *Essi code versus g-code.* Retrieved 10/21/2015, from https://groups.google.com/d/msg/alt.machines.cnc/2rzK1_6gYIc/NJ8fU5VlGbcJ

Mulyar, N., & van der Aalst, W. (2005, April). *Patterns in colored petri nets* (Tech. Rep.). Department of Technology Management, Eindhoven University of Technology. Retrieved 2015-08-05, from http://wwwis.win.tue.nl/~wvdaalst/publications/p258.pdf

Murata, T. (1989, April). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*. Retrieved 2014-03-15, from http://embedded.eecs.berkeley.edu/Respep/Research/hsc/class.F03/ee249/discussionpapers/PetriNets.pdf

Peterson, J. L. (1981). *Petri net theory and the modeling of systems.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

Petri, C. A. (1962). *Kommunikation mit automaten* (Doctoral dissertation, Universitt Hamburg, Hamburg, Germany). Retrieved 2014-03-15, from http://edoc.sub

.uni-hamburg.de/informatik/volltexte/2011/160/pdf/diss_petri.pdf

Petri, C. A. (1966, January). *Communication with automata* (Tech. Rep. No. Supplement 1 to Technical Report No. RADC-TR-65-377). Griffiss Air Force Base, New York: Reconnaissance-Intelligence Data Handling Branch. Retrieved 2014-03-15, from http://edoc.sub.uni-hamburg.de/informatik/volltexte/2010/155/pdf/diss_petri_engl.pdf (Approved translation of Kommunikation mit Automaten.)

Popova-Zeugmann, L. (2013). The classic petri net. In *Time and petri nets* (p. 7-30). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-41115-1_2 doi: 10.1007/978-3-642-41115-1_2

Reisig, W. (2010). *Understanding petri nets.* Springer.

Reppy, J. (Ed.). (2002). *The standard ml basis library.* Retrieved 08-27-2015, from http://sml-family.org/Basis/index.html

Sannella, D., & Tarlecki, A. (1996). Mind the gap abstract versus concrete models of specifications. In W. Penczek & A. Szalas (Eds.), *Mathematical foundations of computer science 1996: 21st international symposium, mfcs' 96, crakow, poland, september 2 - 6, 1996. proceedings* (p. 114-134). Springer.

Srba, J. (2008). Comparing the expressiveness of timed automata and timed extensions of petri nets. In F. Cassez & C. Jard (Eds.), *Formal modeling and analysis of timed systems* (Vol. 5215, p. 15-32). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-85778-5_3 doi: 10.1007/978-3-540-85778-5_3

Tucker, J., & Zucker, J. (1998). Computable functions and semicomputable sets on many-sorted algebras. In D. G. S. Abramsky & T. Maibaum (Eds.), *Handbook of logic in computer science, volume 5* (p. 319-445). Oxford University Press.

van der Aalst, W. (n.d.). personal communication.

van der Aalst, W. (1993). *Interval timed coloured petri nets and their analysis.*

Wikimedia. (2007). *Hole cut out with a plasma cutter.* Retrieved 10/17/2015, from `https://en.wikipedia.org/wiki/File:Plasma_cut.jpg` (Resized from original, licensed Creative Commons http://creativecommons.org/licenses/by-sa/3.0/)

Wikimedia. (2013). *Plasma cutting with a cnc machine.* Retrieved 10/17/2015, from `https://en.wikipedia.org/wiki/Plasma_cutting#/media/File:Retro_Systems_Plasma_Cutting_Machine.jpg` (Resized from original, licensed Creative Commons http://creativecommons.org/licenses/by-sa/3.0/)

W.M.P. van der Aalst, M. W., C. Stahl. (2013). Strategies for modelling complex processes using colored petri nets. In *Transactions on petri nets and other models of concurrency vii* (p. 6-55). Berlin Heidelberg: Springer.