DaNAMiCS: Modelling Concurrent Systems

Byron H. Changuion, Ian Davies, Micheal A. Nelte

Supervisor: Prof P.S. Kritzinger

October 16, 1998

Contents

List of Figures

List of Tables

Abstract

Concurrent communicating systems are, by nature, complex. They are widespread through the modern technological world in fields as diverse as engineering and business science. Designing concurrent systems is a complex task which must result in an efficient, working system. The cost of implementing concurrent systems is high, and the price of failure of these systems can be devastating. One must ensure that the concurrent system is functionally perfect before implementation. In addition to this, the system should provide a certain level of performance under certain conditions.

The design of concurrent systems is facilitated by the use of abstract models. These models provide a simplified representation of the problem which can be readily understood and easily analysed for the appropriate properties. Concurrent system models can be built using a wide range of techniques. A popular formalism for the construction of such models is that of Petri nets. Petri nets provide an understandable graphical structure, and a solid mathematical background which makes them ideal for modeling concurrent systems.

Software tools are required to adequately model concurrent systems with Petri nets. DaNAMiCS is such a tool, and the project to construct this tool was undertaken in the Data Network Architectures Laboratory at the University of Cape Town. DaNAMiCS uses Generalised Stochastic Petri nets (GSPNs) as its underlying formalism, which allows it to perform both correctness and performance analyses. DaNAMiCS also provides the use of a subset of the Coloured Petri net class in the form of coloured tokens and the use of subnets for more abstract hierarchical models. Additionally, DaNAMiCS allows capacities to be imposed on the places of the Petri net and inhibitor arcs to be drawn, which increases the modeling power of the Petri net formalism to that of a Turing machine.

DaNAMiCS supports a wide array of tools and techniques for the analysis of Petri nets. Invariant analysis can be performed for correctness analysis or the entire coverability graph can be exhaustively generated. The existing algorithm for constructing the coverability graph was found to be incomplete and unable to cope with the extensions such as capacities and inhibitor arcs implemented in DaNAMiCS. Novel work was completed on a coverability graph construction algorithm to correctly analyse the new additions.

Since DaNAMiCS uses GSPNs as its formalism it is capable of analysing Petri nets for performance. This can be done by simulation or by means of steady state analysis. All the results of the analyses are presented to the user in a format which enhances the user's perception and draws attention to the significant details of the analysis.

The DaNAMiCS project succeeded in creating a tool to construct Petri net models of any real-world system and provides thorough analysis of these models. The environment in which this occurs is friendly and intuitive and may serve as a teaching tool for new-comers to Petri net theory. In summation, DaNAMiCS should serve as a good addition to the existing array of case and Petri net modeling tools.

Chapter 1

Introduction

1.1 Concurrent Communicating Systems

A concurrent communicating system is a system which consists of multiple sub-processes which work together towards some common goal. These sub-processes act independently but access the same resources and can perform tasks which will assist the functioning of other processes.

Concurrent communicating systems are commonplace in the modern world and particularly are found in such diverse fields as control engineering, network protocols, manufacturing systems, and business processes. In each of the above technologies, we find machines or persons interacting to achieve a common end.

Whether the goal is to manufacture widgets, to maximise profit or to minimise network traffic, the underlying idea is the same: to bring together many diverse talents in an efficient manner to complete a task.

This ideal is not simply attained. The task to design a concurrently interacting system which will serve its purpose efficiently is very complicated. There are many different ways to achieve this goal and it is infeasible to assess these all by hand. Thus, we require some system in which to specify and analyse concurrent systems for functional correctness and efficiency or performance. Petri nets can serve this purpose.

1.2 Petri nets

Petri nets are a concurrent systems modeling formalism that have been available since the early nineteen-sixties. They are well-liked because of their flexibility and extensibility. Petri nets have a solid mathematical backing and also possess an intuitive graphical format which is appreciated by theoreticians and practioners. In addition, Petri nets can be analysed efficiently to provide results on the functional correctness and performance of the system.

One uses Petri nets to create an abstract model of the system which can be more easily analysed by formal methods. Real-world systems cannot be directly analysed, and so this intermediate format is essential for formal analysis.

Petri nets, despite their worth as a modeling tool, are, unfortunately lacking in one area. If one builds a suitably detailed model of the system to be analysed, the Petri nets become unmanageably large. In addition, the analysis of these nets becomes infeasible by hand. Thus, there is a need for software tools to model the systems, and provide automatic analysis of the models.

1.3 DaNAMiCS

The DaNAMiCS project, proposed by Professor P.S. Kritzinger, was to design a tool to facilitate the modeling and analysis of Petri nets. DaNAMiCS takes the form of a software tool to aid the construction of Petri nets which can be used solve real-world problems.

Currently, Petri net modeling tools are mainly used in academic circles. However, businesses are starting to use them for the design and testing of protocols, production systems, enterprise modelling and other systems development. By providing a powerful and flexible modeling tool, DaNAMiCS should make using Petri nets more convenient and accessible to any systems developer.

DaNAMiCS extends and greatly enhances previous work in this field and presents the user with a wide range of tools for modeling with Petri nets. Many of the extensions to Petri nets have been implemented in DaNAMiCS which allows any possible system to be modeled.

DaNAMiCS is a flexible system which is robust and presents the user with sufficient modeling power to adequately specify and analyse the target system using Petri nets.

Chapter 2

Background

2.1 Overview

The development of tools to analyse concurrent communicating systems has been a longstanding interest of the DNA Laboratory. Many such tools have been developed as Honours or Masters projects. The DaNAMiCS project is related to two other projects completed in the DNA Laboratory. These projects were DNAnet, a Petri net editor, and DNAmaca a Markov chain analyser. In this chapter, we briefly discuss these two projects and introduce DaNAMiCS in this context.

2.2 DNAnet - A Petri Net tool

DNAnet was written in 1994 by a team of three Honours students [1] for the DNA Laboratory. It is a Petri net editor and analyser which supports the Generalised Stochastic Petri net class, and thus allows both correctness and performance analysis.

DNAnet performs correctness analysis by means of invariant and coverability graph analysis. The performance metrics can be obtained via simulation or by steady state analysis which is actually performed by USENUM, an external state space analyser. DNAnet provides a simple, intuitive user-interface and allows users to play the "token-game" via interactive animation.

DNAnet was well received and has been used by more than 150 persons, organisations, or companies since its release. However, since DNAnet uses GSPNs, its functionality is limited in terms of the types of systems that can be modeled. Moreover, DNAnet is only available on two platforms, and its interface has become outdated. Also, the analysis of the correctness or performance results that are attained in DNAnet is clumsy and time-consuming.

It is thus clear that DNAnet could be improved upon. However, before we discuss its successor, we will describe another tool in the history of DaNAMiCS – DNAmaca.

[1] W. Knottenbelt, A. Attieh, M. Brady

2.3 DNAmaca - A Markov Chain analyser

DNAmaca is a tool which can be used to solve large Markov chains. It was written in 1996 by W. Knottenbelt and uses analytical means to solve Markov chains. We will see in later chapters how Markov chains relate to Petri nets, but for the moment it is sufficient to know that the coverability graph of a Petri net can be analysed like a Markov chain.

Steady state analysis can be performed on Markov chains and yields exact results. This allows one to exactly calculate the performance metrics which are of interest to Petri net modeled systems.

DNAmaca is a computationally heavy program which requires vast memory resources and is extremely CPU intensive. Ideally, we would like DNAmaca to run on a fast, dedicated machine. In DNAnet, DNAmaca is tied to the application, which may slow the users machine considerably, and even render it unusable for the duration of the calculating period.

We would like a system where DNAmaca run on any machine, chosen by the user, and not slow the users terminal. In this way we modularise the components of the Petri net editing and analysing system. Thus, we introduce DaNAMiCS, which is a vast improvement over DNAnet and includes the ability to network [2] to DNAmaca.

2.4 DaNAMiCS

DNAnet was a great success. It was developed to fill the gap in the existing array of Petri net editors by being very flexible and allowing more advanced modeling techniques.

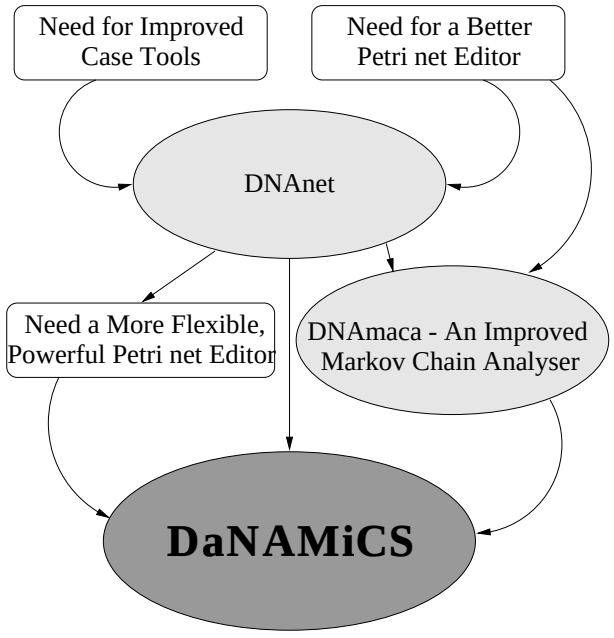


Figure 2.1: Background to the DaNAMiCS project

[2] See “Implementation Approach for a description of this networked connection”

However, current users of DNAnet have made several suggestions for improvement. These suggestions were highlighted above, and thus a new tool, DaNAMiCS, was proposed to incorporate some of these suggestions, and to further extend the flexibility of the Petri net editor.

Ideally, DaNAMiCS will be used as a replacement for DNAnet. Because of DNAnet's large user-base, DaNAMiCS was designed to support all existing DNAnet functions as well as the suggested new functions. DaNAMiCS should also be thoroughly portable, and to this end it was built in Java, which can run on most platforms.

DaNAMiCS is one of the first in the new generation of case tools to be created by the DNA Laboratory. It will fit into the existing set of tools and integrate well into any new tools developed.

Chapter 3

Petri net Theory

3.1 Overview

Petri nets were first conceived in 1962 by Carl A. Petri [4] as a formalism for describing concurrent and synchronous systems. Petri nets are a formal mathematical tool with a graphical format useful for describing and studying information processing systems. The systems modeled are usually concurrent, asynchronous, distributed, parallel, non-deterministic or stochastic.

As a mathematical tool, they can be used to formulate state and algebraic equations, and other mathematical models that govern the behaviour of systems. This will be discussed in more detail later in this chapter. Graphically, they can be used as communication aids, similar to flow charts or block diagrams with tokens simulating the dynamic behaviour of a process. They therefore provide a powerful medium for communication between theoreticians and practitioners.

Since their introduction, they have been extended in many ways. Petri nets were extended to Generalised Stochastic Petri nets which include time and allow performance analysis. The models themselves have become less complex with the addition of subnets and coloured tokens.

Subnets are a form of abstraction which allow hierarchical modeling. Coloured tokens ease the creation of the net by narrowing the gap between the physical problem and the Petri net model of the solution. The extensions made to Petri nets implemented in DaNAMiCS have this as one of their chief goals.

3.2 Petri net Structure

A Petri net is a directed bipartite graph[1] consisting of *Places* and *Transitions*. Graphically, the places are represented as circles, and transitions are represented as rectangles. The edges in the graph are represented as arrows.

3.2.1 Static Petri net Structure

Places and transitions are connected by directed arcs. We work with a general Petri net class, in which there is no limit on the number of arcs that can be drawn to or from places or transitions. Each place in the Petri net has a *marking* which is the number of *tokens* present on that place.

[1] We may only connect nodes of different classes

Tokens are used to represent the value of a place. Their absence or presence could indicate a certain condition. The marking of a Petri net refers to the number of tokens on each of the places in that net. The *initial marking* of a Petri net is the marking in which the process or system starts in.

We now present a mathematical definition of a Petri net.

Definition 4.1 - A Simple Petri net
A Petri net is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ where,

- $P = \{p_1, ..., p_n\}$ is a finite, non-empty set of places,
- $T = \{t_1, ..., t_m\}$ is a finite, non-empty set of transitions,
- $P \cap T = \phi$,
- $I^-, I^+ : P \times T \to N_0$ are the backward and forward incidence matrices,
- $M_0 : P \to N_0$ is the initial marking.

The backward and forward incidence matrices describe the arcs in the net. The backward incidence matrix, I^-, defined in terms of its elements $c^-(i, j)$, is the weight of the arc from place i to transition j. Similarly, the forward incidence matrix, I^+, contains elements $c^+(i, j)$ which is the weight of the arc from transition j to place i.

This describes the static structure of a Petri net. Graphically, if the arc weight is less than two, DaNAMiCS does not display it. Thus, the default arc-weight is one. We will now describe the dynamic behaviour of Petri nets.

3.2.2 Dynamic Petri net Behaviour

So far, we have described Petri nets as a static model. However, Petri nets model dynamic systems and concurrent processes. We will now describe the dynamic behaviour of the Petri net model.

Basics

The idea of transitions *enabling* and *firing* must now be discussed. Firstly, we will define it mathematically.

Definition 4.2 - Dynamic Petri nets
Let $PN = (P, T, I^-, I^+, M_0)$ be a Petri net

- The *Marking* of a Petri net is a function $M : P \mapsto N_0$ where $M(p)$ is the number of tokens in place p.
- A transition, $t \in T$, is *enabled* at a marking M, denoted $M[t >$, *iff* $M(p) \geq I^-(p, t), \forall p \in P$.
- A transition, t, *fires*, if enabled at a marking M yielding a new marking M', where
  $M'(p) = M(p) - I^-(p, t) + I^+(p, t), \forall p \in P$.
  We denote this $M[t > M'$.

Enabling

If a transition is *enabled* then it is able to *fire*. In order for a transition to be enabled, there must be sufficient tokens on all of that transition's input places. By this we mean that, for each of a transitions incoming arcs, the weight of that arc must be less than or equal to the number of tokens on the associated place.

Firing

When an enabled transition fires, there is a change in marking. For each of the input places to a transition, i.e. the places connected to a transition's incoming arcs(I^-), the number of tokens on those places is decremented by the weight of the corresponding arc. Then, for each of the transition's output places, i.e. the places connected to each of the transition's outgoing arcs (I^+), we increment the number of tokens on those places by the weight of the arc.

We speak of the transition *consuming* tokens from its input places, and *producing* tokens on its output places.

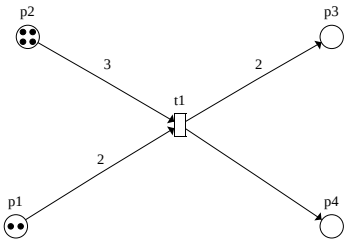


Figure 3.1: Transition t_1 enabled

Referring to Figure 3.1, we see that the transition is enabled because there are sufficient tokens on places p_1 and p_2. The marking in this net is expressed as the vector $(2, 4, 0, 0)$.

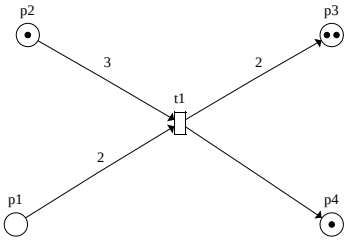


Figure 3.2: t_1 has been fired

In Figure 3.2, t_1 has been fired. We see that 2 tokens were consumed from place p_1, and 3 tokens were consumed from place p_2. One token was produced on place p_4. Note that the arc weight has been omitted and assumed to be one. Two tokens were produced on p_3. It is clear now that t_1 is no longer enabled, and the new marking is $(0, 1, 2, 1)$.

With the continued process of enabling and firing transitions, we build up a passage of tokens through the Petri net. This passage of tokens can be used to model any process or set of processes.

3.3 Generalised Stochastic Petri Nets

The Petri nets thus far discussed lack one vital component of any real world process – Time. The nets we have examined have modeled logical processes, but we are also concerned with the real-world factors involved in such processes, the most important of which is time. Thus, we extend the Petri net model to Generalised Stochastic Petri Nets (GSPNs). A mathematical definition follows.

Definition 4.3 - Generalised Stochastic Petri Nets
A GSPN is a 4-tuple $GSPN = (PN, T_1, T_2, W)$ where

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying Petri net,
- $T_1 \subseteq T$ is the set of *timed* transitions, $T_1 \neq \phi$
- $T_2 \subset T$ is the set of *immediate* transitions,
- $W = (w_1, ..., w_{|T|})$ is an array whose entry $w_i \in R^+$ where,
  - is a (possibly marking dependent) rate of a negative negative exponential distribution specifying the firing delay, when transition t_i is a timed transition, ie: $t_i \in T_1$ or
  - is a (possibly marking dependent) firing weight, when transition t_i is an immediate transition, ie: $t_i \in T_2$.

Generalised Stochastic Petri Nets introduce two new features to our model: timed and immediate transitions.

3.3.1 Timed Transitions

Timed transitions each have an associated rate of firing, w. When more than one timed transition is enabled, they compete with each other in a race model for the ability to fire first. The higher the rate, w, of a timed transition the greater the probability that it will fire first. Specifically, the probability P that transition t_j will fire first in a marking M_k is:

$$P = \frac{w_j(M_k)}{\sum_{i \in EN(M_k)} w_i(M_k)}$$

where $EN(M_k)$ is the list of all the enabled transitions at marking k.

Referring to Figure 3.3 we see a net with two transitions t_1 and t_2. t_1's rate is 5 while t_2's rate is 2. Thus there is a greater chance that t_1 will fire first, and the most probable marking from this net is $(0, 0, 1)$ as depicted in Figure 3.4.

The time passed spent in the marking k is known as the *sojourn time*. The sojourn time for a marking k is given by the exponential distribution:

$$f(t) = \lambda e^{-\lambda t}$$

In this way time is introduced to the Petri net model.

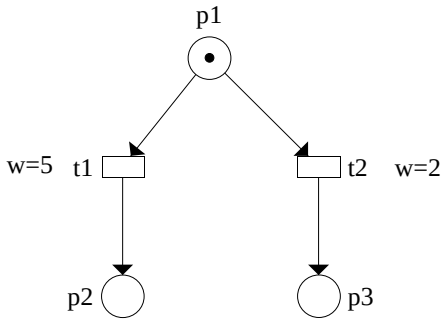


Figure 3.3: t_1 and t_2 are timed transitions

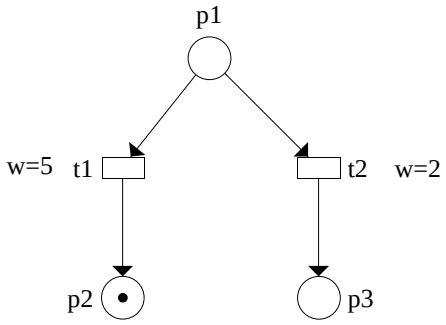


Figure 3.4: t_1 has fired first

3.3.2 Immediate Transitions

Immediate transitions are represented graphically with a filled rectangle and allow us to model timed transitions with infinite rates. This can be useful in a model where there is an instantaneous change in state. If an immediate transition is enabled, then no timed transition can be enabled.

Referring to Figure 3.5 we see, in 'A', that timed transition t_1 is not enabled, since immediate transition t_2 is also enabled. In 'B', t_2 has fired yielding our new marking.

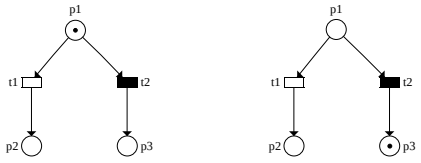


Figure 3.5: t_2 is an immediate transition

Many immediate transitions can be enabled simultaneously, however, and then they compete in a race condition, using their *weights* to determine which transition fires first. The probability equation is the same as the above for timed transitions, except the rate is substituted by the immediate transitions weight. The sojourn time for a marking which enables an immediate transition is zero.

3.4 Subnets

Subnets are a powerful Petri net development aid that enables one to design a hierarchical model. They are represented graphically by large rectangular blocks. Subnets do not change the mathematical properties of a Petri net is any way. They are constructs for simplifying the model and abstracting the complexity of a net to a manageable level.

3.5 Inhibitor Arcs

Inhibitor arcs increase the power of Petri nets to that of Turing machines [16]. They assist in modeling often simplifying the models. Inhibitor arcs allow one to model the *zero-test* in Petri nets. Thus a place in a Petri net can be tested whether or not it contains tokens. This allows us to model *XOR-switches* and *not* gates – which one cannot model with ordinary Petri nets.

Graphically, we represent inhibitor arcs with a circle on the transition. This notation comes from switching theory, where a small circle means *not*. Referring to Figure 3.6 we see two inhibitor arcs. This net models an XOR gate. If there is a token on place p_1 then transition t_2 cannot fire, and vice versa. We can see that this will model the XOR truth table 3.1.

The addition of inhibitor arcs to Petri nets is invaluable, and their applications in network modeling are vast. However, they do have complicating implications for the analysis of Petri nets. Their addition decreases the decidability of the model. This results in some models where the analysis is unknown.

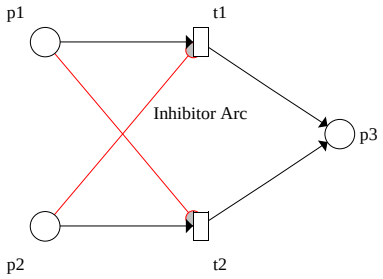


Figure 3.6: Inhibitor Arcs

	1	**0**
1	0	1
0	1	0

Table 3.1: The Truth Table for XOR

3.6 Limited Places

An addition to Petri nets is the notion of defining capacities on places. This means that we allow a place to contain no more than a specified number of tokens. This allows us to model such problems as the bounded-buffer producer-consumer problem with ease. A transition cannot be enabled if firing it would cause any limited place to exceed its capacity.

Petri nets containing limited places can be converted to Petri nets without any limited places. Therefore, allowing places to have a capacity constraint does not increase the modeling power of the net.

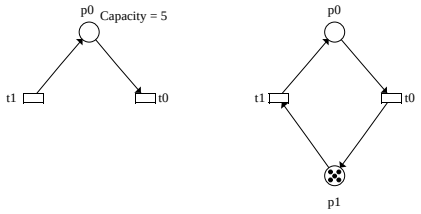


Figure 3.7: Limited places, and their equivalent nets

The limited place can be modeled with the introduction of an extra *bounding* place. The sum of the number of tokens on the *bounding* place and the original place must be kept at the desired capacity. This can be achieved by adding arcs that consume token from the bounding place connected to all transitions that produce tokens on the bounded place. Similarly transition that consume tokens from the bounded place need to produce the equivalent number of tokens on the bounding place. This is demonstrated in Figure 3.7. The two nets shown have equivalent functionality. We see then that limited places are used as a convenience only, and add no power to the Petri net model.

3.7 Coloured Tokens

The basic Petri net model has been extended to Coloured Petri Nets (CPNs) or Stochastic Coloured Petri Nets (SCPN). Coloured Petri Nets were introduced as a method of simplifying Petri nets. Petri nets can become very complex, even when modeling small systems, and this can be partly attributed to the fact that there is only one type of token.

Coloured Petri Nets attach a colour, or type, to each token in the net. Each token will then correspond more directly to some object in the real-world system or process. This is compared to the original Petri net, where each object type needed a place in the Petri net to correspond to the real-world process.

The transition firing rules must be altered to accommodate colours. The transition is only be enabled when certain combinations of coloured tokens are present on its input places. Furthermore, the coloured tokens that a transition produces can be manipulated. Mathematically, a CPN is defined as:

Definition 4.3 - Coloured Petri Nets
A Coloured Petri Net is a 6-tuple $CPN = (P, T, C, I^+, I^-, M_0)$ where,

- P is a finite and non-empty set of places,
- T is a finite and non-empty set of transitions,
- $P \cap T = \phi$,
- C is a colour function defined from $P \cup T$ into finite and non-empty sets
- I^- and I^+ are the backward and forward incidence functions defined on $P \times T$ such that
  $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall (p,t) \in P \times T$,
- M_0 is a function defined on P describing the initial marking such that
  $M_0(p) \in C^*, \forall p \in P$, where C^* denotes a multi-set.

Informally, this means that we now associate coloured tokens with our net and that the rules for enabling and firing a transition are different. Transitions have associated *modes* of firing. These modes specify different combinations of tokens on places that cause the transition to be enabled. They also define the resulting token production if the transition is fired in that mode.

CPNs are a modeling aid, and add no further mathematical power to Petri nets. All CPNs can be *unfolded* to ordinary Petri nets which can be analysed in the usual manner. CPNs allow a simpler representation of the model to be developed. CPNs and the addition of coloured tokens are discussed in further detail in Chapter 12.

3.8 Correctness Analysis

Correctness analysis ensures that the integrity of a system is maintained. It is important that our net conforms to certain conditions. We examine our nets' correctness in terms of two properties: *liveness* and *boundedness*. These two properties do not imply that the net is an accurate model of the system. We will use *correct* to mean that the net is live and bounded, while we will use *accurate* to mean that the model net represents the correct system.

In addition to these major properties we are also interested in some other properties of Petri nets. These include whether or not the net contains *home states*, whether the net is *safe*, and whether it is *persistent* or not. These terms will be outline in this chapter.

3.8.1 Liveness

The net is live.A Petri net is defined to be *live*[2] if for any marking reachable from the initial marking, M_0, it is possible to ultimately fire any transition in the net by progressing though some firing sequence [4].

It is perhaps easier to describe liveness in terms of its counterpart, *deadlock*. If a net is deadlocked, then no transition can be fired from the current marking. The net in Figure 3.2 is deadlocked, as t_1 can no longer fire. Liveness implies that deadlock can never be reached.

3.8.2 Boundedness

The net is bounded.A Petri net (N, M_0) is said to be *k-bounded* or simply *bounded* if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e., $M(p) \leq k$ for every place p and every marking $M \in R(M_0)$. [16]

This implies no place in the net can ever contain an infinite number of tokens. This is best described in the Producer-Consumer problem. Figure 3.8 shows a Petri-net modeling the Producer-Consumer problem. Place p_5 is unbounded since, by repeatedly firing t_1, and then t_2, we can increase its number of tokens without limit.

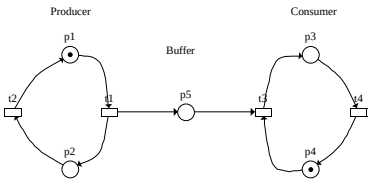


Figure 3.8: The unbounded-buffer Producer-Consumer problem

This Petri net models the unbounded buffer problem, and by analysing the Petri net, we can prove that it is unbounded. The net in Figure 3.9 models the bounded-buffer Producer-Consumer problem with a buffer-size of two, and this can be proved using Petri net analysis.

3.8.3 Home States

The presence of home states in a net has important implications for the real-world system being modeled. A home state is a marking that is reachable from every other marking in the net. This is crucial, say, for planning systems where, no matter what the current condition, the final goal may always be reached.

[2]This is also known as L4-liveness.

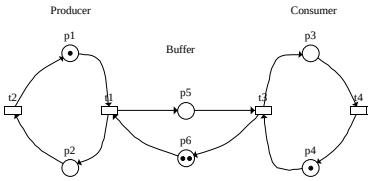


Figure 3.9: The bounded-buffer Producer-Consumer problem

3.8.4 Safeness

A Petri net is said to be *safe* if no place in the net can ever contain more then one token. This is a very important property for nets whose places are symbolic of conditions in the system. If the place contains a token then the condition is valid. If it does not, then the condition does not hold. A place containing more than one token in these types of nets has no logical meaning, and usually indicates that there is an error somewhere else in the design.

3.8.5 Persistence

A Petri net is said to be *persistent* if, for any two enabled transitions, the firing of one transition will not disable the other. A transition in a persistent net, if enabled, will remain enabled until it is fired. This is very useful in the context of parallel program schemata and speed-independent asynchronous circuits.

3.9 Performance Analysis

Having proved that the net is *correct*, we can then test it for performance. Performance analysis is essential for *real-time* systems such as safety-critical systems, control systems and computer networks. In fact, for any concurrent process, we would like to know that our solution is optimal. Performance analysis can give us an insight into this problem.

Performance analysis can only be computed on nets extended with time. These, as we have mentioned, are called GSPNs and are available in DaNAMiCS. There are two metrics DaNAMiCS obtains from performance analysis: *Mean Number of Tokens*, and *Transition Throughput*. *Mean Number of Tokens* is the mean number of tokens on each place in a net. This, in a real world system, could indicate how often a resource is utilised. The *Transition Throughput* metric reveals how often transitions in the net are fired. This, in a real-world system, would show how often a sub-process is executed, compared to other sub-process. The next chapter contains more information on the theory behind performance analysis.

3.10 Limitations of Petri nets

For a modeling technique to be useful, one needs to determine the *modeling power* and the *decision power* of the tool. The modeling power defines the set of systems which can be modeled with a Petri net, while the decision power determines the set of characteristics always decidable on the modeled system.

First we will look at what restrictions there are on the modeling power of a Petri net. This can best be illustrated with an example. Dijkstra [8] defined his P and V operations on semaphores to facilitate sychronization and communication in systems consisting of concurrent processes. These respectively decrement and increment the semaphore and are atomic operations which cannot be interrupted by other processes. It cannot be guaranteed that two operations will not be interrupted between them. Additionally a semaphore can never be negative. Thus a P operation must wait until the semaphore is positive before it can continue. It has been shown that all P and V operations can be modeled as Petri nets where each transition has at most two inputs,

P and V operations were proposed as a method for solving all synchronization problems. Questions naturally arose as to there completeness. Patil [17] set out to try to disprove this.

He created a problem which cannot be solved with these P and V operations. The problem was the *cigarette smokers' problem*. This problem consists of four processes: an agent and three smokers. Each smoker repeatedly makes a cigarette and smokes it. To make a cigarette three ingredients are needed: tobacco, paper and matches. Each smoker has a infinite supply of a different one of the ingredients, while the agent has an infinite supply of all three. The agent places two ingredients on the table and the smoker who has the third ingredient can then make and smoke a cigarette, signalling upon completion. The agent then places another two ingredients on the table and the process is repeated.

A semaphore is associated with each ingredient. They are all initially zero. The agent will increment two of the three semaphores with V operations and wait on a $done$ semaphore. The appropriate smoker must decrement the two semaphore with P operations and increment the $done$ semaphore.

Patil proved that no sequence of P and V operations can solve this problem. The solution is a Petri net with at least three inputs on some transitions, which cannot be converted to a Petri net with at most two inputs on a transition without introducing possible deadlock.

This was then followed up by Kosaraju [13] and Agerwala and Flynn [1] where they produced a problem which cannot be solved by P or V operations or by Petri nets.

This produces a real problem in the modeling of concurrent systems. Petri nets are *permissive* by nature. If a transition should be enabled when there are one or two tokens on a place it will also be enabled when there are three tokens on the place. This makes it impossible to model priority systems.

> "More specifically, the limitation on Petri nets modeling is precisely the inability to test for exactly a specific marking in an unbounded place and take action on the outcome of the test. This is commonly stated as an inability to test for a zero marking in a place." [19]

All other extensions do not increase the modeling power of the Petri net. They do however make it easier to model or analyse the systems.

The ability to test for zero has been shown to increase the power of the Petri net model to that of a Turing machine. Thus a Petri net with zero testing can model any system, but almost all analysis problems become undecidable, since they are undecidable for Turing machines. [19] The analysis can however still be decided in many cases. In practice the decidable nets cover most correct systems and many incorrect systems.

3.11 Extensions to the Petri net

Extensions to Petri nets either increase their modeling power, simplify the process of model creation, or simplify the analysis. Petri nets have been extended in DaNAMiCS. These extensions are:

- Add time and immediate transitions to extend the net to a Generalized Stochastic Petri net.
- Add the *zero check* on the place in the form of an *inhibitor* arc.
- Place bounds on the number of tokens on a place at any time.

Petri nets can be extended with time by associating with each transition an exponentially distributed delay. These transitions are then called timed transitions as there is a time delay at each transition. This extends the Petri net to a stochastic Petri net (SPN). This does not alter the correctness analysis as any transition that is enabled could still fire. By allowing the firing rate of transitions to increase to infinity we introduce the concept of *immediate transitions*. These transitions have no delay before firing and always take precedence over timed transitions. This effects the correctness analysis of Petri nets as a different set transitions is enabled for a given marking.

The *zero check* can be added with the introduction of the inhibitor arc. Inhibitor arcs are control arcs. They do not produce or consume tokens as ordinary arcs do. An inhibitor arc from a place to a transition inhibits or prevent a transition from firing if there are any token on the place. The addition of the inhibitor arc increases the modeling power to that of a Turing machine. Unfortunately it does also considerably decrease the decision power.

Placing bounds on the maximum number of tokens allowed on a place at any time does not increase the modeling power as bounded places can be modeled with an ordinary Petri net. They can make the models of systems smaller, more manageable and easier to understand.

3.12 Summary

Petri nets are a useful formalism for describing concurrent systems. They are a graphical and mathematical tool for analysing the correctness and performance of systems. Petri nets can be used to communicate ideas from theoreticians to practicians via a neat, understandable mechanism. Additionally, we can prove, mathematically, that certain desirable properties of our system will hold such as, in the dining philosophers problem, we can prove that each philosopher can at most have one fork in his or her left and one fork is his or her right hand at any time[3].

Petri nets have a static and dynamic behaviour which can be tested for *correctness* and *performance*. This allows us to create a desirable modeling tool. However they do have some limitations on what they can model. Specifically it is impossible to model a *zero-check*. This can be overcome with the introduction of *inhibitor* arcs.

The inhibitor arc increases the modeling power and decreases the decision power. Other additions do not increase the modeling power beyond that of a Petri net with inhibitor arcs. They have been included because they make the process of designing the model easier.

[3]We determine this from P-Invariants

Chapter 4

Markov Theory and DNAmaca

4.1 Markov Theory

Markov theory is briefly discussed in this chapter. It is not a tutorial on Markov theory, but rather aims to provide the background information needed to understand the use of DNAmaca for performance analysis. Markov processes are stochastic processes which exhibit a certain property (these terms are discussed in the sections which follow). Using Markov theory, these processes can be analysed for performance. Generalised Stochastic Petri Nets can be represented as Markov processes, so it is possible to carry out performance analysis by analysing the GSPN's underlying Markov process.

4.1.1 Stochastic Processes

A variable is said to be *random* if we cannot tell for certain what its value will be[4] (for example, the number of packets arriving at a network node). Although its exact value may be unknown, it is often possible to calculate that the random variable will take on a specific value with a certain probability.

A *stochastic process* is defined as a family of random variables $X(t), t \in T$ defined on a given probability space and indexed by parameter t, where t is an element of the index set T[21]. T is a subset of $(-\infty, \infty)$ and is usually thought of as time. Markov processes are stochastic processes for which the Markov property holds. The Markov property will be defined in the next section. The values taken on by the random variable $X(t)$ are called *states*, and the set of all possible states makes up the *state space* of the process.

4.1.2 Markov Processes

It is often possible to represent the behaviour of a system by describing the various states that the system could be in, and by specifying how the system changes state over time[21].

A system is assumed to occupy only one state at any given time, and is said to be *Markovian* if the *Markov property* holds. The Markov property holds if the next state of the system depends only on the current state, and not on the states previously visited, or on the time spent in the current state. A Markov process with a *discrete* state space is also known as a *Markov chain*.

If the time set associated with the Markov chain is discrete, then there is a set of *transition probabilities* associated with the chain that describes the likelihood of moving from one state to another. If the time

is *continuous*, then there is a set of associated *transition rates* which describes the likelihood of changing states[1].

Steady state is achieved if the Markov chain reaches a stage where the transition probabilities (or transition rates) remain constant, even when the time parameter is increased to infinity. A great deal can be deduced from the steady state distribution, for example, if the steady state probability distribution is $(0.1, 0.6, 0.3)$, we can see that the system spends 10% of its time in state one, 60% of its time in state two, and 30% of its time in state three. This is very useful for performance analysis.

4.1.3 GSPNs and Markov Chains

The performance analysis of Generalised Stochastic Petri Nets can be carried out by analysing the underlying Markov process. In order to find the corresponding Markov process, we notice that:

- The *reachability set* of the GSPN is the state space of the Markov chain. Each marking in the set corresponds to a state in the Markov chain.
- The transition rate from state M_i to M_j is given by the sum of the rates of transitions that lead from M_i to M_j.

Once the steady state distribution has been calculated for this Markov chain, performance measures such as probability of being in a subset of markings, mean number of tokens, probability of firing a specific transition and the throughput of a transition can be calculated. Refer to **Stochastic Petri Nets**[4] to see how these are calculated. DaNAMiCS uses DNAmaca to solve for the steady state distribution and calculate the performance measures for the GSPN.

4.2 DNAmaca - A Markov chain analyser

4.2.1 Overview

DNAmaca is a Markov chain analyser written by William Knottenbelt in 1996 as part of his Master of Science degree for the University of Cape Town. DNAmaca takes in a high-level model description of the Markov system, generates the state space, solves for the steady state and produces corresponding performance statistics of the model.

Efficient numerical techniques are used by DNAmaca to *analytically* solve for the steady state distribution. The exact method which is used depends on the size of the state space which is generated - some methods are more efficient at handling small state spaces and others handle large state spaces well.[2].

4.2.2 Specifying model files

Appendix B shows DNAmaca's *interface language specification*, which is the grammar for DNAmaca model files.

[1] See [4] for an introduction to Discrete Time Markov Chains (DTMC) and Continuous Time Markov Chains (CTMC).
[2] Refer to **William Knottenbelt's dissertation for Master of Science**[12] for more detailed information.

It would be tedious to enumerate every state in the Markov chain, so the model file contains a high-level description of the model to be analysed. DNAmaca uses this description to generate the entire state space of the Markov chain that the model represents. The model file also contains the conditions and effects of state these transitions. The initial state of the system must also be specified.

The model file contains the *state description vector* which is a set of attributes that, taken together, specify the state of the system. Each unique value of these components specifies a single state. DNAmaca allows the specification of the state description vector to be specified as a set of the basic C/C++ variable types (such as char, int etc.). For example, if the attributes which uniquely identified a state were the elements *CPU*, *programNumber* and *numCycles*, then a suitable state state description vector could be specified as:

```
\statevector{
  \type{int}{CPU, programNumber, numCycles}
}
```

Each state in this case can be specified as $\{int, int, int\}$, and the state space would be all the unique values that this vector can take on e.g. $\{1, 0, 1\}$, $\{2, 1, 3\}$ etc... An obvious requirement of the model is that the state space is finite. If it has an infinite state space, DNAmaca will continue generating new states until it runs out of resources.

The initial state is specified inside an *initial* entry and contains assignment statements which initialize the state vector e.g.:

```
\initial{
  CPU = 1;
  programNumber = 0;
  numCycles = 1;
}
```

This would set the initial state to $\{1, 0, 1\}$.

State transitions are described in the model file by specifying:

- the transition name
- the *enabling conditions* which involve the elements of the state vector relating to the current state. For example, some condition could be that the **programNumber** is greater than 2.
- the *action* to be executed if the transition is taken. This normally involves the updating of the state vector to the *next* state.
- a *rate* for timed transitions or a *weight* for instantaneous transitions. These rates may be dependent on the current state.

An example is shown below:

```
\transition{process0}{
  \condition{(numCycles > 0) && (programNumber != 0)}
```

```
    \action{
      next->CPU = 1;
      next->programNumber = 0;
      next->numCycles = 100;
    }
    \rate{0.3}
  }
```

This transition will be enabled (but not necessarily taken) if **numCycles** is greater than 0 and **program-Number** is not 0. The action specifies that the next state will be $\{1, 0, 100\}$, and the rate of this transition is 0.3.

4.2.3 Performance measures

There are two types of performance measures that DNAmaca provides: *state measures* and *count measures*.

State measures

State measures are used to calculate the *mean* and *variance* of an *expression* which is defined at every state of the system. An expression is any valid C++ expression whose terms relate to the state description vector. In terms of Petri nets, a state measure would be used to calculate the mean number of tokens on a place. Given n states, the steady state distribution $\pi = (\pi_1, \pi_2, ..., \pi_n)$ and a vector of expression values $v = (v_1, v_2, ..., v_n)$ where v_i is a function of the elements of the state descriptor of state i, the mean of state measure m can be calculated by:

$$E[m] = \sum_{i=1}^{n} \pi_i v_i$$

v_i can be regarded as the ith state's contribution towards the value of the state measure. For example, if we are interested in the mean number of tokens on place p, v_i would be the number of tokens on p in state i.[3]

DNAmaca must be told to calculate a state measure for each state that the measure is required. The state measure entry must contain an expression which is evaluated at every state. A sample state measure entry is:

```
\statemeasure{Mean number of cycles}{
  \estimator{mean variance}
  \expression{numCycles}
}
```

This entry specifies:

- the state measure's label is **Mean number of cycles**.
- the mean and variance of this state measure is required.
- the expression to be evaluated is simply the value of **numCycles**.

[3]Refer to [12] for more detailed information as to how DNAmaca calculates the performance measures.

Count measures

Count measures are used to calculate the mean rate at which events occur e.g. the transition throughput. Given a system with n states, the steady state distribution $\pi = (\pi_1, \pi_2, ..., \pi_n)$, and a function r_i which returns the rate at which the event occurs in state i, the mean of a count measure m is:

$$E[m] = \sum_{i=1}^{n} \pi_i r_i$$

The value of r_i is specified in the count measure entry. For example:

```
\countmeasure{Mean throughput of run}{
  \estimator{mean}
  \precondition{(numCycles > 0)}
  \postcondition{1}
  \transition{run}
}
```

This entry specifies:

- the count measure's label is **Mean throughput of run**
- the mean of this count measure is required.
- the pre-condition is that **numCycles** is greater than 0.
- there is no post-condition (in this case it is always true).

4.2.4 Use of DNAmaca by DaNAMiCS

DaNAMiCS uses DNAmaca for analytical performance analysis. A model file, representing the Petri net being analysed, is created by DaNAMiCS. The required performance measures are specified in the model file. DaNAMiCS sends this file to a server (via a network connection), which starts DNAmaca running on the file. DNAmaca writes the results to a file, which is then processed by the server, and the relevant results are returned to DaNAMiCS. Refer to Chapter 8 for more information on the implementation of the server, and Chapter 10 for more detail on performance analysis.

Chapter 5

Scope and Objectives

5.1 Overview

In this section we discuss the scope and major objectives of the DaNAMiCS project. These were defined early the project and were, of course, refined through the course of the project development.

5.2 Previous Work

The DNA Laboratory at UCT has been developing a set of tools to aid in the modeling of systems. In 1994 a Petri net editor called DNAnet was developed by a team of Honours students. At the time of DNAnet's development there were several Petri net editors available but they all lacked the flexibility required in modeling modern systems and protocols.

Among DNAnet's chief goals was the desire to allow hierarchical modeling with subnets and to allow performance analysis. This meant using GSPNs as the underlying formalism of the tool. DNAnet was intended to be as portable as possible and be able to interact with the DNA Laboratory's other case tools. DNAnet was received well, but users made several suggestions for its improvement, hence the DaNAMicS project.

Another tool related to DaNAMiCS conception is a tool called DNAmaca, which is a Markov chain analyser written by W. Knottenbelt for his Masters thesis. DNAmaca was written to improve on existing analysers and parallel and distributed versions of this system are currently under development.

5.3 Scope of DaNAMiCS

In this vein, DaNAMiCS is one of the first of the DNA Laboratory's new generation of modeling software tools. DaNAMiCS should fit into the existing set of tools, which is why it is important that DaNAMiCS uses DNAmaca for steady state analysis, and that it can analyse models created with DNAnet.

The scope of the DaNAMiCS project thus stems from the existing work, but extends it in many ways. The existing Petri net editing tools lack flexibility, so DaNAMiCS should allow flexible modeling while ensuring that the analysis of the extended models is comprehensive.

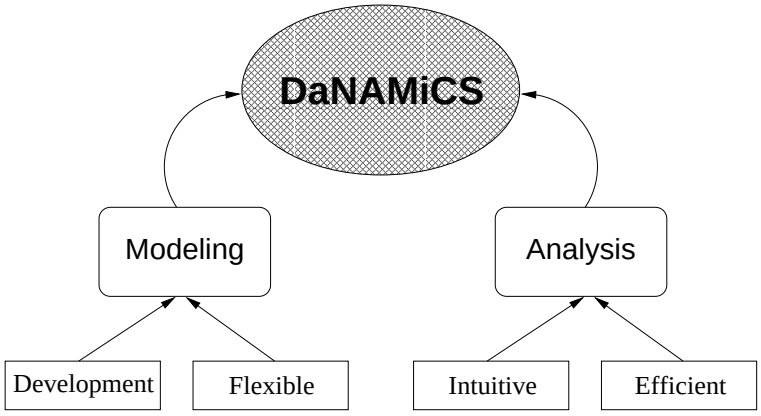


Figure 5.1: Scope of the DaNAMiCS Project

DaNAMiCS will allow the modeling of GSPNs for performance and correctness analysis. It should also allow for the modeling of a subset of the Coloured Petri net class. It is important that DaNAMiCS be applicable to a broad spectrum of the concurrency field, since we would like one package where users from many disciplines can formulate their ideas and communicate them to one another.

In short, DaNAMiCS is a Petri net editor with the ability to analyse a broad class of Petri nets in terms of correctness and performance. It allows users to model any system and analyse these systems suitably. Finally, it facilitates the analysis of the results in an intuitive fashion.

5.4 Major Objectives

The major objectives of the DaNAMiCS project are summarised in this section.

5.4.1 System Modeling

We would like to design a package that provides the flexibility to model widely differing systems. For this reason, we have used the underlying formalism of GSPNs which allow both correctness and performance modeling. We should be able to use hierarchical modeling techniques and the models should be further simplified with coloured tokens.

The above features really only increase the usability of the modeling package and ease of modeling. Modeling power would be vastly increased with the addition of control or inhibitor arcs to the nets. This feature would increase our modeling power to that of Turing machines. This addition is invaluable because it allows any possible system or protocol to be modeled.

5.4.2 System Analysis

With our increased modeling power we should not lose the ability to analyse our system models for correctness and performance. We will see later that as the power of the modeling tool increases, the ability to analyse the models decreases. However, the system should provide accurate results for the widest possible set of systems, and alert the user if accurate results are not achievable.

The performance analysis is unaffected by the extensions to the Petri nets and thus can be carried out as normal, either by an internal simulator or by an external steady state analyser.

5.4.3 User Interface

The user interface should facilitate the quick and easy creation of the Petri nets. Convenience features not present in DNAnet, such as printing and viewing results will be included.

Users should be able to design their models as efficiently as possible and repetitive tasks should be minimal. If they are present, they should be simplified as much as possible. In addition to this, the users should be able to analyse their result in an efficient and simple manner. Facilities should be provided for the results to be displayed and analysed as quickly as possible. Since the results are essential to any modeled system, it is important that the results not only be easily accessed, but that they are also presentable to audiences which will decide on the model. The results should thus be visually striking and allow important features to be highlighted.

5.4.4 Platform Independence

The DaNAMiCS project should be platform-independent. Modeling tools are used by many different professionals in many different fields. These professionals will be using different systems. DaNAMiCS is designed to allow flexible modeling, thus it should also be flexible in the environments in which it can be used.

5.4.5 Efficiency

DaNAMiCS will be required to perform intense computations, thus it should be as efficient as possible. It is important that the interface has an adequate response time, and particularly, that the analysis completes timeously.

5.4.6 Extensibility

There are many different classes of Petri nets, and it is important that DaNAMiCS is extensible for future developers to enhance it with the new classes. This implies the use of modular coding techniques that are well documented. File formats should be conventional and adequate documentation of this should be provided.

Chapter 6

Program Specification

6.1 Specification of the Petri net

DaNAMiCS should include the ability to model Generalised Stochastic Petri Nets (GSPNs) and elementary Coloured Petri Nets (CPNs). DaNAMiCS is a GSPN modeling tool, but will include the ability to model with coloured tokens. These two Petri net classes have been chosen as they allow flexibility, and reduce the complexity of the modeled net. DaNAMiCS will include support for inhibitor arcs to aid the creation of a system model. A description of the objects that should be included in the Petri net data structure follow. Refer to Figure 6.1 for a pictorial representation.

The Petri net object should maintain a store of the places and transitions that are contained in it. It should store the subnets that it contains. Each place in the Petri net should contain:

- The screen coordinates
- The unique place identification
- The number of tokens on that place and the capacity of that place
- Any arcs attached to the place

Each transition in the Petri net should contain:

- The screen coordinates
- The orientation of the transition
- The unique transition identification
- The arcs, incoming and outgoing from this transition
- The weight of the transition, if it is an immediate transition
- The rate of the transition, if it is a timed transition

Each subnet should contain:

- The unique subnet identification
- A Petri net
- Lists of those places and transitions that can be accessed from outside the subnet, so-called *public* places or transitions. These are called *interaction points*.

The arcs between each place and transition should be easily accessible and contain the following information:

- The screen coordinates and relevant spline information
- The arc weight
- Whether the arc is a *normal* or *inhibitor* arc.
- The place and transition which the arc connects.

The structure should allow easy access to the basic logical operations that will be performed on the net. It should be easy and efficient for the following operations to be performed on the net:

- Detect which transitions in a certain marking are enabled,
- Fire an enabled transition from a particular marking,
- Set the places to a particular marking,
- *Flatten* the net by removing the subnet hierarchy,
- Construct the incidence matrix from the flattened net,
- Create files for storage and interface to other programs such as DNAmaca ,
- Graphical construction and manipulation of the net.

6.1.1 Coloured Petri Nets

DaNAMiCS will support the use of coloured tokens in Petri nets. The ability to edit, animate, and perform analysis on the coloured tokens shall be provided. Either the whole net will use CPNs or GSPNs. In CPNs at least three colours shall be supported After this point, any analysis will first involve the *unfolding* of the CPN.

The transitions need to contain user controllable *modes* or different ways in which the transition can fire. These modes need to be stored and displayed in a concise method.

There is a standard algorithm for unfolding a CPN. The unfolded net will be used in performance and correctness analysis. No clutter control shall be implemented if the unfolded net is displayed to the user.

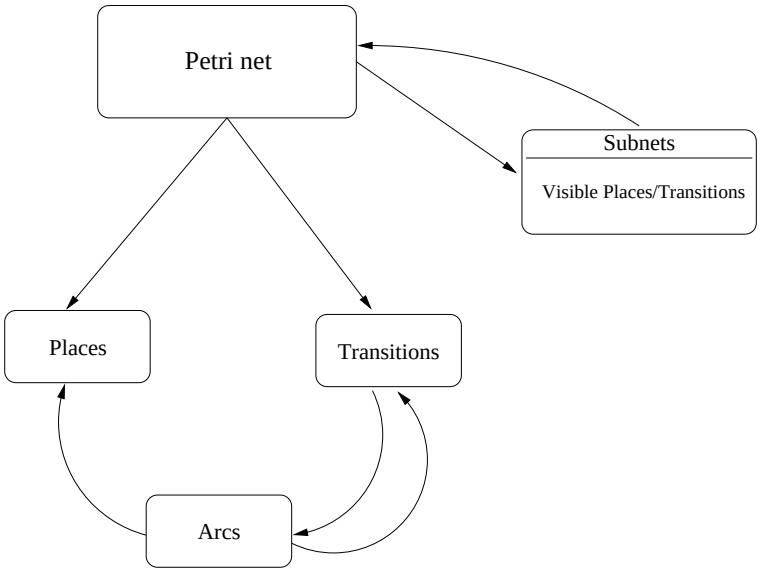


Figure 6.1: The Petri Specification Data-Structure

6.1.2 Inhibitor Arcs

Inhibitor arcs can be implemented by having a reserved weight on an arc. The inhibitor arc should appear different to the user. It must still have the generally accepted form of an arc from a place to a transition with a circle on the transition instead of an arrow. The animation functions available in DaNAMiCS should account for the presence of inhibitor arcs, and the functions for firing and detecting enabled transitions should be adjusted accordingly.

6.1.3 Limited Places

These are places where there is a maximum number of tokens of a place. They are a short hand for the Petri net that bounds the place. They are a modeling aid to the user.

6.2 Correctness

The following correctness functionality should be present in DaNAMiCS.

6.2.1 Invariant Analysis

The choice of the Petri net structure should facilitate invariant analysis. To the Petri net object, we add an incidence matrix. The incidence matrix is essential to invariant analysis and it is important that, not only should its structure enable efficient analysis, but also that its construction should be uncomplicated.

Once the incidence matrix has been generated, the production of P- and T-invariants is fairly mechanical. For P-invariants, we must be able to transpose the matrix efficiently. As described in the chapter on correctness analysis, the algorithm for calculating the invariants is well-understood, and must be implemented as efficiently as possible.

Analysis of the results from invariant analysis should be displayed in such a way as to allow intelligent conclusions to be drawn about the model.

6.2.2 Coverability Graph

The algorithm for the construction of the coverability graph must be examined. It will need to be updated to deal with the added complexity of inhibitor arcs and places which have capacities. The implemented algorithm must be correct and must run optimally.

There are standard algorithms for finding final, strongly connected components which indicate liveness and show the existence of home states. The most efficient of these algorithms must be chosen.

A Marking class, and a set of Graph classes, containing markings, should be produced. The marking class must simply contain the number of tokens on each place. The graph should be stored as an adjacency list to facilitate the ease of access and allow the construction and depth-first search algorithms to be performed efficiently.

6.2.3 Simulation

Simulation of the net will follow standard algorithms. The basic format is to start from the initial marking and calculate which transition is the most likely to fire. If the fired transition is immediate then there is no addition to the sojourn time. If it is a timed transition, then the probable exponentially distributed sojourn time in the current marking is calculated. The next marking in the trace is then treated in precisely the same manner.

The simulator should be able to return as many meaningful metrics as possible for interpretation.

6.2.4 Performance

DNAmaca, a steady state analyser written by William Knottenbelt, will be used to calculate the performance metrics. The link between DaNAMiCS and DNAmaca should be implemented using client/server architecture. This will enable DNAmaca to be executed on a fast super-computer. The analysis of Markov Chains involved in steady state analysis is memory and CPU intensive. Thus this method will allow larger GSPNs to be analysed.

The input for DNAmaca takes the form of model files. These model files contain a list of all of the places, and transitions. For each transition, they contain a condition, an action that occurs when that transition fires, and its weight. They also specify a list of the performance measures required.

For a Petri Net with about 1000 places and 1600 transitions these files are approximately 350 Kbytes in size. If this would cause too large of a delay in the transfer of these files, they can be compressed. It would be a possible to compress these files to at least a tenth of their original size. A simple daemon on the server machine should be able to decompress these files and pass them to DNAmaca.

The performance metric calculated should be returned to DaNAMiCS and presented to the user.

6.3 The User Interface

The DaNAMiCS GUI will provide tools for the creation and editing of the Petri nets. It will serve as a front-end for the analytical functions. A convenient way of viewing the analysis results will be provided by the GUI.

The following GUI options will be provided:

6.3.1 Drawing Objects

- *Create Place* : A place will be created wherever the user indicates.
- *Create Transition* : A transition will be created wherever the user indicates. A timed transition or an immediate transition can be created.
- *Create Arc* : An arc can be created between a place and a transition. With respect to coloured tokens, the default will be that the arc transfers any colour. The user can edit the attributes of the arc in order to specify the weighting of the specific colours.
- *Create Subnet* : This will create a box which represents a subnet. The specific subnet that it represents will be indicated by the user at creation time, when the user will be asked for the subnet filename from a dialogue box.

6.3.2 Manipulating Objects

- *Attributes* : An object's attributes can be modified. Different types of objects have different attributes eg. a transition has a name and a rate while an arc has a weight for each colour.
- *Move* : Allows the object to be moved.
- *Delete* : Deletes the object.
- *Rotate* : Rotates the object.
- *Toggle Type* : Toggles transition types between *immediate* and *timed*.
- *Straighten Edge* : Straightens an arc which has been moved by the user.

6.3.3 Group Selection of Objects

- *Deselect All* : Deselects all.
- *Add to Selection* : Adds objects to the selection.
- *Remove from Selection* : Removes objects from the selection.

6.3.4 Group Manipulation

- *Copy* : Make a copy of the selected group of objects.
- *Cut* : The current selection is cut to the clip-board.
- *Paste* : The clip-board contents are pasted to DaNAMiCS.
- *Delete* : Deletes the selected group of objects.
- *Move* : Moves the selected group of objects.

6.3.5 Zooming

- *In* : Zooms in by a set amount.
- *Out* : Zooms out by a set amount.
- *Zoom into Subnet* : Loads the subnet.
- *Zoom out of Subnet* : Returns to the subnet's parent.

6.3.6 File

- *New* : Creates a blank Petri net.
- *Open* : Opens an existing Petri net.
- *Save* : Saves the current Petri net.
- *Export* : Exports the Petri net to a different format such as postscript.
- *Print* : Print the Petri net.

6.3.7 Analysis

These options initiate the various analysis functions. If the Petri net uses coloured tokens, then it will automatically be unfolded before the analysis takes place, and the analysis will be done on the unfolded Petri net.

Correctness Analysis using Invariant Analysis

DaNAMiCS will be able to calculate the incidence matrix. From the incidence matrix it will be able to determine the P- and T-invariants. These can be used to detect boundedness and the absence of deadlock.

Correctness Analysis using Coverability Analysis

The user will be able to select options in DaNAMiCS by specifying which areas of the functional analysis need to be explored. The coverability graph will be generated and tested for correctness. For example, selecting the deadlock option will test the Petri net for deadlock.

Performance Analysis using Simulation

When the user selects this option, performance analysis will be carried out on the Petri net via simulation. A dialogue box will allow the user to set the following options:

- *Number of Cycles* : The user can enter in the number of cycles to run the simulation for.
- *Transition Throughput*
- *Mean number of Tokens per Place*

Performance Analysis using DNAMACA

When the user selects this option, performance analysis will be carried out on the Petri net by DNAmaca. A dialogue box will allow the user to enter in the name/location of the server running DNAmaca. The Petri net will be converted into the appropriate format and (possibly) sent across the network to the DNAmaca server for analysis. The results will be returned via the network to DaNAMiCS. All this will occur transparently with respect to the user.

6.3.8 Animation

Animation can be thought of as a user-driven simulation. Essentially, the user decides which of the enabled transitions to fire, and the new state is shown. The option here are:

- *Start* : Start the animation.
- *Stop* : Stops the animation.
- *Return to previous marking* : The Petri net is restored to the state it was in before the last transition was fired (like an *undo* button for transition firings). This can be done multiple times.

6.3.9 Results Viewer

The Results Viewer will be invoked whenever any analysis is carried out. The Results Viewer should present the information to the user in a concise manner. All possible information should be gleaned for the analysis with no calculated values being suppressed.

Data filters should be applied to the results only on the users' request. These filters will not reduce the data produced, but rather display a subset of it to reduce cognitive overload.

Wherever possible and applicable, the data should be displayed in a graphical format. So, unbounded places should be highlighted, and deadlock firing sequences should be shown.

There should be options in the results viewer to save the results to a file and also to repeat the analysis on the current net. The saved results should be able to be loaded into the results viewer for consistency. The grammar to the results file is shown in the appendix.

Chapter 7

Object Structure

7.1 Design Approach

Petri nets are mathematical structures. Certain implementations of Petri net editors have used the mathematical representation of a Petri net to store and manipulate them digitally. While this results in a neat and logical mathematical model, it is not necessarily the best way in which to represent Petri nets using computers.

The formal mathematical structure of Petri nets does not contain any display details. While this information does not change the mathematical properties of a Petri net, they are necessary in an editor or viewer.

Currently, the best, and most logical way to express Petri nets in a computer environment, is via an object model. There are several reasons for this. Mathematical models infer no redundancy. While this is conservative of memory and storage space, it might also make calculating certain other attributes of the Petri net very complicated and inefficient. Objects also allow extra information to be recorded about the Petri net.

For example, if we simply store the incidence matrix and the initial marking, we completely define the net mathematically. However, calculating the coverability graph from this would be very difficult and time consuming. We are interested in creating the fastest possible system for all types of nets and analysis.

In order to produce such a system, it is not feasible to create a minimal mathematical representation. Instead, we use the object oriented approach to the model and reduce the Petri net to a base form that can then be used equally in all types of analysis. In addition, this will enable the fast graphical rendering of the Petri nets.

7.2 Overview

Petri nets lend themselves very well to the object-oriented methodology, and one can think of Petri nets of containing three simple objects:

- places,
- transitions,
- and arcs.

Places can be thought of as *having* arcs and transitions have *producing* and *consuming* arcs. From this simple foundation we can extend the model to include subnets and coloured tokens. It is the aim of this chapter to describe the objects used in the DaNAMiCS system in detail. Firstly though, let us take a high-level look at the DaNAMiCS object structure.

7.3 High-level Petri-net Object Structure

The Petri net object is an instance of a Petri net class called *Petri*. Petri is an aggregation of places, transitions, and subnets. A subnet merely contains another Petri, thus providing the hierarchical structure required by the system.

We used Subnets as different entities from Petris because they differ technically. For example, when drawing a Petri, we draw all its associated places, transitions and arcs, while when drawing a subnet, we simply draw a large box. In addition, Petris do not have names or locations.

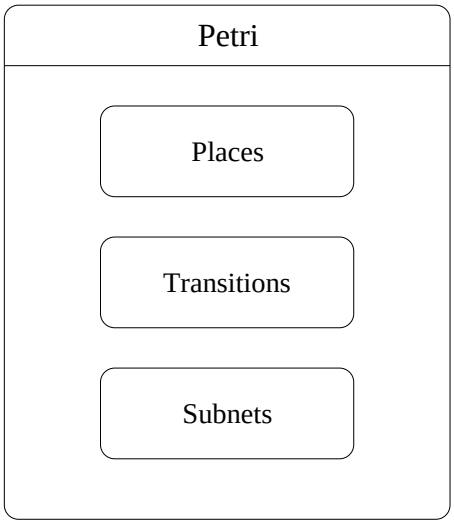


Figure 7.1: High-level Petri Object Structure

Figure 7.1 shows the simple object structure. No graphical or drawing details are included here, since they will be discussed in the section *Graphical Objects* on page 40. We will now provide a high-level view of the data-structure used to store the Petri nets.

Figure 7.2 shows the basic interactions of the core objects in the system. The Petri object stores the Places, Transitions, and Subnets as linked-lists of handles to the objects. Every Transition contains two linked-lists, one of handles to its incoming arcs and one of handles to its outgoing arcs. Every Place contains one list of handles to all arcs that are connected to it. This is a convenience for functions such as *delete* and finding the places based on a mouse-click.

The linked-lists of Places, Transitions, and Subnets are generic lists of type *PetriUnitList*. A PetriUnitList is a list of *PetriUnits*. Place, Transition, Subnet, and Arc objects all extend PetriUnit. The PetriUnit object contains attributes common to all of the above objects, such as a name and screen coordinates. Using inheritance in this way, we are able to write generic functions to operate on lists of all types of objects. This greatly reduced implementation time and the complexity of the code. This inheritance model is shown

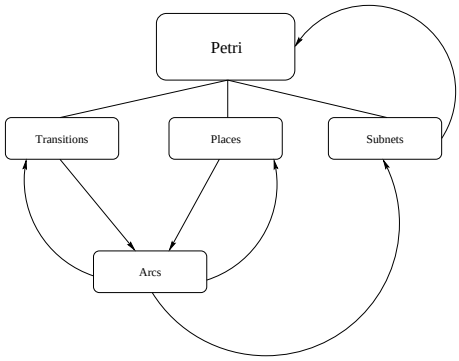


Figure 7.2: Petri net Data Structure

in Figure 7.3. The Figure also shows *Comment* and *Highlight* objects which will be described later in Section 7.7. They add no functionality to the Petri net, they are merely a tool for the users' convenience, but are stored as PetriUnits in a PetriUnitList.

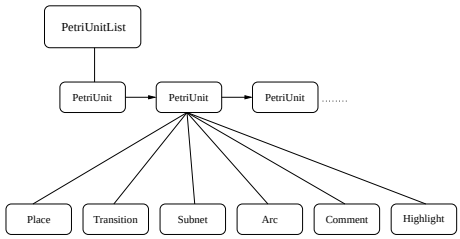


Figure 7.3: PetriUnit Inheritance

The Arc object in Figure 7.4 forms the bridge between places and transitions and also extends PetriUnit. The Arc object contains many convenience handles which may be redundant, but which decrease object access time from the canvas. They are used in some analysis functions for the same purpose. The Arc contains a handle to its associated place and transition. In addition to this we also store the PetriUnit it originates *from* and the PetriUnit *to* which it leads. These additional handles will either be replicas of the handles to the place or transition, or, if the arc spans a subnet, they will point to the subnet that the arc originates from or into which it leads. This increases the efficiency of the drawing functions.

We will now look at each of the objects in the system in greater detail and further describe the data structure implemented in DaNAMiCS.

7.4 Petri net Objects

In this section we provide detailed descriptions of the objects used by DaNAMiCS. The objects described here are those used for the Petri net data structure. Later, we describe the objects that are used in the interface and analysis sections of DaNAMiCS.

7.4.1 PetriUnit

As has been mentioned, all the core objects in DaNAMiCS inherit from the class PetriUnit. PetriUnit is seen as the basic, smallest unit of functionality in a Petri net.

Each PetriUnit contains a name and a set of screen-coordinates. The screen-coordinates also include information about the relative location of the unit's label which can be moved. In addition, we include a *unit type* attribute for the run-time type identification of the Petri Units. This has been implemented as it will be faster than using Java's RTTI[1] *instanceof*.

Each PetriUnit is ascribed a parent, which is the subnet of which this PetriUnit is a member. If the parent is **null** then this means that the PetriUnit in question is at the highest possible level and is not contained inside a subnet. We will now describe the objects that extend PetriUnit.

7.4.2 Place

A Place object extends the PetriUnit and adds the following attributes:

- the initial number of tokens,
- the current number of tokens,
- the capacity of that place, or the maximum number of tokens on that place,
- and the list of arcs connected to that place.

Place has two constructors, one which is used from graphical input, and the other, used when data is read in from a file. Place has limited functionality and the only method of real interest is the *delete* function, which must not only remove the selected place from the net, it must also remove its connected arcs.

7.4.3 Transition

A Transition object adds the following attributes to PetriUnit:

- the type of the transition, ie: timed or immediate,
- the firing rate of the transition,
- a list of incoming arcs,
- a list of outgoing arcs,
- a boolean indicating whether or not the transition is enabled,
- the orientation of the transition, ie: vertical or horizontal,
- a boolean indicating whether or not the net is being animated, which is used in the drawing functions to draw enabled transitions differently.

Transition also has two constructors that are used the same way that Places' constructors are used. Transition contains methods to add incoming and outgoing arcs. Transition contains methods such as *fire* and *setEnabled* which provide the the required functionality.

[1] Run-Time Type Identification

7.4.4 Subnet

Subnet simply contains the name of the file that this subnet was produced from, and a handle to the Petri object that is contained in this subnet. Subnet contains methods to *flatten* the net for analysis purposes. Since subnets in no way enhance the functional capabilities of Petri nets, they are simply discarded during any analysis. This is achieved by means of recursive functions to extract all the places and transitions into a single list with all the units' associated arcs correctly connected.

7.4.5 Arc

An Arc extends PetriUnit with the following attributes:

- the weight of this arc,
- the type of this arc, ie: splined or straight,
- the graphical handles on this arc,
- the place and transition that this arc connects,
- the PetriUnit this arc points to which can be a place, transition, or subnet,
- the PetriUnit this arc originates from which can be a place, transition, or subnet,

The arc structure is best described in Figure 7.4. As one can see in the diagram, the arc points directly to the place and transition it connects, no matter what subnet it is in. This is useful for analysis purposes, but it also points to the PetriUnit that it originates from and connects to; this is useful in the graphical construction of the nets. The Arc object is used in checking whether transitions are enabled and facilitates the firing of the enabled transitions. These functions will be describe in detail in the section on algorithms.

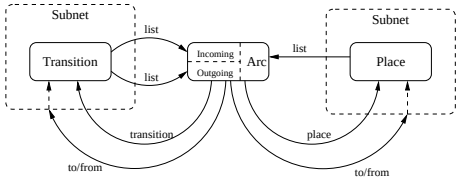


Figure 7.4: The arc structure

7.5 Coloured Petri net Objects

Inheritance is used to incorporate the Coloured Petri net objects into the existing DaNAMiCS object structure. The first new object for the use of Coloured Petri nets (CPNs) is the *CPetri* object. It extends Petri with two basic additions. Firstly, it replaces a method in Petri to detect enabled transitions because the rules for a transition being enabled are different. Secondly, it adds a new method *unfold* to convert the CPN into a normal Petri net. This method will be described in detail in the chapter on Coloured Petri nets.

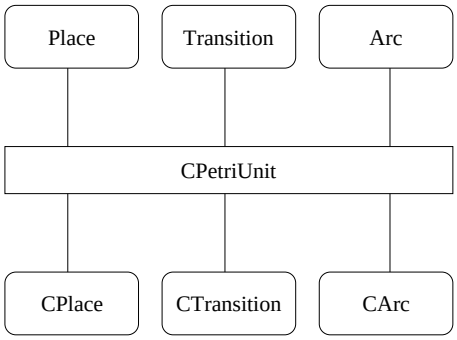


Figure 7.5: Coloured Object inheritance

Figure 7.5 shows how the new coloured petri net objects inherit from the existing petri net structure. *CPetri-Unit* is an interface which describes an abstract method *unfold*. *CPlace*, the coloured place object, *CTransition* the coloured transition object, and *CArc* the coloured arc object all extend their respective normal Petri net components via the interface CPetriUnit. We will now describe, in more detail, the individual sub-classes for the CPN.

7.5.1 Coloured Place

CPlace adds three new arrays to the existing Place object. The length of the array corresponds the number of allowed colours. The three arrays define the current number of tokens of each colour in the place, the initial number of tokens in each place, and the limits or capacities on each of the number of coloured tokens in a place.

7.5.2 Coloured Transition

CTransition extends the existing Transition object with an array of *Modes* which define the respective modes in which a transition can fire. These will be described in detail in the chapter on coloured tokens.

7.5.3 Coloured Arc

CArc includes a two-dimensional array to specify the combination of colours of each type of token that is consumed or produced when a transition fires. We refer the reader to the chapter on coloured tokens for a full description of the functionality of this class.

7.6 Analysis Objects

All analysis functions first *flatten* the net to decrease the time to access places or transitions in subnets. These recursive flattening functions are found in the Subnet class itself, thus isolating the code and promoting reuse. In this section we will describe the objects used in the various types of analysis.

7.6.1 Invariants

The invariant analysis object performs all its computations on a simple two-dimensional array. Since allocating and re-allocating memory in Java is slow, we re-use the array by simply overwriting the existing array elements and keeping track of the size of the array.

The constructor intialises the incidence matrix, which is an $n \times m$ matrix for a net with n places and m transitions. To calculate P-invariants we first transpose the incidence matrix producing an $m \times n$ matrix, to calculate T-invariants, the incidence matrix is left as is for the rest of the algorithm.

The incidence matrix is then extended by adding a square $n \times n$ matrix to produce a $m + n \times n$ extended matrix which is then reduced. The added square matrix is initialised to the identity matrix.

7.6.2 Coverability

Coverability analysis aims to produce a coverability graph. There are many ways to store graph structures digitally, and we chose to use an adjacency list for efficiency. Our adjacency list is modified from the usual structure as it must facilitate the following actions:

- elements should be easily inserted,
- it should be possible to search through all the existing elements and,
- it should be possible to trace backwards through the existing arcs.

For this purpose DaNAMiCS implements a hash table instead of a linked list. Each marking hashes to a unique value. This facilities the insertion of new markings. Threading through the elements is a linked list which allows one to search through all the elements efficiently. Finally, each marking has a *father* marking which is the marking which enabled the transitions to fire into the current marking. Thus, the data structure implemented, and illustrated in Figure 7.6, meets all the requirements set. For a full description of the algorithm implemented, refer to the section on coverability analysis.

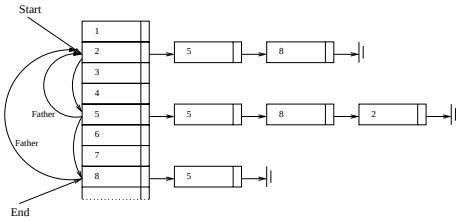


Figure 7.6: The coverability graph data structure

Coverability analysis makes use of a *Marking* class. This class contains a simple array whose length is equivalent to the number of places in the net. Figure 7.7 shows the relationships between each marking. This marking class was made available to all other classes as it is also used in simulation.

7.6.3 Simulation

The Simulation class, *Simul*, is very simple and very efficient. It contains arrays to store the values we desire to obtain. An array whose length is equal to the number of transitions stores the values of transition

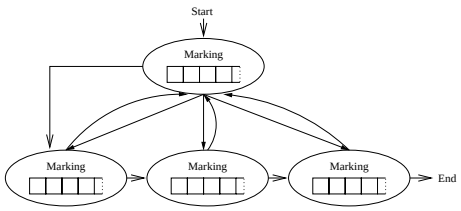


Figure 7.7: A marking, and its associations

throughput and an array whose length is equal to the number of places stores the mean number of tokens for each place. This allows $O(1)$ access to all the elements.

7.6.4 Steady State

The steady state analysis functions make use of the class *Steady*. The Steady object flattens the existing Petri net and writes it to *model* file format. This file is then transferred across the network via the object *DNAmacaClient* to the server which runs DNAmaca . The results are returned to the DNAmacaClient which notifies the Steady object through a call-back method. The Steady object then interprets the results. The Results Viewer, described later, is automatically invoked with the steady state results focused.

7.7 Graphical Objects

7.7.1 Editing

The basic drawing object is the *PetriCanvas*. This object contains a Petri which is to be drawn. Each place, transition, subnet, and arc in the Petri net contains a *paint* method which is called to draw that object onto the PetriCanvas. The PetriCanvas extends predefined classes in the Java Swing libraries.

Associated with the PetriCanvas is a *Graphics* object. This object is passed to the Petri object which places the Graphics object into a *Draw* object along with the current scale and other formatting information. The Draw object stores the Graphics object and maintains information on how the overall net is to be displayed.

The Draw object is passed to the Place, Transition, Subnet, and Arc objects which draw themselves onto Graphics. In the process of drawing the net, we ensure that arcs that connect places and transitions in different subnets are drawn such that the arc connects to the subnet and not the actual place or transition.

CPetriCanvas is the Coloured net PetriCanvas. This object simply allows for different types of actions to be performed on the canvas and paints the net such that the different colours in the places can be distinguished.

7.7.2 Results Viewer

The Results object is a large object that controls the interface to the results generated by the various types of analysis. The most interesting component of the results viewer is the coverability results pane which displays the net and animates the possibilities for unboundedness and deadlock. This is achieved by inheriting from PetriCanvas into an object called *ResultsPetriCanvas* which overrides the *paint* method. The canvas displayed in the results viewer is threaded to ensure that the interface remains functional during animation.

Chapter 8

Implementation Approach

8.1 Programming Platform

DNAnet was implemented in C++ and made use of the zApp libraries for the interface. This allowed it to be ported to several platforms, but was still tied to those that support zApp.

One of the original specifications of the DaNAMiCS project was to implement it in Java to ensure portability. Java runs on virtual machines which are written for most platforms. The benefit of using Java in this way is that the Java class files do not have to be recompiled when the platform is changed. This aids distribution and makes system maintenance much simpler.

However, the run-time performance of C++ is vastly superior to that of Java. Java consumes more memory and is much slower than C++. This is because of the overhead associated with Java classes, and that Java bytecode must be interpreted by the Java virtual machine before the instructions can be executed on the CPU. Despite this, Java is a suitable language for development because of its excellent safety features and strict language constraints.

During the course of the implementation of DaNAMiCS, we noticed several discrepancies with the use of Java. Certain functions written on Windows NT and '95 platforms did not behave as expected on the Sun systems. This proved very problematic during development and thus we needed to ensure that all code generated would have the same function across platforms. These discrepancies were limited,however, to the graphical user interface, and this is mainly due to the fact that DaNAMiCS made use of the Java Swing libraries, which are still in early release. Later versions of these libraries became available very late in the development phase and it was found that DaNAMiCS integrated seamlessly with new libraries.

The major benefit of using Java is that when later and superior versions of the Java virtual machine become available, the DaNAMiCS system will be able to make use of these with little or no maintenance. This enhances the robustness of DaNAMiCS and ensures that it will be a viable application for use in future systems.

8.2 Performance Considerations

We have mentioned that Java is slower than C++. The type of analysis we perform with DaNAMiCS is fairly computationally heavy, and thus optimising the analysis algorithms was essential. Wherever possible,

memory was reused to save the overhead of declaring new memory space. Additionally, we took advantage of the Java keyword **final** to bind attributes at compile time.

Despite all this, DaNAMiCS' analysis will never be as fast as that performed in DNAnet with C++ and so we had to ensure that DaNAMiCS would still be usable in smaller systems. For example, in the drawing functions, there are routines to determine the smallest possible portion of the net which must be drawn when the user makes a change to the net.

8.3 Extensibility

Another important feature of DaNAMiCS is its extensibility to future work. There are many Petri net classes which could be implemented to serve users' specific needs. It was important then that DaNAMiCS was designed in such a way as to allow easy extensibility.

To this end, DaNAMiCS used modular object-oriented design and attempted to make the modules as independent as possible. All the analysis modules are separable components which can be removed or modified with ease. Additionally, the underlying Petri net structure is easily extendable, as was demonstrated with the inherited inclusion of coloured token objects. In this way, in should be easy to add new net classes to the existing DaNAMiCS system.

8.4 Client/Server interaction with DNAmaca

DaNAMiCS communicates with DNAmaca in a client/server fashion. The DaNAMiCS client requests the DNAmaca server to perform analysis on the specified model. Results are then returned to the client for viewing.

This approach was taken for a number of reasons:

- DNAmaca is very resource intensive. Not only is it computationally heavy, but it also requires a large amount of memory and disk space to execute. DaNAMiCS on the other hand requires relatively few resources.
- C++ was used to code DNAmaca and so it can only run on platforms that is has been specifically compiled for.
- The DNAmaca server could be a powerful computer which multiple users can share. Having multiple users connect to the same the server would not cause a performance bottle-neck, since the performance analysis by DNAmaca would happen infrequently compared to the editing of the model by DaNAMiCS.
- Shipping the performance analysis functions to a remote server would enable the client to continue working while waiting for the results.

8.4.1 Server Implementation

DNAmaca is actually a single-user, stand-alone program. It accepts a file name as a command line parameter, and it writes the results to file. To enable users to send analysis requests over a network, a **Server** class was created, which acts as a wrapper around DNAmaca.

Original size in bytes	Compressed Size in bytes	Compressed size vs. original size
243 602	17 874	7.34 %
74 096	7 971	10.76 %
35 285	5 214	14.78 %
1 938	288	14.86 %
2 233	482	21.56 %

Table 8.1: Table showing packet sizes before and after compression

An object of the **Server** class starts up and listens for connections on a specified port. When an incoming request is received, a child process is spawned to handle the request. The child unpacks and uncompresses the model file from the request and writes it to disk. DNAmaca is then run on that file. The results are parsed from the DNAmaca output file, compressed and sent back to the client. In this way, the server can handle multiple requests simultaneously.

8.4.2 Client implementation

The requirements of the client are fairly simple. It has to package requests for the server, send the request over a network, and receive the results. DaNAMiCS obviously required a multi-threaded implementation, else the client would be idle and unusable while waiting for the server's response. This would be unacceptable, since the performance analysis could take a considerable amount of time. The implementation of a multi-threaded client was a simple task using JAVA's multi-threading capabilities.

8.4.3 Use of compression

The model files used by DNAmaca are plain text files, as are the results files. These files can easily become large for complicated Petri nets. They have to be sent across the network between the client and server, and so the large files could take up a lot of bandwidth. However, text files are good candidates for compression, and it was decided that DaNAMiCS would use JAVA's **GZipStream** class to perform compression on the model files and the corresponding performance results files.

Table 8.1 shows some sizes of the model files before and after compression. Files which would normally take multiple packets to send can easily fit into a single packet after compression. As can be seen from the results, compression was certainly worth implementing.

Chapter 9

Correctness Analysis

9.1 Overview

The analysis of Petri nets seeks to verify that certain properties of a given net are satisfied. The main aim of correctness analysis to to prove that a Petri net is *live* and *bounded*.

Petri nets model real-world systems. Typical models include business systems, safety critical systems, network protocols, and control systems. In all of these, it is vital that the system is correct. That is, the system must never be allowed to become deadlocked – which means a total halt in execution. Or, alternatively, the system must never be caught in a livelock – which means that certain areas of the system will execute endlessly, never relinquishing control to other processes. Additionally, in computer systems, we must ensure that there are no memory leaks, and that no buffers can overflow. This is related to the *boundedness* of a Petri net. Modeling these systems with Petri nets, and then analysing the model, can provide us with these results.

There are three methods in which to conduct correctness analysis. We can analyse Petri nets by means of *invariants*, by *reachability graph exploration*, or by means of reduction techniques. The last of these is only applicable to certain subclasses of Petri nets, and thus will not be used here.

Reachability graph exploration is extended to Coverability graph exploration and, since the algorithm for this is exhaustive, it can prove absolutely that the correctness properties of a net are conserved. However, because it is exhaustive, it is also the slowest of the three analysis techniques. Finally, invariant analysis is possibly the fastest analysis technique, however, it cannot yield absolute results on the correctness of all nets and thus its use is limited.

It is the goal of this chapter to describe the mathematical background to invariant and coverability analysis. The algorithms for implementing these analyses will be shown and explained and the method by which they were implemented described. The reader is refereed to Chapter 4 for an introduction to the theory of correctness analysis in Petri nets.

9.2 Invariant Analysis

Invariant analysis involves the study of matrix equations which govern the dynamic behaviour of systems modeled by Petri nets. The solvability of these equations is somewhat limited however, due to the non-deterministic nature of Petri nets, and because the solutions must be non-negative.

Invariant analysis provides solutions which are valid for all initial markings of a given net. Thus, the invariant solutions are not marking dependent as we will see is the case with coverability analysis.

There are two types of invariants which are useful to Petri nets:

- **P-Invariants**: Place-Invariants, also known as S-Invariants, specify a sum of tokens over a number of places that remains constant in any marking in a net.
- **T-Invariants**: Transition-Invariants specify firing vectors whose combined effect leave the marking of the net unchanged.

P-Invariants are useful in such problems as the bounded Producer-Consumer problem where we wish to show that the the total buffer size remain constant. T-Invariants can be useful in showing which sequences of actions have no side-effects.

9.2.1 Invariant Theory

In this section we present a brief overview of the mathematical theory behind Invariant analysis. Experienced readers may wish to skip to the next section where the algorithm for automatically determining invariants is described.

Before the theory of invariants is described, we will define some constructs used by the invariants.

Incidence Matrix

The incidence matrix is a mathematical view of a Petri net. It depends solely on the Petri net structure and not on the initial marking or current state of the net.

Definition 10.1: Incidence Matrix for a Petri net $PN = (P, T, I^-, I^+, M_0)$

- *The Backward Incidence Matrix* : $C^- = (c_{ij}^-) \in \mathbb{N}_0^{n \times m}$ where $c_{ij}^- = I^-(p_i, t_j), \forall p_i \in P, t_j \in T$
- *The Forward Incidence Matrix* : $C^+ = (c_{ij}^+) \in \mathbb{N}_0^{n \times m}$ where $c_{ij}^+ = I^+(p_i, t_j), \forall p_i \in P, t_j \in T$
- *The Incidence Matrix* : $C = C^+ - C^-$
- where n is the number of places and m is the number of transitions.

The Backward Incidence matrix is defined in terms of the arcs that lead from places to transitions, in other words, these arcs are the arcs that are involved in *consuming* tokens. The Forward Incidence matrix is defined by the arcs the lead from transitions to places, which means that they describe the arcs involved in *producing* tokens. Finally, the Incidence Matrix is the difference of these two matrices.

The Petri net shown in Figure 9.1 models the unbounded Producer-Consumer problem and its corresponding Incidence Matrix is found in Figure 9.2

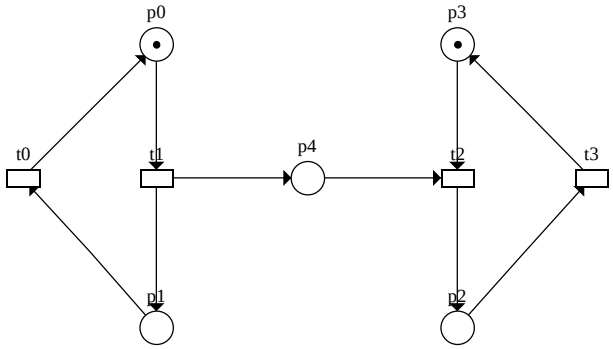


Figure 9.1: The unbounded Producer-Consumer problem

	t0	t1	t2	t3
p0	1	-1	0	0
p1	-1	1	0	0
p2	0	0	1	-1
p3	0	0	-1	1
p4	0	1	-1	0

Figure 9.2: The Incidence Matrix for the net in Figure 9.1

Dynamic Behaviour

With the Incidence Matrix we now have a structure which can completely define the structural properties of Petri nets. It is useful to have a definition for enabling and firing transitions expressed in terms of the Incidence Matrix so that we can describe the net's dynamic behaviour mathematically.

Definition 10.2 Enabled Transition:
A transition, t_i, is *enabled* in a marking, M, iff $M \geq C^{-} e_i$, where e_i is the i_{th} element of the unit vector $V = (0, ..., 1, ..., 0)$ where $\mid V \mid$ is equal to the number of transitions

Definition 10.3 Firing Transitions:
If a transition t_i fires in a marking M then the resultant marking M' is given by $M' = M + Ce_i$.

We can describe a firing sequence in terms of successive e_i by means of the firing vector $f = \sum_{i=1}^{j} e_i$. Any element, f_i, of f describes the number of times that transition t_i fires in the sequence.

It follows then that any reachable marking, M, from the initial marking, M_0, can be defined as:

$$M = M_0 + Cf$$

However not every marking described like this is necessarily reachable from M_0. In this way we describe the structural and dynamic properties of Petri nets using the Incidence matrix.

P- and T-Invariants

With our definition of an Incidence Matrix and related definitions of the dynamic behaviour of Petri nets, we are now in a position to define, mathematically, P- and T-Invariants.

P- and T-Invariants are vectors which operate on the matrix equation derived in the previous section. We multiply the incidence matrix by these vectors on the left, in the case of P-Invariants, or on the right in the case of T-Invariants.

The equation is of the form:

$$v^T M = v^T M_0 + v^T C f$$

where v is a P-Invariant and,

$$Mw = M_0 w + Cfw$$

where w is a T-Invariant depending on the following definitions.

Definition 10.4 P-Invariants:
A *P-Invariant*, $v \in \mathbb{Z}^n$, $v \neq 0$, exists iff $v^T C = 0$, where n is the number of places in the net.

In other words, this *invariant* is a special vector which reduces the Incidence matrix to zero. This has an important implication for our dynamic equations:

- If $v \in \mathbb{Z}^n$ is a P-Invariant, then $\forall M$ reachable from M_0, $v^T M = v^T M_0$

This implies that the weighed sum of the number of tokens on certain places in the net always remains constant.

Definition 10.5 T-Invariants:
A *T-Invariant*, $w \in \mathbb{Z}^m$, $w \neq 0$, exists iff $Cw = 0$, where m is the number of transitions.

This time, if we multiply by the special vector, w, on the right of the incidence matrix, we reduce the incidence matrix to zero. The implication for T-Invariants is that the firing vector w does not change the current marking of the net.

Their are two more definitions we need to make before we can continue with the Invariant theory.

Definition 10.6 Covered by P-Invariants:
A Petri net is *covered* by positive P-Invariants iff $\forall p_i \in P, \exists$ P-Invariant $v \in \mathbb{Z}^n$ with $v_i > 0$

Definition 10.7 Covered by T-Invariants:
A Petri net is *covered* by positive T-Invariants iff $\forall t_i \in T, \exists$ T-Invariant $w \in \mathbb{Z}^m$ with $w_i > 0$

Figure 9.1 has two P-Invariants and one T-Invariant. Its P-Invariants are $v_1 = (1,1,0,0,0)^T$, and $v_2 = (0,0,1,1,0)^T$. Its T-Invariant is $w_1 = (1,1,1,1)$. By our above definitions, the net is covered by T-Invariants but not covered by P-invariants because in neither of the two vectors is their a positive number in the fifth row. This row corresponds to the place p_4. With these results it is only possible to say that the net may be unbounded, particularly, that place p_4 may be unbounded. Additionally, the net may, or may not be live.

Invariant Laws

With the above definitions, we can now compose the two laws relating to the correctness of a Petri net.

1. *The net is covered by positive P-Invariants* $\Rightarrow$ *the net is* **bounded**
2. *The net is* **bounded** *and* **live** $\Rightarrow$ *the net is covered by positive T-Invariants*

Thus, invariant analysis can give us definite results about the boundedness of a GSPN, but can tell us nothing about its liveness. However, if the net is not covered by T-Invariants, it cannot be both bounded and live. Additionally, it should be stressed that if the net is *not* covered by P-Invariants, then it is not necessarily *unbounded*. As an example of this refer to the net in Figure 9.3.

The incidence matrix for this net is:

$$C = \begin{pmatrix} 0 & -1 & 1 & 0 \\ 1 & -1 & -1 & 1 \\ 0 & -1 & 0 & 1 \\ -1 & 1 & 1 & -1 \\ 0 & 1 & -1 & 0 \end{pmatrix}$$

Solving this matrix determines the P-Invariants $v_0 = (10001)$ and $v_1 = (01010)$ and the T-Invariant $w_0 = (2222)$. This shows that the net is covered by T-Invariants but not by P-Invariants. Specifically, p_3 is not covered by P-Invariants. This does not prove that p_3 is unbounded, and this is shown by the fact that p_3 clearly is bounded. The only possible firing sequence in this net is t_1, t_3, t_2, t_0. This clearly can never make p_3 unbounded, since the initial marking is in fact a home state to which we always return. This shows the limitations of invariant analysis but, because they are simple to calculate, they are used frequently in correctness analysis.

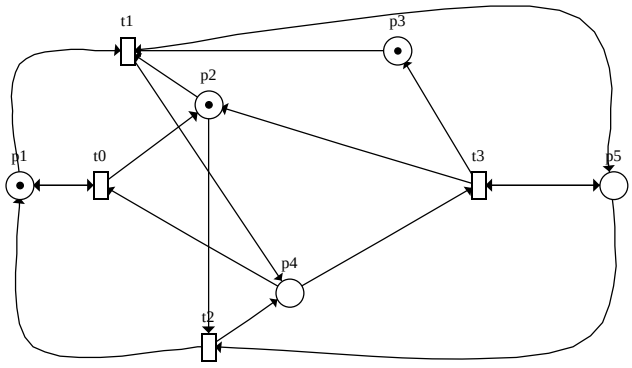


Figure 9.3: A net that cannot be solved with invariants

Implications for Inhibitor Arcs

Thus far we have not discussed the implications for invariant analysis made by the addition of inhibitor arcs to our Petri net model. Normal Petri net arcs model data flow. Inhibitor arcs are *control* arcs, and thus do not function in the same manner.

Inhibitor arcs are not included in the construction of the incidence matrix as there are no rules for their inclusion since they are control arcs. We need to determine if we can perform invariant analysis on Petri nets with the inhibitor arcs removed.

T-Invariants specify sequences of transitions which, when fired in any sequence, will leave the marking of the net unchanged. With Petri nets with inhibitor arcs there is still no guarantee that that sequence can be fired. However if it can then it will leave the marking unchanged.

P-Invariants define groups of places where some weighed sum of their markings remains constant. These are dealt with the same way as T-Invariants since inhibitor arcs can never contribute positively to a place becoming unbounded. Thus, it is safe for us to ignore the inhibitor arcs during the invariant calculation process. This will not contradict the invariant laws described earlier.

9.2.2 Automatically Determining Invariants

Invariants can be automatically calculated in reasonable time. This is very useful for analysis. In DaNAMiCS, we use two algorithms for the calculation of the invariants. We use the same algorithm as implemented in DNAnet , but with the modifications outlined in the next section.

In 1982 Alaiwan and Memmi [2] an algorithm specific to Petri nets is proposed. This algorithm consisted of two $O(2^n)$ parts, the first reducing the problem, and the second calculating the invariants. D'Anna

and Trigila [7] made improvements to this algorithm by reducing the first part to an $O(n^3)$ polynomial time function. This was implemented in DNAnet, and because of its performance, is also implemented in DaNAMiCS.

Figure 9.4 shows the pseudo-code for D'Anna and Trigila's Algorithm. The aim of the algorithm is to find a *minimal generating set* for the vectors in the basis of C.

The algorithm is initialised with the construction of the *extended matrix*. This is simply the $n \times m$ incidence matrix, C, where n is the number of places and m the number of transitions, concatenated with an $n \times n$ identity matrix, B. This produces an $n + m \times n$ extended matrix, E.

To calculate T-Invariants, the matrix is left as is, to calculate P-Invariants, the matrix is first transposed. For our discussion, will leave the matrix as is and assume we are interested in T-Invariants. The algorithm is exactly the same for both invariants, however.

Part one of the algorithm attempts to reduce the size the matrix as much as possible so that the exponentially ordered [19] second part of the algorithm operates on as small a space as possible. Part one is polynomially ordered with $O(n^3)$.

The algorithm maintains sets P^+ and P^-, which are lists of the indexes of the columns of C which contain elements greater or less than zero respectively. If either of these sets is empty, then we delete all the columns with indices in the other set. The columns of the extended matrix are further reduced and deleted by adding combinations of the columns and deleting unnecessary columns. Part one of the algorithm continues until there are no longer non-zero elements in C.

Part two of the algorithm in Figure 9.4 is exponentially ordered. It generates the minimal generating set for the basis of C by operating appropriate linear combinations on the columns of the extended matrix. Since our solution can have no negative solutions as these solutions are meaningless, the algorithm seeks to minimize the number of negative elements which are introduced. This will reduce the complexity of the second part, and thus increase the speed at which it operates.

9.2.3 Implementation

As detailed in the chapter on Object-Oriented design, the invariant analysis was carried out on simple two-dimensional arrays. DNAnet made use of linked lists of arrays of integers. The motivation for using linked lists was that fast insertion and deletion of the columns of the matrix could be carried out.

In DNAnet, is was possible to quickly allocate and deallocate memory for use in the lists. In DaNAMiCS, this is not as easily accomplished and this is due to the fact that DaNAMiCS uses Java. Java is much slower than C++, which DNAnet uses. For this reason, allocating and deallocating reasonable portions of memory is not feasible.

Instead, in DaNAMiCS, we simply overwrite the existing array elements and maintain information on the bounds of the two-dimensional array. In both of the two parts of the algorithm we delete columns from the matrix. Instead of deleting and copying over the old values of the array all the time, we maintain a list of those columns which are to be deleted. At the end of the outermost loop of either of the two parts of the algorithm, we search through two-dimensional array and copy over the deleted columns. This serves as a major improvement in efficiency.

Our implementation, is however, still slower than that of DNAnet as DNAnet uses C++. Please refer to the chapter on "Implementation Approach" for a description of the differences between Java and C++, and how this affected our design.

Initialisation: Construct the extended matrix

$$\left|\begin{array}{c} C \\ B \end{array}\right|$$

where C := Incidence Matrix $\subset \mathbb{Z}^{m \times n}$ and B := Identity Matrix $\subset \mathbb{Z}^{n \times n}$
Part 1: Reduce extended matrix ($O(n^3)$)
while (there exists a non-zero element in C) **do**
if (there exists a row h in C such that the sets $P^+ = \{j | c_{hj} > 0\}$,
$P^- = \{j | c_{hj} < 0\}$ satisfy $P^+ = \emptyset \vee P^- = \emptyset$) **then**
delete from the extended matrix all columns of index $k \subset P^+ \cup P^-$
else
if (there exists a row h in C such that $\mid P^+ \mid = 1 \vee |P^-| = 1$ **then**
let k be the unique index of the column belonging
to P^+ (resp. to P^-)
for $j \subset P^-$ (resp. $j \subset P^+$) **do**
substitute to the column of index j the linear
combination of the columns indexed by k and j with
coefficients $|c_{hj}|$ and $|c_{hk}|$ respectively
endfor
delete from the extended matrix the column of index k
else
let h be the index of a non-zero row of C
let k be the index of a column such that $c_{hk} \neq 0$
for j such that $j \neq k$, $c_{hj} \neq 0$ **do**
substitute to the column of index j the linear
combination of the columns of indices k and j with
coefficients α and β defined as follows:
if sign(c_{hj}) $\neq$ sign(c_{hk}) then $\alpha = |c_{hj}|, \beta = |c_{hk}|$
else $\alpha = -|c_{hj}|, \beta = |c_{hk}|$ **endif**
endfor
delete from the extended matrix the column of index k
endif
endif
endwhile
Part 2: Find minimum generating set $O(2^n)$
while (the matrix B constants a row of index h with negative elements) **do**
let $P^+ = \{j | c_{hj} > 0\}$, $P^- = \{j | c_{hj} < 0\}$
if ($P^+ \neq \emptyset$) **then**
for $(j, k) \subset P^+ \times P^-$ **do**
operate a linear combination on the columns of indices j and k in
order to get a new column having the h-th element equal to zero,
divide this column by the GCD of its elements and append this
column to the matrix B
endfor
endif
delete from B all columns of index $k \subset P^-$
endwhile
delete from B all the columns having non-minimal support

Figure 9.4: Invariant algorithm pseudo-code

9.3 Coverability Analysis

Invariant analysis cannot, in many cases, yield conclusive results regarding the correctness of Petri net models. We need a method of analysis that can exactly determine correctness properties of any net. The best and most thorough means to achieve this is by exhaustively searching the reachability space of the Petri net. This means calculating all the possible markings that the net can ever reach from the initial marking. Once we have generated this space, we can search through it to find the properties we are interested in, and make conclusions about the Petri net. This process is know as *Coverability analysis*.

Coverability analysis is a two phase process. The first phase involves the construction of the *coverability graph* which is the set of all reachable markings. The second phase involves the analysis of this graph. The first phase is by far the most time-consuming and complex and thus we will concentrate on it here. This first phase is called *coverability graph generation* and we will deal with it in detail; the second phase is the analysis phase and it will be dealt with later.

Coverability analysis of ordinary Petri nets is accomplished by following a straight-forward algorithm, however, since DaNAMiCS provides an extended class of Petri nets, that is, nets which can contain inhibitor arcs, timed and immediate transitions, and place capacities, the construction of the coverability graph is more involved. We had to construct our own algorithm to find the coverability graph for nets with these extensions which meant revising and vastly extending the original algorithm. A summary of our findings and the new algorithm is thus presented here.

9.3.1 Coverability Graph Generation of Ordinary Petri nets

The coverability set is often drawn as a graph where the nodes in the tree represent the possible markings of the Petri net. Two nodes in the graph M and M' are connected with a directed arc if and only if there exists some transition which is enabled in marking M and transforms the marking to M'. The coverability graph, needed for the functional, as well as Markov analysis, is generated from the reachability graph. The latter is generated by starting at the initial marking and adding all reachable markings. Reachable markings are generated by firing all enabled transitions in turn, which are then connected to the initial marking via an arc. This process is repeated on each of the new generated markings.

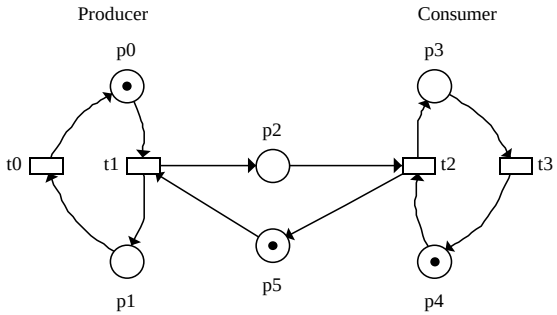


Figure 9.5: A Petri net model of the Producer - Consumer problem with a buffer of size 1.

Figure 9.5 illustrates a Petri net modeling the Bounded-Buffer Producer Consumer Problem. The coverability graph that is generated from this Petri net is shown in Figure 9.6. The algorithm to determine the

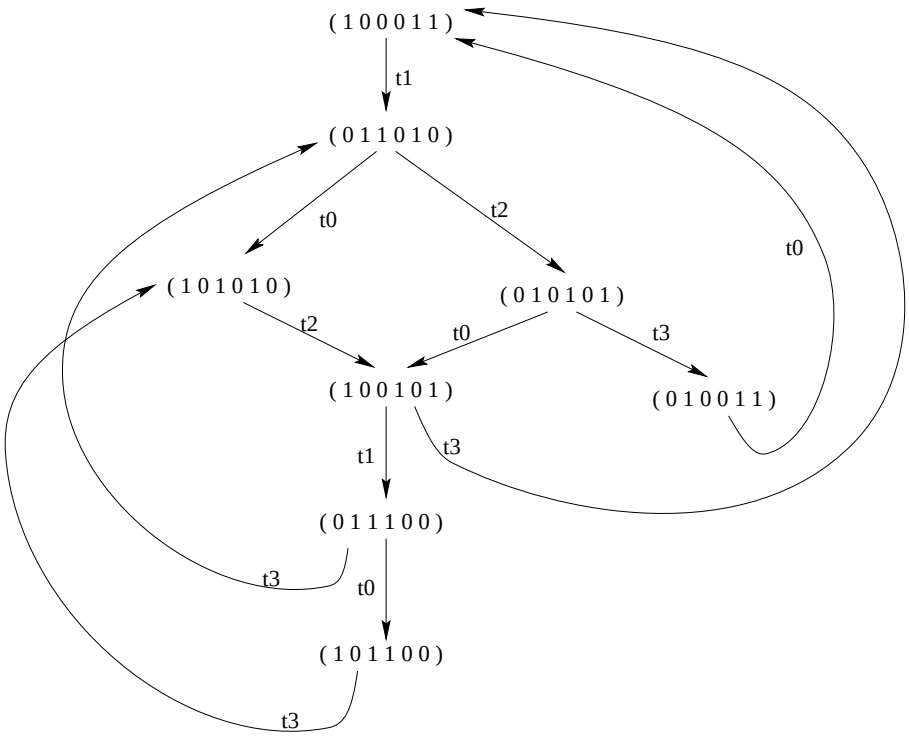


Figure 9.6: The Coverability graph for Figure 9.5.

coverability graph for an ordinary Petri net is illustrated in Figure 9.7.

The algorithm explores all possible markings. For each marking, all markings reachable from that marking by firing a single transition, are added to the list of markings. The marking is then examined to determine if any of the places in it could become unbounded.

As soon as it is found that it is possible for a place to become unbounded, the number of tokens on that place is set to infinity, represented by ω. It is obviously essential to find and mark these unbounded places, because if we did not, then we would have an algorithm that would loop infinitely on certain inputs, leading to an infinite reachability graph.

9.3.2 Extension to coverability graph generation

Analysis of the graph generated by the above algorithm solves the correctness of ordinary Petri nets, but it does not work on all the variations of Petri nets. This algorithm does not cope with the complications introduced by GSPNs, inhibitor arcs or the ability to place bounds on the maximum number of tokens in a place. In particular, the rules for deciding when a place becomes unbounded have to be refined. In this section we discuss how the extended algorithm with these refined rules, given in Figure 9.16, was derived.

For each of the three extensions to Petri nets that we have made we first show where the original analysis techniques break down. We discuss why the techniques fail and then generalise the problem to determine how the coverability analysis method must be improved to ensure it will always provide accurate results.

Initial step:
$S := \{M_0\}$, where S represents the markings to be explored, and M_0 is the initial marking
while $S \neq \phi$ **do**
choose any $s \in S$
remove s from S
for each transition t enabled by s **do**
find new marking s' generated by the firing of t from s
connect s and s' by a directed arc labeled with t
if s' does not exists as a node in the graph **then**
add marking s' to S
set the parent of s' to s
for all markings on path from s' to M_0 **do**
if s' covers m
for all places p where $s'[p] > m[p]$
set $s[p]' = \omega$
endfor
endif
endfor
endif
endfor
endwhile
end

Figure 9.7: Coverability graph generation algorithm for an ordinary Petri net.

Generalised Stochastic Petri Nets

Generalised Stochastic Petri net behaviour is not the same as that of an ordinary Petri net. This change in behaviour is due to the introduction of transitions that have an infinite firing rate. These transitions are called immediate transitions.

We start by extending the coverability graph generation algorithm to cover the case of immediate transitions in GSPNs. The presence of immediate transitions could prevent a sequence of firings from being repeated. Places that are unbounded without immediate transitions might be bounded with them. Rules need to be formulated to ensure that a firing sequence can be repeated an arbitrary number of times - each time increasing the number of tokens on a place.

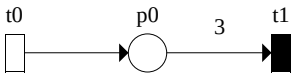


Figure 9.8: An example GSPN were the ordinary Coverability analysis does not work.

Figure 9.8 illustrates a simple GSPN. Using the original rules for coverability graph generation, place p_0 will be marked as unbounded since t_0 can fire an infinite number of times. However place p_0 is in fact

bounded with a maximum of 3 tokens. The immediate transition t_1 always has precedence over t_0 and thus prevents it from firing when place p_0 contains 3 tokens.

The condition for marking a place as unbounded needs to be more stringent. It must ensure that the firing cycle does not eventually change the marking from *tangible* to *vanishing*, i.e. from a marking without any immediate transitions enabled to a marking with immediate transitions enabled.

After finding a marking m that is covered by s', two additional rules must be checked before a place can be marked as unbounded:

- s' and m are both vanishing or tangible.
- setting place p in marking s' to ω does not change marking s' from tangible to vanishing.

The first of these two rules ensures that the firing sequence will not be forced to generate a different path by the immediate transitions. The second ensures that repeating the firing sequence will not eventually force the firing sequence to diverge from the one that would otherwise cause unboundedness. Together, these rules ensure that the firing sequence from s to s' can be repeated infinitely often. Thus, if they are satisfied, then $s[p]'$ can safely be set to ω, marking p as unbounded.

Inhibitor arcs

Inhibitor arcs prevent a transition from firing if the place to which they are attached contains tokens. Inhibitor arcs thus could cause some places that were previously unbounded to become bounded. However it cannot cause a net that was bounded without the inhibitor arcs, to become unbounded with them, since the enabled transitions with inhibitor arcs are a subset of the enabled transitions without them.

It is not possible in Turing machines, or in Petri nets with inhibitor arcs, to determine the correctness of the system. Since it is possible to decide the correctness of a system from the coverability graph there will be cases [19] when it is not possible to create the coverability graph.

We can, however, still decide the correctness and generate the coverability graph for many of these nets. The cases that DaNAMiCS is capable of performing analysis correctly are:

1. the net becomes unbounded by a firing sequence that can be repeated.
2. the net is bounded and the coverability graph does not exceed some depth that is relative to the size of the net.

In the first case, the net is unbounded as there exists a sequence that can be repeated to cause the number of tokens on some place to become infinitely large. In the second case, the complete coverability graph would be generated. This would indicate that the net is bounded.

Figure 9.9 illustrates a system modeled using Petri nets with inhibitor arcs. Figure 9.10 shows the correct coverability graph of the same system. The original algorithm for generating the coverability graph would have marked place p_3 as unbounded when marking M_3 is reached. This would happen because marking M_3 covers marking M_1 and $M_3[p_3] > M_1[p_3]$. This is inaccurate since the net becomes deadlocked when, in M_3, the only enabled transition fires, moving to marking M_4.

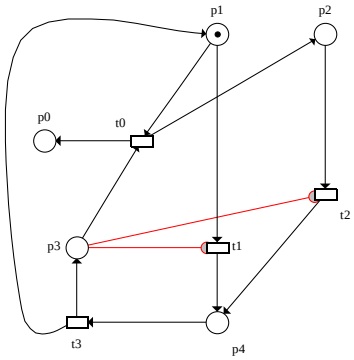


Figure 9.9: A Petri net with inhibitor arcs.

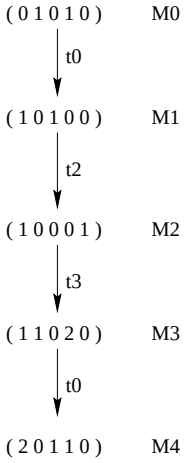


Figure 9.10: Coverability graph for the Petri net in Figure 9.9.

The problem is that as the number of tokens on a place increases from zero to one, certain transitions that might have been enabled become disabled because of the inhibitor arcs. In that case, if a marking M_3 covers marking M_1, there is no guarantee that the sequence of transitions fired from M_1 to M_3 can be repeated as was the case in ordinary Petri nets.

Creating the additional rules to handle inhibitor arcs was one of the more difficult aspects in the development of the coverability graph generation algorithm for DaNAMiCS. We decided that the only unbounded places that we would detect are those that become unbounded by repeating a fixed firing sequence. This should be sufficient for most real-world systems, such as queueing systems.

Many different techniques were examined to gain an intuitive feel for how the algorithm should work. From these examples, different methods for deciding when a place can be determined as unbounded, were discussed. These variations were then examined and checked for accuracy.

One of the first examples studied was a simple Producer - Consumer system. This system is illustrated in Figure 9.11. The original method declared the buffer p_2 to be unbounded. However it is bounded with a

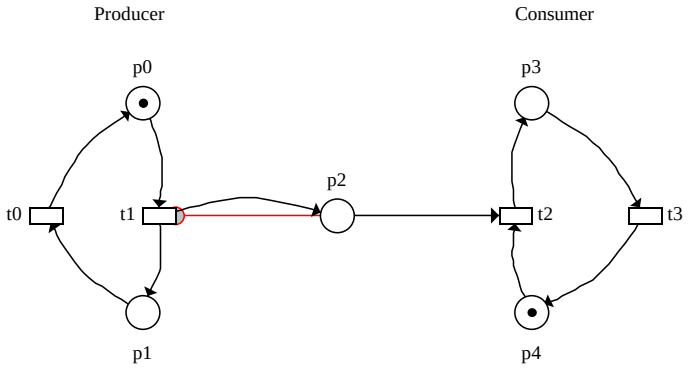


Figure 9.11: A Producer - Consumer problem with the buffer bounded by an inhibitor arc.

maximum value of one because the inhibitor arc from p_2 to t_1 inhibits t_1 from firing whenever there is a token on p_2. This seemed to indicate that some additional checks on the places with inhibitor arcs from them, were necessary. Such places could start inhibiting transitions as their number of tokens increased from zero to one.

If we start with some initial marking, M_0, and follow some sequence of transitions to M_1, where M_1 covers M_0, and the number of tokens on the place in question has increased, we need to consider how the inhibitor arcs are behaving. Either they were inhibiting transitions all of the time, some of the time, or none of the time.

In the first case, where none of the inhibitor arcs are inhibiting in that firing sequence, the net is behaving as if their are no inhibitor arcs present. Thus nothing needs to be added to the algorithm to handle this. If, on the other hand, they were inhibiting throughout the firing sequence, then the net is behaving as if the inhibited transitions are not present. This is similar to the first case and also does not prevent places from becoming unbounded, since the sequence from M_0 to M_1 can be repeated.

The most interesting case is where transitions are sometimes inhibited in the cycle. Here, the place could be bounded as is the case in Figure 9.11, or unbounded. One method of determining whether the place is unbounded, is to check whether all places that have inhibitor arcs from them start and finish with the same number of tokens on them. If this is the case, the firing sequence from M_0 to M_1 can be repeated. This is so, since if the firing sequence is repeated then the marking on places with inhibitor arcs from them will be identical in subsequent loops. Thus transitions are being inhibited if and only if they where inhibited at the respective position in the first loop of the firing sequence.

If, on the other hand, the number of tokens on places with inhibitor arcs has increased, but was zero somewhere in the cycle, then there is no guarantee that the cycle can be repeated. An example of a bounded net with this property is shown in Figure 9.9, while an example where the net is unbounded is seen in Figure 9.12. This is caused by certain transitions that might not have been inhibited at a certain stage in the firing sequence, while in successive iterations become inhibited, thus preventing this sequence from causing a cycle.

If, however, the place becomes unbounded in this cycle then it will be detected within the next loop of the cycle. This follows from the definition of the marking covering another. If that place has different values in the different markings then it must have increased. This would also increase its lowest value in the cycle.

Thus it would reach zero and the inhibitor arcs leaving it would no longer inhibit. They would then be resolved by the first part of the rule.

These rules for inhibitor arcs can be summarised as follows:

- all places p, for any markings M or s', that have inhibitor arcs connected to them have **either**
  - $M(p) = s'(p)$ **or**
  - all markings N on the path from M to s' are such that $N(p) > 0$.

The first of these rules states that if the marking of a place is unchanged by a loop of a cycle, then that cycle can be repeated. The second of the rules shows that if the inhibitor arcs were not inhibiting a transition at any stage in the firing sequence, then the net is behaving as if they were not there.

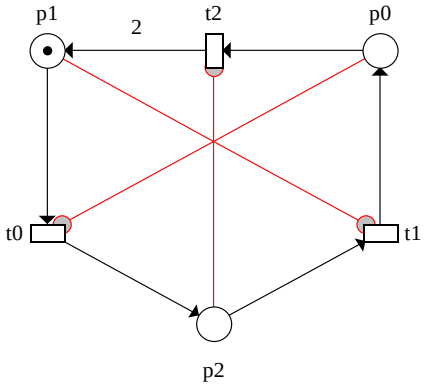


Figure 9.12: An unbounded Petri net without a repeating cycle.

This does not imply that if these rules do not determine a place to be unbounded, that it is necessarily bounded. But, if they do detect a place as being unbounded, then that place is certainly unbounded. In DaNAMiCS, if these rules determine a place to be unbounded, then the cycle which causes it to become unbounded is displayed and animated for easy visualisation.

Since the analysis on Petri nets with the *zero check* is not decidable in the general case, it is impossible to create a Turing machine or computer program that generates the coverability graph in all cases. There are cases, such as in Figure 9.12, where there is no firing sequence that is repeated, yet all the places are unbounded. In Figure 9.12 the following firing sequence will lead to all places having an arbitrary large number of tokens on them.

$$t_0, t_1, t_2, t_0, t_0, t_1, t_1, t_2, t_2, t_0, t_0, t_0, t_1, t_1, t_1, t_2, t_2, t_2, \ldots$$

If something likes this occurs during the graph generation then DaNAMiCS halts when the length of the path from M_0 exceeds a predetermined length. This length is a function of the Petri net size. Places will then get marked as *possibly unbounded* instead of as *unbounded*. DaNAMiCS will also catch the “OutOfMemory”

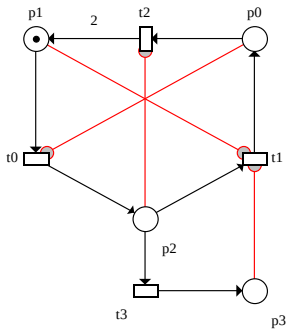


Figure 9.13: An unbounded Petri net without a repeating cycle.

exception when the coverability graph becomes very large. In this no assumptions can be made about the correctness of the Petri net, although it is likely to be unbounded.

In Figure 9.13 there is another example of a Petri net without a repeating cycle. For place p_3 to become unbounded we need to fire the same sequence that led to the net in Figure 9.12 becoming unbounded. However, any sequence that increases the number of tokens on place p_3 to greater than zero, also leads to deadlock. Thus, the initial part of the sequence is different and leads to a different marking in p_3 upon reaching deadlock. The number of tokens in p_3 can be arbitrary large implying that the place is unbounded.

In DaNAMiCS, both of these examples are detected to be *possibly unbounded*, based on how the firing sequences behave.

Capacities

Capacities place bounds on the maximum number of tokens allowed on a place.

Figure 9.14: A simple Petri net where p_0 has a capacity of 3.

The generation of the coverability graph needs to be modified to handle capacities on places. The simple example in Figure 9.14 shows a net where the graph generation would break down without any adjustments.

The reason it needs to be modified is quite obvious. No limited place can ever be unbounded. However, that is not all that needs to be checked. Introducing capacities has side-effects in the remainder of the net.

Figure 9.15 shows a net where the repeated firing of transitions t_0 and t_1 would cause places p_2 and p_3 to become unbounded - if there were no limit on their capacities. The interesting feature of this net is that placing a capacity on p_2 causes p_3 to become bounded where before it would not be. Thus we need to check more than just whether a place has a capacity.

We need to check all places that have finite capacities. If any have increased from the start to the end of the cycle then that cycle cannot be used to determine whether any places are unbounded. The rule for capacities

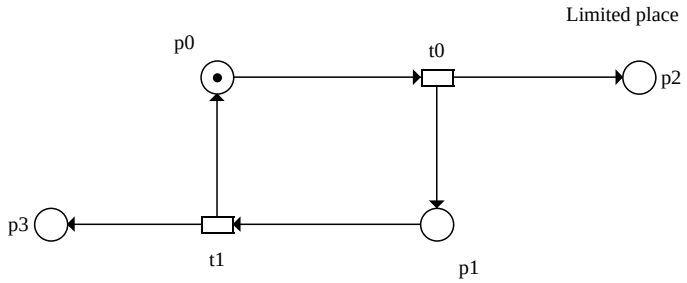


Figure 9.15: A Petri net where capacity on p_2 causes p_3 to be bounded.

is to insist on all places that have a capacity less than infinity to have an equal number of tokens in markings s and s'. This will ensure that the firing sequence can be repeated.

9.3.3 A New Algorithm for Coverability Graph Generation

With the extensions we have mentioned, and the way we handle them in the coverability graph generation process, we are in a position to present a refined algorithm for building coverability graphs to handle more general cases. In this way we can check the correctness of a modeled system in most cases.

The algorithm is given in Figure 9.16. This algorithm will be able to predict with more accuracy, whether a net is *correct* or not. It was designed specifically to perform correctness analysis on nets extended with inhibitor arcs, capacities, and time. It is important to note that it cannot be absolutely determined whether the net is correct or not, but this algorithm will allow accurate analysis on a much wider range of Petri net models.

After having discussed the graph generation process, we will now describe the process of analysing the coverability graph to find such properties as *liveness*, *boundedness*, *home states*, *safeness*, and *persistence*.

9.3.4 Coverability Graph Analysis

With the coverability graph now constructed, we are in a position to analyse it for the various correctness properties. The means by which we accomplish this is discussed here.

Liveness

Perhaps the most important property of a net, is its liveness. We wish to know whether a given system will ever crash or whether certain parts of the system will loop infinitely. The liveness of a Petri net can be determined from the coverability graph by means of the following two laws:

- *Net is live and bounded* $\Rightarrow$ *Strongly connected coverability graph.*
- *Net is bounded, the net is live* $\iff$ *all transitions appear as a label in at least one of the final strongly connected components of the coverability graph*

Initial step:
$S := \{M_0\}$ (S represents the markings to be explored)
while $S \neq \phi$ **do**
choose any $s \in S$
remove s from S
for each transition t enabled by s **do**
find new marking s' generated by the firing of t from s
connect s and s' by a directed arc labeled with t
if s' does not exist as a node in the graph **then**
add marking s' to S
set the parent of s' to s
for all markings on path from s' to M_0 **do**
if s' covers m
and s' and m are both vanishing or tangible
and for all places p where $m[p] > s'[p]$
converting $s'[p]$ to ω
does not change marking s' from tangible to vanishing
endfor
and for all places p that have connected inhibitor arcs
$m[p] = s'[p]$
or for all markings n on path from m to s'
$n[p] > 0$
endfor
endfor
and for all places p that have finite capacities
$s[p] == s'[p]$
endfor
then
for all places p where $s'[p] > m[p]$
set $s[p]' = \omega$
endfor
endif
endfor
endif
endfor
endwhile

Figure 9.16: Coverability graph generation algorithm.

A strongly connected graph is graph where, from any particular node it is possible, via some path, to reach all the other nodes in the graph. A final component in a graph is a set of nodes which, once entered, can never be left by any path. Note that a strongly connected graph has exactly one final component. To sum up the above two laws into a form useful for coverability analysis we can say that if every transition can be enabled in at least one of the markings of all final components of the coverability graph then the net is live. So, the problem for determining liveness becomes a problem of finding final strongly connected

components, which is a well-understood algorithmic problem.

The algorithm is a simple depth-first search algorithm which is shown in Figure 9.17. This algorithm is of linear complexity and was implemented efficiently on the adjacency list graph structure.

The algorithm maintains two arrays *dfsNumber* and *low* which correspond to vertices in the graph. "dfsNumber" is a list of the orders in which the graph nodes were visited. Each element in "low" is initially set to the corresponding value of "dfsNumber" which is the currently lowest valued vertex in the same component of the graph. After visiting a node we "back up" from it and check to see whether we have traversed a boundary of a strongly connected component. This is done by checking if the current node's "dfsNumber" is the same as its "low" number. If we have traversed a strongly connected component, then we mark all those elements as being in the same component. If this is not the case then the "low" array is updated accordingly.

Once we have determined all the components in the graph we check their finality by examining each node in each component. If any node in a particular component contains an arc to a node in a different component, then the component to which that node belongs is not final.

Boundedness

Determining whether a Petri net is bounded from the coverability graph is straight-forward. In the process of generating the coverability graph we mark places as being unbounded with an ω. We simply search through the entire graph for ω and, if we find one, we alert the user that the corresponding place is unbounded.

In DaNAMiCS, we allow the use of an extended net class. As was previously mentioned, it is not always possible to determine the correctness properties of a Petri net with these extensions and thus we not only mark places and being *unbounded*, but also with a different symbol if the place can be considered to be *possibly unbounded*. If a place is possibly unbounded, we are alerting the user to the fact that the respective place may be unbounded, but we cannot be certain. From this point onwards, the user will have to use heuristics to determine whether that place is definitely unbounded or not.

Home States

To reiterate, a home state is a marking of the Petri net that is reachable from every other marking in the graph.

The presence of home states is very simply calculated. If the number of final, strongly connected components is equal to one, then the net contains home states.

Safeness

As was outlined in the theory chapter, a Petri net is *safe* if the number of tokens in a place never exceeds one. Safeness is simply determined by linearly searching all the markings of the graph. If we never encounter a marking with a place which contains more than one token, then the net is safe.

Persistence

Persistence was also outlined in the theory chapter. A net is persistent if the firing on one transition never disables another, concurrently enabled transition. Persistence is very important for the analysis of systems, and is determined in the following manner.

```
procedure StrongComponents(V, E)

var     dfsNumber, low: array[VertexType] of integer
        dfn: integer
        componentStack: Stack
        removed: array[VertexType] of boolean

procedure SCompDFS(v:VertexType)
        begin
                dfn := dfn + 1; {set depth first search number in order encountered}
                dfsNumber[v] := dfn; low[v] = dfn;
                removed[v] :=false;
                for each edge e leaving v do
                        w = destination of e
                        if dfsNumber[w] = 0 then {haven't visited w yet}
                                SCompDFS(w); {visit it}
                                low[v] = min(low[v], low[w]);
                        else {w already visited}
                                if w not removed then
                                        low[v] := min(dfsNumber[w], low[v])
                                endif
                        endif
                endfor
        {backing up from v}
        if low[v] = dfsNumber][v] then {new component encountered}
                output v
                removed[v] = true
                while componentStack is non-empty and dfsNumber[Top(componentStack)] > dfsNumber[v] do
                        output Top(componentStack)
                        removed[Top(componentStack)] = true
                        pop componentStack
                endwhile {while vertices on stack are in current component}
        else {not a new component}
                push v onto componentStack
        endif
end

begin {StrongComponents}
        for v := 1 to n do dfsNumber[v] := 0 endfor
        dfn := 0
        for v := 1 to n do
                if dfsNumber[v] = 0 then SCompDFS(v) endif
        endfor
end {StrongComponents}
```

Figure 9.17: Strongly connected components algorithm

begin isPersistent
    $S =$ { All enabled transitions at M_0, $(EN(M_0))$}
    $M =$ { All the markings reachable from M_0, $R(PN, M_0)$}
    while $(M \neq \phi)$ **do:**
        for ($R(PN, m)$ the reachable markings from $m \in M$)
            $N = \{EN(m) -$ the transition required to get to m $\}$
            if $(N \neq S)$
                The net is Not persistent.
            endif
            $M = M \cup \{R(PN, m)\} - m$
        endfor
    endwhile
end

Figure 9.18: Detecting Persistence Automatically.

We visit each marking in the graph. At each marking we obtain a list of all the enabled transitions. We then follow all the outgoing arcs of the current marking and obtain the list of newly enabled transitions at each marking. If ever the situation occurs where an enabled transition in the previous marking is now longer enabled, then the net is not persistent. This algorithm is given in Figure 9.18

9.3.5 Implementation

The object structure for the coverability graph has been shown in Chapter 8. This structure is optimal, and was decided upon when it was realised that Java's memory management capabilities were severely lacking. Previously we were creating and discarding marking objects as necessary, but this was found to be far too slow. Consequently a system whereby only one of each possible marking is generated. This greatly reduces memory usage in a problem where the state space can explode exponentially.

For the coverability graph generation problem, the algorithms to detect enabled transitions and then fire them in turn was optimised as it is called several thousand times in a typical graph generation sequence.

The structure is searchable linearly and as a graph equally quickly. The importance of this was seen in the section describing the second phase of the algorithm where we need to search the entire list of markings linearly, and where we need to perform a depth-first search algorithm on the graph. These two operations were thus adequately catered for.

9.4 Summary

In this chapter, we have presented two means to conduct correctness analysis. The exhaustive method of generating the coverability graph of a Petri net was examined and extended in DaNAMiCS to cope with the extensions implemented in the DaNAMiCS environment. Additionally, invariant analysis was formalised and discussed. Finally, we presented on overview of the means by which these analysis techniques were implemented in DaNAMiCS.

Chapter 10

Performance Analysis

10.1 Overview

Implementing a system that is not functionally correct is costly and wasteful of resources. We examined the concept of functional correctness in the previous chapter. It is clear that correctness in a system is of primary concern.

However, even if the modeled system is *correct*, it may be so inefficient that it is infeasible as a solution to a real-world problem. This is where performance analysis can prove invaluable. Performance analysis can give us insight into how well the system will behave in a real-world environment. The information gleamed from performance analysis could be as essential to modeling as the system correctness is. We would like to ensure that our modeled systems not only provide functionally correct systems, but also systems that perform optimally.

In real-world systems part of the specification is usually a measure of how well the system should function under certain conditions. It is often required that the system should maintain a certain level of performance and this is specified in the original systems contract. For example, in safety-critical systems, such as auto-pilot controls, it is vital that the system maintains a certain level of responsiveness.

Performance analysis is the process of determining performance metrics regarding our system and can be carried out on Generalised Stochastic Petri nets (GSPNs). The metrics DaNAMiCS can calculate include *transition throughput* and *the mean number of tokens on a place*. These two metrics can relate directly to such real-world problems as the number of packets through a router in a given interval, and the mean size of a database queueing buffer.

Performance analysis is carried out in two different ways in DaNAMiCS. It can be performed by simulation, which monitors the behaviour of the net over a given time period, or by steady state analysis, which solves the net exactly using Markov theory. The steady state analysis is not performed by DaNAMiCS directly, but rather by DNAmaca which is an external program written by W. Knottenbelt.

In this chapter the theory behind simulation and steady state performance analysis is detailed as well as DaNAMiCS' approach to solving the performance problems.

10.2 Simulation

Simulation is the process of iterating through the possible markings of a Petri net and firing enabled transitions automatically. The benefit of simulation is that it yields results quickly, while the disadvantage is that it does not produce exact results. The longer one allows the simulation to "run" for, the more accurate the results become. This makes simulation a fast, flexible performance analysis tool.

Each simulation cycle generates a new marking from which the metrics we are interested in are calculated. It is an iterative process and in addition to the above metrics, we also keep track of the probability of error in our results.

10.2.1 Statistics

In this section, we describe the mathematics behind simulation performance analysis.

At any particular marking, M_k, in the simulation cycle, an enabled transition t_i fires with probability:

$$r_i = \frac{\sum_{j \in EN(M_k)} \lambda_j}{\lambda_i}$$

where $EN(M_k)$ is the set of enabled transitions in the marking M_k and λ_j is the rate of the specific transition if it is a timed transition, or the weight of the transition if it is immediate.

Since we specifically are interested in the time taken for events to occur in our performance model, we need to know how long the model spends in each state. This is known as the *sojourn time*, $t_s(M_k)$, and is given by the exponential distribution function:

$$f(t) = \lambda e^{-\lambda t}$$

where $\lambda = \sum_{i \in EN(M_k)} \lambda_i$.

The two metrics we are interested in are transition throughput and the mean number of tokens per place. In the simulation cycle, we continually update these two metrics with the following formulas:

Transition throughput is given by:

$$t_r(i) = \frac{f(t_i)}{\sum_k t_s(M_k)}$$

where $f(t_i)$ is the number of times transition t_i as fired thus far.

Mean number of tokens per place:

$$\overline{m_{k+1}} = \frac{t_n \overline{m_k} + t_s(M_k)}{t_{k+1}}$$

and this equation is calculated incrementally with t_k being the elapsed time since the simulation began.

These values all have associated errors and their variances can be found by the following formula:

$$\sigma^2 = \frac{1}{n-1}\left[\sum x^2 - \frac{1}{n}\left(\sum x\right)^2\right]$$

A confidence level of 95% can be determined by:

$$x_i = \pm 1.95\sqrt{\sigma}$$

Thus, with the statistics in place, we can determine the simulation metrics and their associated error values within the specified confidence.

10.2.2 Simulation Cycle

In this section we present the simulation cycle algorithm which incrementally calculates the transition throughput and mean number of tokens per place in the net. The simulation pseudo-code is shown in Figure 10.1.

The algorithm is initialised with the initial marking and the transitions that are enabled at that marking are calculated.

Initialisation: Set Timer = 0, Set Initial Marking. $k := 0$
$EN(M_k) = EN(M_0)$
while ($k <$ Simulation Length **and** $EN(M_k) \neq \phi$) **do**
    if (M_k is vanishing) **then**
        select a certain t_i from $EN(M_k)$ with probability $p_i(M_k) = \frac{W_i(M_k)}{\sum_{j \in EN(M_k)} W_j(M_k)}$
    else /* M_k is tangible */
        select a certain t_i from $EN(M_k)$ with probability $p_i(M_k) = \frac{\lambda_i(M_k)}{\sum_{j \in EN(M_k)} \lambda_j(M_k)}$
    endif
    Calculate sojourn time for the selected transition
    Fire the selected transition
    $k := k + 1$
    $EN(M_k)$ = { all transitions enabled at M_k }
    update transition throughput and the mean number of tokens per place
endwhile if ($EN(M_k) = \phi$) **then** Note that the system deadlocked and that the values thus far calculated are unreliable

Figure 10.1: Simulation Cycle Pseudo-Code

A transition is selected based on its probability of firing, and is fired generating the new marking. If the fired transition was a timed transition then the sojourn time must be calculated and added to the total time elapsed.

This process is repeated until the total number of transitions fired is equal to the length of the simulation. In other words, the simulation length, specified by the user, is actually the number of transitions fired, or the number of markings that must be explored.

10.2.3 Implementation

The implementation of the above simulation algorithm is accomplished simply by maintaining lists of the places and transitions with their associated metrics and errors. Two simple functions were implemented to select a transition to fire, based on its probability and whether or not the transition is timed or immediate. Finally, the results of the simulation can be easily retrieved and displayed or saved.

10.3 Steady State

10.3.1 Overview

In order to analyse the performance of a GSPN analytically, we need to analyse the embedded *Markov chain* of the corresponding *stochastic process* [4]. Refer to Sections 4.1.1 and 4.1.2 for an overview of stochastic processes and Markov theory.

10.3.2 DNAmaca and the creation of the Markov models

Recall that the behaviour of a Markov chain can be characterised by specifying all the states of the system and the transitions from one state to another. The *state description* vector is a vector of attributes which, jointly, characterises the system. Particular values of these attributes uniquely describe a particular state. All possible states of the system can be obtained by enumerating the state description vector .

Specifying the state description vector

For any Petri net, each distinct marking which is reachable from the initial marking M_0 is a different state of the system. The *state space* or *reachability set* of a Petri net is therefore all markings reachable from M_0. A marking is defined by the number of tokens on each place, so:

$M = (p_1, p_2, ..., p_n)$ (where p_i is the number of tokens on place P_i)

is the state description vector of the underlying Markov chain.

For this reason DaNAMiCS creates DNAmaca models with state descriptors which are integer arrays, where the size of the array is the number of places i.e. the vector defines a marking.

Specifying the change in state

When an enabled transition fires, it yields a new marking. In other words, firing transitions corresponds to changes in state of the underlying Markov process. The enabling conditions for a transition give the conditions for the Markov chain to make a change in state. When a transition fires, the state vector is updated according to the normal Petri net firing rules.

These enabling conditions are written to the model file in algebraic form. For example, Figure 10.2 shows a simple Petri net with one transition and three places. From the input arcs of transition t_1, we can see that two tokens will be removed from place P_1 and one token from P_2. The markings in this case are $M_0 = (2, 1, 0)$ and $M_1 = (0, 0, 1)$ (the reachability graph is shown in Figure 10.3). So, the two states in this case are M_0 and M_1, and the condition for moving from M_0 to M_1 is :

$((p_1 \geq 2) \wedge (p_2 \geq 1))$

These are given by the input arcs of transition t_1.

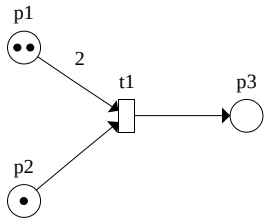


Figure 10.2: A simple Petri net with 3 places and 1 transition

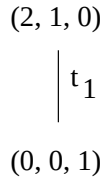


Figure 10.3: The reachability graph of Figure 10.2

When a transition fires, it removes tokens from its input places and adds tokens to its output places. This corresponds to the *action* that is performed when a state it entered. In the above example, when state M_1 is entered, the number of tokens on place P_1 is decreased by two, the number of tokens on place P_2 is decreased by one and the number of tokens on place P_3 is increased by one. In algebraic form, this is:
$p_1 = p_1 - 2$
$p_2 = p_2 - 1$
$p_3 = p_3 + 1$
These state transition actions are specified in the model file by DaNAMiCS for use by DNAmaca.

The probability of changing states is calculated from the weights/rates of the transitions that are enabled at the marking which corresponds to the present state. DNAmaca calculates these probabilities and uses them to find the steady state solution. DaNAMiCS simply adds the required weight/rate information to the Markov model file.

10.3.3 Models with inhibitor arcs

Section 9.3.2 shows how the introduction of inhibitor arcs creates many complications for coverability analysis. However, with performance by steady state analysis, the introduction of inhibitor arcs does not drastically change the analysis.

The use of inhibitor arcs does alter the conditions imposed on changing state. The normal conditions for changing state is that there are enough tokens on each of the input places to the transition which initiates the state change. Now, for each transition that has inhibitor arcs, every place connected to the transition via an inhibitor arc must have *no tokens* in order to fire.

This test for zero, introduced with inhibitor arcs, is the only addition that must be made for performance analysis by DNAmaca to work. DaNAMiCS informs DNAmaca about these conditions simply by writing them, along with the normal conditions, to the model file.

To illustrate how this works, examine Figure 10.4 which shows a simple net with an inhibitor arc. Figure 10.5 shows the corresponding state space. The condition for changing from state M_1 to M_2 is:

$$((p_1 > 1) \wedge (p_3 = 0))$$

The first term of the condition comes from the normal arc of weight 1 connecting place P_1 to transition t_1. The second term is a result of the inhibitor arc connecting place P_3 and transition t_1. Once in state M_3, a change in state can no longer occur, since the input conditions of both transitions are no longer met (a change in state only occurs when a transition fires). Looking at state M_3, we see that:

$p_1 = 1$

$p_2 = 0$
$p_3 = 1$

Transition t_2 cannot fire since $p_2 = 0$ and its firing condition is:

$(p_2 >= 1)$

i.e. there must be at least 1 token on place P_2. Unfortunately, p_3 has one token on it, so the inhibitor arc prevents it from firing since it imposes the condition that p_3 must be 0. DNAmaca takes this into account by using these conditions when it is generating the state space of the Markov chain.

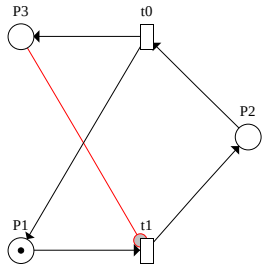


Figure 10.4: A Petri net with an inhibitor arc

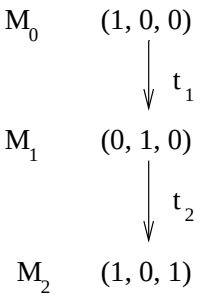


Figure 10.5: The reachability graph for the Petri net in Figure 10.4

10.3.4 Performance measures

The performance measures that DaNAMiCS receives from DNAmaca are:

- for Transitions:
  1. transition *throughput*
  2. *enabled probability*
- for Places:

1. *mean number of tokens* per place
2. *mean distribution*

These measures will be examined in the sections which follow.

Throughput

Transition throughput is defined as the mean number of firings at steady state[4]. Throughput figures indicate where the Petri net model spends most of its time. Highly used sections of the code can be detected for concentrated optimization efforts.

Petri net transitions correspond to actions in the real world. For example, in a protocol model, there may be a transition labeled *Send Request*. If the transition throughput for *Send Request* is high, that means that a lot of time will be spent carrying out that particular action. Also, if some estimate of how long it would take to complete one *Send Request* action is known, then the overall time spent doing *Send Request* can be calculated by multiplying this by the transition throughput for *Send Request*.

Refer to Section 4.2.3 for details on how DNAmaca calculates transition throughput.

Enabled Probability

The enabled probability of a transition gives the probability of finding that the transition is enabled at steady state.

If the enabled probability of a transition is low, then the chances of that transition firing in a certain time interval are, of course, low. However, if the enabled probability of a transition is high, it does not necessarily mean that the transition fires often. This is due to the fact that if more than one transition is enabled at a particular point in time, the transitions with the higher rates are more likely to fire than those with lower rates. So, the enabled probability of a transition gives an indication of how often a transition is ready to fire, but the transition throughput must be examined to see how often the transition actually fired.

Mean number of tokens

The mean number of tokens on a place is the average number of tokens that can be expected to be found on a place in steady state.

An example of where this could be a useful performance measure is in the case of the readers/writers problem shown in Figure 10.6. By adjusting the rates of the *Begin Reading* and *Begin Writing* transitions, the effects of this can be seen on the mean number of tokens on the places. The mean number of tokens on *Writing* cannot be allowed to grow too large, since a writer obtains an exclusive lock on a resource, which means that the readers will all be idle when a writer is writing. The impact of the exclusive lock can be seen by the fact that relatively small changes in the mean number of *Writers* results in a relatively large change in the number of *Readers*. The performance implications of this are obvious, since a large number of idle readers (or writers) could mean a drastic decrease in performance.

Mean distribution

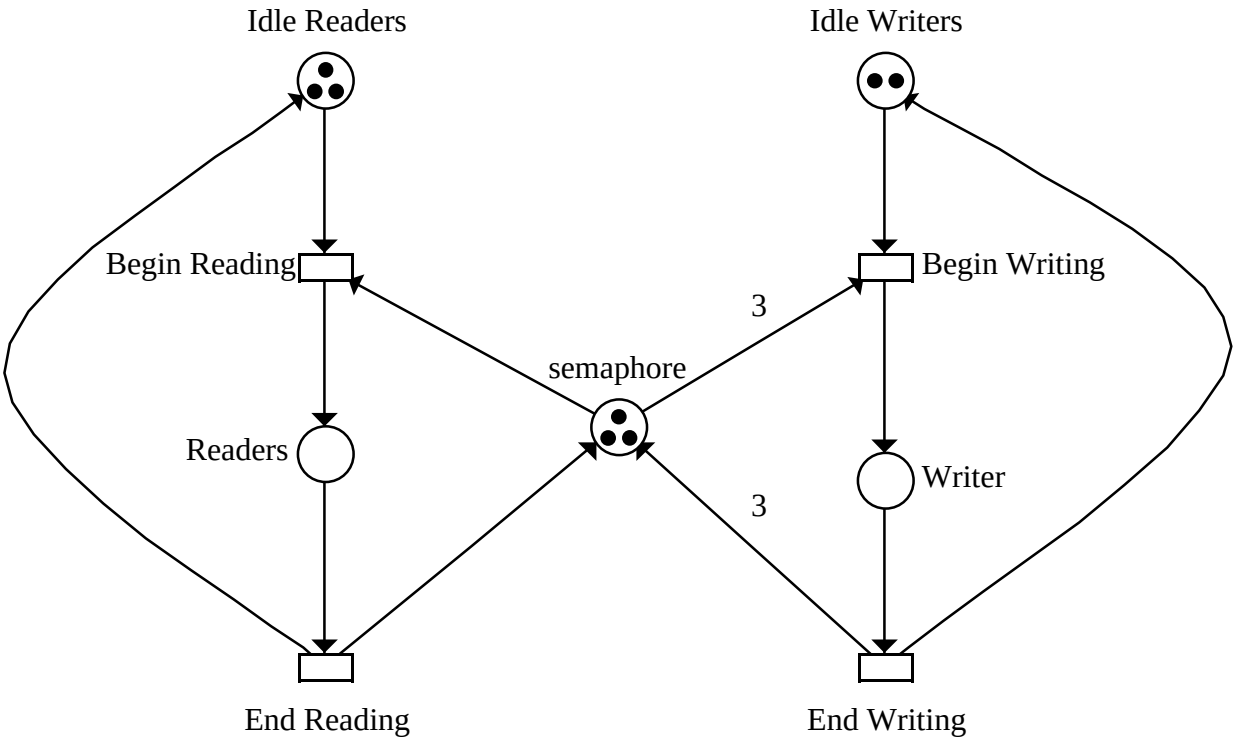
Mean distribution indicates the mean distribution of tokens on a place at steady state. More precisely, this refers to the probability of finding a certain number of tokens on a particular place at any given time.

Since all places must be bounded (else steady state analysis is not possible), a maximum number of tokens for each place can be found. This means that at every state in the Markov chain, a finite number of tokens will be found on every place. As an example of distribution results, consider a place which has a maximum of three tokens on it:

```
 Distribution of tokens for place P1:
0 5.00000000e-01
1 2.00000000e-01
2 2.00000000e-01
3 1.00000000e-01
```

The distribution results show:

- the probability of finding zero tokens on P_1 is 0.5
- the probability of finding one token on P_1 is 0.2
- the probability of finding two tokens on P_1 is 0.2
- the probability of finding three tokens on P_1 is 0.1


Idle Readers
Idle Writers
Begin Reading
Begin Writing
3
semaphore
Readers
Writer
3
End Reading
End Writing

Chapter 11

Graphical User Interface

11.1 Overview

We have seen that there is a definite need for Petri net editors. The reason for this is that Petri nets are very difficult to analyse by hand when the model is a realistic size. We need tools to automatically analyse our nets for correctness and performance.

The analysis of Petri nets is fairly mechanical and the task is well suited to solution by computational methods. So, we need some means in which to present the information to the computer to serve two goals:

- The representation must be easily understandable to humans and,
- The machine must be able analyse the net.

The first of these two goals is clearly the most important as Petri nets are not only used to solve concurrency problems, but also to facilitate communication between theoreticians and practitioners. The simplest and most obvious way of meeting these two criteria is by representing the Petri net in its standard visual form to the user and then converting that to a representation that facilitates analysis by the machine.

In this chapter we discuss our approach to meeting the first of the above two criteria.

11.2 Creating and Editing Petri nets

Creating and editing the Petri nets should be quick and simple. Additionally, the tool for designing the Petri net should be as user-friendly and flexible to cater for all users needs. To this end several features were implemented in DaNAMiCS. The basic methodology is for the user to select a tool from the toolbar and then use that tool for the desired effect.

Inserting places, transitions and subnets is simple and they are given default, unique names to save the user the trouble of naming each component. For flexibility, the user can rename all the components on the screen. Removing inserted components is equally simple and the usual options to *cut*, *copy*, and *paste* segments of the Petri net are available. These functions are conservative, which means that they only cut or copy sections of the net which can be logically removed. For example, if the user selects some place which has a connected

transition, but not that transition itself, then the only object cut or copied is the place itself. The arc will only be copied if both the place and transition connected by that arc selected.

DaNAMiCS uses *WYSIWYG* [1] for the placement of components in the net. For example, when the user draws an arc from a place to a transition or vice versa, the technique known as *rubber-banding* is used to show that arc before it is actually placed. Rubber-banding reminds the user in an intuitive manner what the source component of the arc is. Thus fewer editing errors are made. This feature was not available in DNAnet.

Additionally, when a component is moved to another location, its position in transit is shown, so that the user knows beforehand what the net will look like after the component is placed. A screen-shot of the DaNAMiCS interface with a dialog enabled can be seen in Figure 11.1

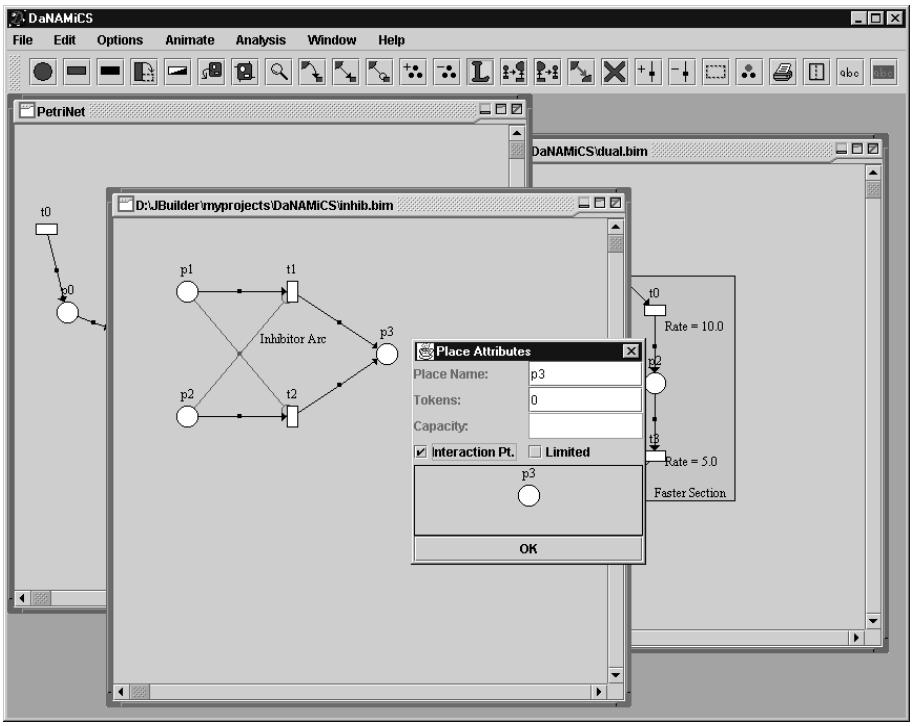


Figure 11.1: A DaNAMiCS ScreenShot

DaNAMiCS includes two visual aids to enhance the graphical appearance of the net. These are *highlights* and *comments*. A highlight is simply a box, the colour of which can be selected, which is placed around an object or group of objects. A comment is a line of text which can be used for clarification of the functionality of a certain section of the net. Neither of these features were present in DNAnet, or in the original DaNAMiCS specification, but they have proved an invaluable addition to the application. DaNAMiCS will not only be useful as an analysis tool, but also as a tool for the presentation of Petri nets, which encompasses the two functions for which Petri nets were, at first, created.

[1] What You See Is What You Get

11.3 Interactive Animation

The user should have the ability to test the dynamic behaviour of the nets to check if they accurately model the system. This can be done by allowing the designer to choose which transition to fire. In the literature, this is known as the *Token-game* and it involves firing enabled transitions and examining the propagation of tokens through the net.

DaNAMiCS provides the functionality of the token game in the form of animating the net. Once the user has opted to animate the net, all the enabled transitions in the initial marking of the net are found and their colour changed to indicate that they are enabled. The user can then click on one of the enabled transitions to fire it, and the new marking of the net is calculated and displayed.

The dynamic behaviour of Petri nets can be very divergent. Firing one transition and not another could lead to a very different path of enabled transitions flowing through the net. In DaNAMiCS the ability to *step back* along the path of fired transitions and *unfire* them has been added since it was not present in DNAnet. This reduces the time taken for a user to test a net for its basic functional correctness.

11.4 Drawing the Petri net

A Petri net is drawn onto a *PetriCanvas*. The *PetriCanvas* is an intelligent object that handles all mouse interaction with itself. It also takes care of the displaying and printing of the Petri net. The Petri net needs to be redrawn frequently. Thus the redraw methods are optimal.

The Petri net is stored as a linked lists of places, and a linked list of transitions. Each transition contains a list of incoming arcs and a list of outgoing arcs. Each place, transition, arc and subnet stores a bounding box associated with it. The bounding box of a transition includes the bounding box of all the arcs that enter or leave that transition.

When anything changes, the smallest rectangular bounding box that covers the change is calculated. This bounding box is then passed through to the drawing method. All places and transitions are then checked to determine if their bounding box intersects the changed area. If they do intersect then that object redraws itself. If the object is a transition it then draws all the arcs from that transition if their respective bounding boxes intersect the redraw area.

Having these bounding boxes associated with each object in the Petri net speeds up the drawing. This speed up allows for realtime interaction with the drawing canvas.

All *PetriUnits* are completely scalable. A *PetriUnit* contains a method for drawing the label of the object. This label is included in the bounding box of the object. These labels can be displayed in one of 9 different positions. These positions are illustrated in Figure 11.2.

All *PetriUnits* can be selected. The selection is indicated by drawing a coloured area behind the *PetriUnit*.

11.4.1 Drawing a place

A place is drawn as a circle with a dot for each token on that place. If the number of token exceeds five then the number of tokens is displayed as a number.

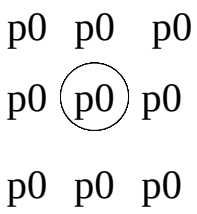


Figure 11.2: Positions where a PetriUnit can be labeled.

11.4.2 Drawing a transition

A transition is displayed as a rectangle. The orientation of the transition can either be vertical or horizontal. A transition is drawn in white if it is a timed transition and in black if it is an immediate transition. If the net is in animate mode and it is enabled then it is drawn in green or orange for timed and immediate transitions respectively.

11.4.3 Drawing an arc

An arc contains a list of handles. The arc is then drawn as either a spline or a straight line through these points. An arrow is drawn on the end of normal arcs indicating a direction for the arc. If the arc is an inhibitor arc then it is drawn in red, and a circle is drawn on the transition instead of an arrow.

11.4.4 Drawing a subnet

A subnet is displayed as a large rectangle. The arcs to and from places or transitions in the subnet are then drawn from the subnet.

11.4.5 Drawing a comment

Comments are a feature that we added to allow the user to label components of the Petri net. These labels are drawn as text to the screen. They can still be manipulated the same way as other *PetriUnits*.

11.4.6 Drawing a highlight

These are drawn as solid rectangles in any color. They are drawn behind all other *PetriUnits* to highlight groupings in the net on the visual display.

11.5 Results Viewer

DaNAMiCS provides a results viewer so that the results of any analysis can be easily obtained. Previous Petri net editors have concerned themselves solely with the editing aspect of Petri net design and not with

the more fundamental purpose of the original Petri net, which is to analyse modeled protocols and systems for correctness and performance.

DaNAMiCS attempts to remedy this by making the results of any analysis easily obtainable and readily understandable. We are careful to not overload the user with information by allowing him or her to select the desired information. The results viewer can be divided into three sections.

11.5.1 Invariant Results

Invariant analysis is a mathematical solution of the net. The results provided cannot be modified from their mathematical form easily. We thus present the results in a concise, clear manner.

Before the invariant analysis is performed, the user is prompted for information on what results he or she is interested in. All of the results described below are optional, depending on what the user required.

In the results viewer for the invariants, shown in Figure 11.3, there are six panes. The top two panes give a quick synopsis of the results of the invariant analysis. They tell the user whether the net is covered by P- or T-Invariants, and if not, inform the user as to which places or transitions are not covered.

The following two panes show the actual P- and T-Invariant vectors if required for the user's own analysis. The final two panes show respectively the incidence matrix if requested, and the generated P-Invariant equations.

The P-Invariant equations are automatically generated by DaNAMiCS. These equations are possibly the most useful aspect of the invariant analysis. They tell the user which places of the net always have a constant number of places on them. The equations concisely tell the user which places are involved and what the constant number of tokens is given the initial marking of the net.

An example of the Invariant results viewer is shown in Figure 11.3. In the panes containing the matrices, or the invariants, a list of the orders in which the places or transitions appear is provided.

11.5.2 Coverability Results

Once again, as with invariant analysis, the user is prompted to indicate which correctness properties he or she is interested in. All of the text and graphical panes described below can be effectively turned on or off, and the analysis required to fill these panes is not performed. This has the two-fold purpose of increasing efficiency and allowing the user to select how much information should be shown at one time.

The coverability results pane shown in Figure 11.4 consists of five sub-panes. The top four panes present the coverability results textually while the bottom pane shows a graphical display of the results.

The first three panes down the left-hand side show a synopsis of the results of the analysis. The first of these panes contains information on the liveness of the net. If the net can ever deadlock, the sequence of transitions leading to this deadlock is printed. If certain transitions in the net can never be fired once the marking is within a certain final component, then the defective component is printed. Along with this, the list of transitions that are enabled and not enabled in this component are shown.

The second pane on the left displays the Petri net's boundedness information. If the net contains unbounded, or *possibly* [2] unbounded places then these places are listed here.

[2] Refer to the section on coverability graph analysis

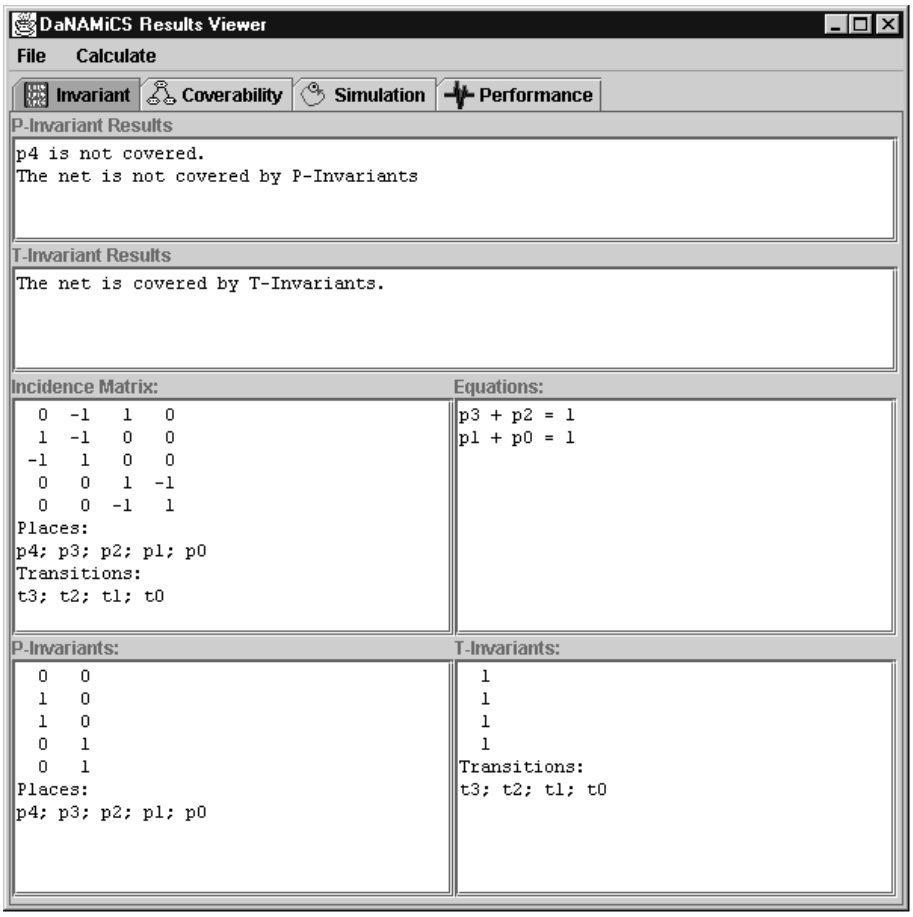


Figure 11.3: An Invariant Results ScreenShot

The final pane displays any general information the user may have expressed interest in. There are three possible results that can be printed here. The first is an indication as to whether the net contains home states, the second reveals whether or not the net is safe. Finally, if the user requested persistence analysis, then these results are shown here. If the net is not persistent then the markings where the firing of one transition disables another are shown along with the transitions that disable each other.

The top right pane, if the user opted to have the coverability graph shown, shows an adjacency list of the coverability graph. In this listing, the component numbers of the nodes in the coverability graph are shown, so that, if required, the liveness of the net, or or whether it contains home-states, can be calculated by hand. In addition, the component numbers are useful for cases where the graph contains final components which do not enable all the transitions in the net. We feel that the adjacency list is the most intuitive way to show a graph structure since drawing it graphically would be beyond the scope of this project.

The bottom pane contains the graphical representation of the coverability results. This allows the user to see the results at a glance. The graphical display of the coverability analysis results has several important features. Firstly, all unbounded places are circled in red. This alerts the user immediately to the incorrect areas of the net. Secondly, if the net contains a deadlock, an animation is shown of how the net can become deadlocked by firing the sequence of transitions which lead to the deadlock. Finally, if the net contains unbounded places, the sequence of transitions which, when repeated, lead to these places becoming unbounded

is shown graphically by means of animation.

During these animated sequences, which are suitably delayed so that the users can actually follow it, the transition currently being fired is highlighted briefly in red. If ever the sequence of transitions to be fired leads into a subnet, then the focus is automatically shifted to that subnet to show the details which would be otherwise hidden.

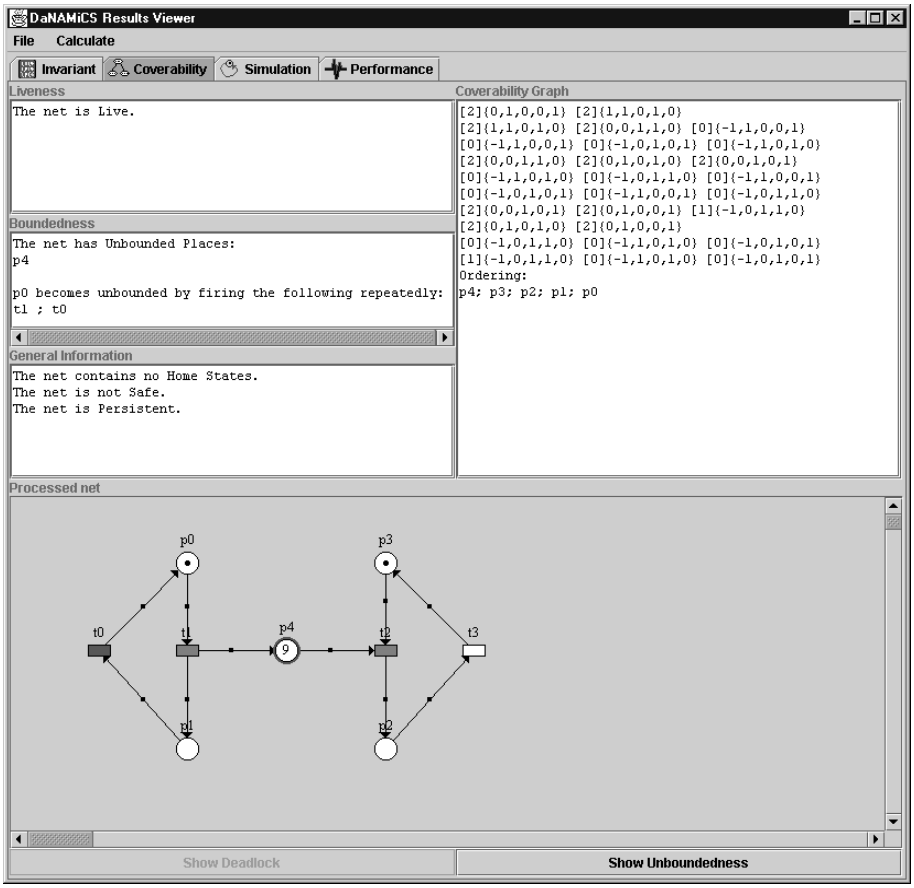


Figure 11.4: A Coverability Results ScreenShot

11.5.3 Performance Results

The performance results consists of the simulation and steady state analysis results which are very similar. The results are simply presented textually with the text panes for each metric. In other words, their is a pane listing the mean number of tokens per place, and a pane for the transition throughput. The steady state analysis has one additional pane for the distribution of the results, and the simulation results pane also shows the length of time for which the simulation ran. A screenshot of the results for the performance analysis can be seen in Figure 11.5

Since the performance results are handled externally, the returned results are difficult to analyse, and thus their textual representation is simply shown. For simulation, however, we can make intelligent assumptions about the net and show the results in a more intuitive fashion.

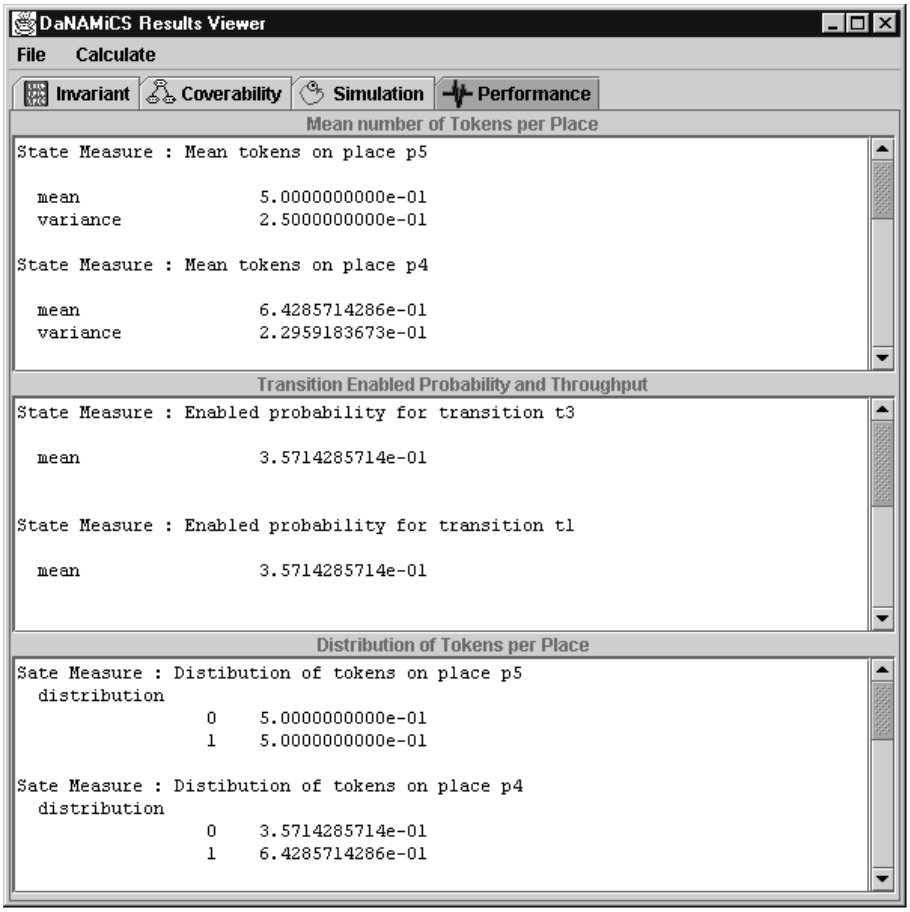


Figure 11.5: Performance Results ScreenShot

To this end we present another view of the net in a pane below the textual results. Next to each place and transitions name, we display the metrics calculated for that place or transition. This allows the user to quickly obtain the information relevant to those units in which he or she is interested.

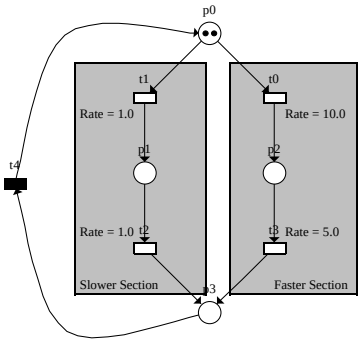


Figure 11.6: A simple Petri net

A net modeling a concurrent system is shown in Figure 11.6. The simulation results viewer for this net can be seen in Figure 11.7.

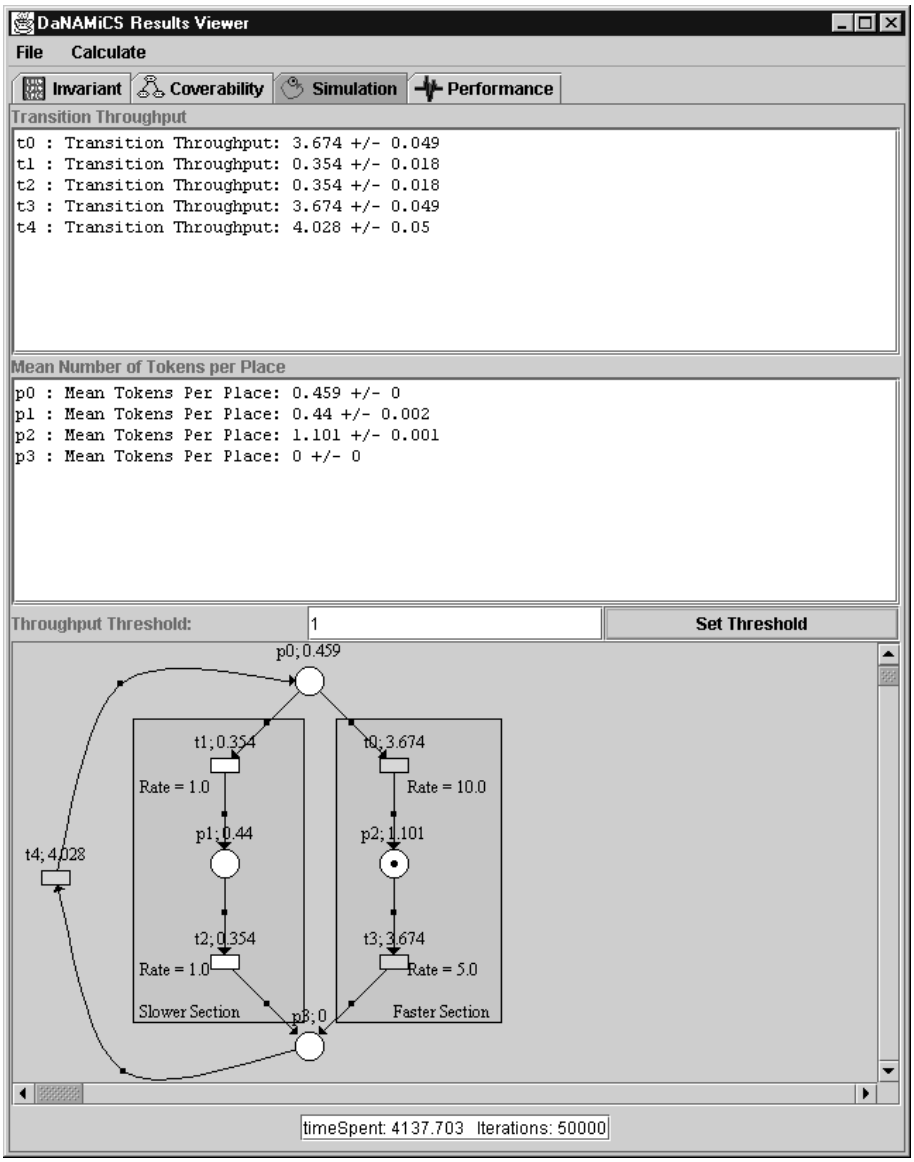


Figure 11.7: The DaNAMiCS Simulation Results Viewer

Additionally, we present a data-filter tool for examining the transition throughputs. Users may wish to impose a condition on their nets that each transition has a certain minimum throughput. This we call the threshold throughput. The users may set this threshold, and then, by clicking the appropriate button, the transitions which have throughput *above* the threshold throughput are highlighted in yellow. Transitions which do not satisfy the condition are not highlighted. This is an intuitive mechanism to speed up the analysis of the simulation results.

In addition to the thresholds and the display of the appropriate data in close proximity to the actual component it describes, we also show the mean number of tokens per place in an intuitive fashion. The marking of the displayed net is set to rounded values of the actual mean number of tokens per place. This vastly

enhances the usability of the tool and one can instantly observe the results of the simulation.

11.5.4 Saving and Loading

Any results obtained in DaNAMiCS can be saved to disk. This is so that the user can make use of them at a later stage and compare them with new results. It is important to remember that multiple copies of the results viewer can be opened simultaneously and thus comparison of results is greatly simplified. The results are saved in a standard format, similar to latex and a grammar is provided in the Appendix such that new or other programs will be able to make use of the results files.

Chapter 12

Coloured Tokens

12.1 Overview

Coloured Petri Nets (CPNs) provide a higher level of abstraction than ordinary Petri nets. With each token in the net we now associate a "colour". This colour can be thought of as the tokens *type* or *category*. Coloured tokens reduce the model size thus making the models easier to create and maintain.

The use of coloured tokens results in a more compact representation of the system. In ordinary Petri nets, we require a place for each possible state or value in the system. Now, with coloured tokens, we can model these different types with tokens of differing colours.

A place is no longer needed to model each low-level variable, but rather a place could be used to store a collection of different types in it. This is similar to the concept of a *Record* in Pascal or a *struct* in C. The comparison here is that it is easier to think in terms of the abstracted grouping. In developing a program it is easier to design it at a high level and convert it to the underlying bit representation. This is somewhat similar to how a CPN can be *unfolded* to a Petri net. In this way the model can be developed at an abstract level.

In DaNAMiCS we provide the use of a subset of the Coloured Petri net class in the form of coloured tokens. We will discuss our approach to solving the problems imposed by coloured tokens, and elaborate on our implementation of the coloured token class.

12.2 An Introduction

In this section we present an introduction to the use of coloured tokens and the basic functionality that DaNAMiCS provides. The features implemented in DaNAMiCS are novel, and thus novel means had to be devised to support the graphical and analytical use of coloured tokens. In this section we will provide and example of coloured tokens to illustrate their use, and then describe features DaNAMiCS supports.

12.2.1 An Example

Assume we wish to model a manufacturing process where we build widgets with nails and screws. Let us assume their are many different types of widgets that can be produced, each requiring a different combination of screws and nails. The ordinary Petri net for this is shown in Figure 12.1. We see there are two

input places, one for the nails and one for the screws, which serve three different transitions, which are the different manufacturing processes. Each of these three transitions has an output arc to a different place, one for each of the possible types of widgets that can be made.

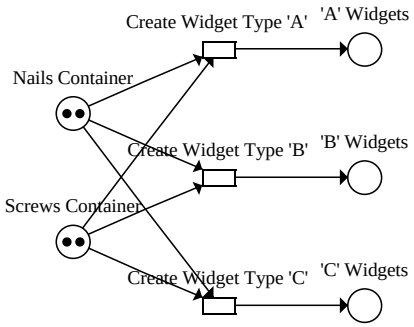


Figure 12.1: An Ordinary Petri net.

With coloured tokens, this net could be greatly simplified. Each type of widget and building component could be modeled with a different colour token. Figure 12.2 shows the CPN for the system. Instead of having two places, we now have one place with two types of coloured tokens in it. Instead of having three output places, we have a single output place with three different colours in it. Accordingly, we no longer need three transitions for each of the possible processes. We now have one transition which can fire in different *modes*, described in the next section.

12.2.2 Features in DaNAMiCS

Each of the modes, introduced above, relates to some real-world action and specifies the number and combination of coloured tokens to be consumed and produced by the transitions. In DaNAMiCS, for simplicity, we specify these modes on the arcs leading from and to transitions. Then, when firing a transition, we select which combination of the possible input and output modes we wish to fire. Note, that only certain modes will be enabled at any particular time.

The modes are specified as logical operations and, in DaNAMiCS, we support the logical operations of *and*, ($\wedge$) and *or*, ($\vee$). These operations were supported because of their ease of translation to an ordinary Petri net and that they allow a wide range of systems to be modeled.

In DaNAMiCS we provide three colours: red, green, and blue. This is not a real limitation, and can be extended if necessary. The interesting point about coloured tokens is that individual colours can have different symbolic meanings depending on which place the coloured tokens are currently in. So, only allowing three different colours does not drastically reduce the modeling possibilities.

Finally, it should be stressed that all of the original Petri net constructs are still available. We can still inhibit transitions with inhibitor arcs and place capacities on the number of places in the net. Inhibitor arcs have the same function as in ordinary nets, and can be instructed to inhibit specific colours on a place. This adds the *not*, ($\neg$) to our coloured token logic. This allows us the complete use of first order logic expressions on

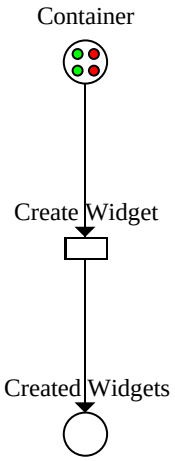


Figure 12.2: The Coloured Petri net of the net in Figure 12.1.

our arcs which greatly enhances our modeling and abstraction power with coloured tokens. Additionally, we can set firing rates for each of a transitions' modes. This gives us the full functionality of GSPNs as well as the modeling power of coloured tokens.

12.3 Theory

In this section we formally define a Coloured Petri net. The accepted CPN definition is presented. Which is now an accepted Petri net formalism.

Definition 12.1 - A Coloured Petri Net [14]
A Coloured Petri Net is a tuple CPN = $(\Sigma, P, T, A, \tau, G, E, I)$ where,

- Σ = a finite set of non-empty types, called *colour sets*.
- P = a finite set of *places*.
- T = a finite set of *transitions* with $P \cap T = \emptyset$
- A = a finite set of *arcs* such that $A \subseteq P \times T \cup T \times P$ and $A \cap (P \cup T) = \emptyset$
- τ = a *colour* function, $\tau : P \rightarrow \Sigma$
  where $\tau(p) = C^*$ for $p \in P$ and $C^* \in \Sigma$
- G = a *guard* function, $G : T \rightarrow expr$
  where $\forall t \in T : [Type(G(y)) = bool \wedge Type(Var(G(t))) \subseteq \Sigma]$.

- E = an *arc expression* function, $E : P \times T \cup T \times P \rightarrow expr$
  where $E(x_1, x_2) = \emptyset$ if $(x_1, x_2) \notin A$
  and $\forall a \in A : [Type(E(a)) = \tau(p(a)) \wedge Type(Var(E(a))) \subseteq \Sigma]$
  where $p(a)$ is the place of arc a.

- I = an *initialisation* function, $I : P \rightarrow expr$
  where $I(p)$ is a closed expression (i.e. it contains no free variables)
  and $\forall p \in P : [Type(I(p)) = \tau(p)]$.

The places hold coloured tokens, and the distribution of these tokens around the net defines the state or marking of the net. Transitions change the state, while arcs are used to link the transitions and places together. The guard function of a transition evaluates to true when the transition is enabled. This guard function can be set to any logical expression on the places linked to the transition. In DaNAMiCS, as we have mentioned, we provide a subset of the above class, called *Coloured Tokens*. This is so because DaNAMiCS does not allow the guard function to be any arbitrary function, which is the case in full CPNS, but is limited to the elements of first-order logic: *and*, *or*, and *not*. This is not a real limitation on the modeling power, but certain systems will be more difficult to model as the expressions on the arcs will be more involved. Full CPNs allow entire first-order logic programs to be placed on the individual arcs, which is a great modeling advance.

12.4 Implementation

As can be seen in Figure 7.5 a new set of objects has been inherited through a *CPetriUnit interface*. These objects behave slightly differently from superclasses. They all have the added functionality of being able to unfold themselves into an ordinary Petri net.

The first change in the structure of these classes is to the *coloured place*. This place no longer stores the initial, maximum and current number of tokens. Rather for each colour it needs to store these values. These are stored in arrays, whose length is the number of colours.

The arcs needed to be upgraded so those different modes could pass through them. These different modes each specify an amount of each of the colours which is to be produced or consumed. The modes are specified in terms of a *Mode* class which defines the number of each of the colours and can determine whether a specific mode is enabled.

The transitions contain multiple modes. At least one of these modes needs to be evaluated to true for the transition to be enabled. These modes link the modes on the incoming arcs and the modes on the outgoing arcs together, thereby defining the specific possibilities of actions that the transition could perform.

This is explained in Figure 12.3. We show a transition with an incoming and an outgoing arc. The user has defined several modes on each of the arcs, and in the transition, has linked these modes together. So, when the transition fires, the option is to consume tokens in modes one or two, but tokens can only be produced in mode four. Spreading the modes to the arcs seemed the simplest means of specifying the actions of the transitions otherwise, if we left everything to the transition, the analysis would be much more complex, and the user-interaction would be very cumbersome.

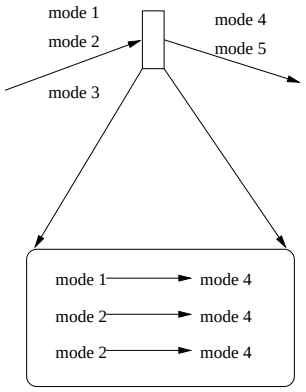


Figure 12.3: Transition modes explained.

12.5 User Interface

All the components in a Petri net have dialogs associated with them that allow user interaction and control. These had to be replaced with more complicated dialogs. It was a challenge deciding on an intuitive system for this interaction.

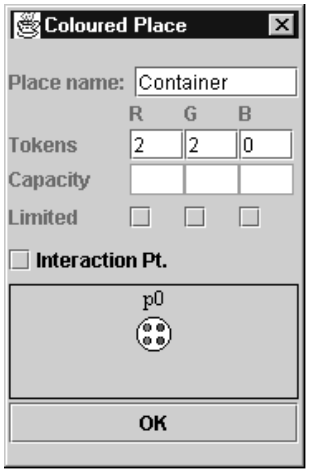


Figure 12.4: The dialog box for a coloured place.

The first dialog that we needed to change was for coloured places. This dialog is illustrated in Figure 12.4. It has three columns to allow the user to specify the number of tokens of each of the colours on the place. Additionally, the place can have a limit imposed on the number of tokens of each colour present in that place.

The second dialog gives the user control over how the arcs behave. This dialog is shown in Figure 12.5. Modes can be added, edited and removed. A mode on the arc is a possible combination of coloured tokens on the place that would be consumed or produced if the transition where fired. When an arc is drawn it starts, by default, with three modes. Each of these modes consumes or produces one red, green, or blue token. As seen in Figure 12.5, the user can add or remove modes on an arc at will, and specify the constraints of each mode.

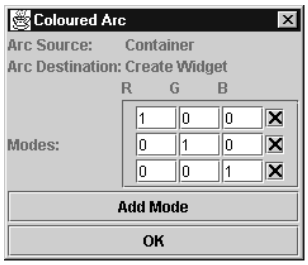


Figure 12.5: The dialog box for a coloured arc.

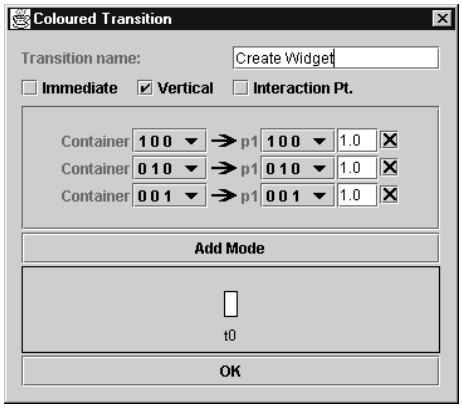


Figure 12.6: The dialog box for a coloured transition.

The third enhanced dialog is shown in Figure 12.6. This dialog allows the user to group together modes from incoming and outgoing arcs. These modes can be deleted, created and edited. The behaviour for an arc in a specific firing of the transition is defined by these modes. This behaviour is chosen from the modes defined in an arc.

From all the possible firing modes, the user can select as many as he or she wishes. A subset of these modes will be enabled depending on the number of tokens in the source places, and the capacities of the destination places. We should remember here that all the original features of Petri nets are still present, such as inhibitor arcs, capacities and timed transitions. Thus, rates can be specified for each of the firing modes for use in simulation and we may also specify immediate transitions.

12.6 Analysis

The analysis of coloured Petri nets can be done in two different ways. It can be done directly on the CPN [6], or the CPN can be unfolded into a Petri net and the analysis carried out on that Petri net.

In DaNAMiCS we choose to unfold the CPN before analysis. The unfolding of the CPN creates three places for each coloured place. These places are offset from each other in the unfolded net. They are named after their original place with an ’a’, ’b’, or ’c’ appended after the name to imply that this place originated from the red, green, or blue tokens.

The moving of the places and changing their name allows the unfolded net to be displayed to the user. Each mode in a transition unfolds to a transition. These unfolded transitions are also offset in the x and

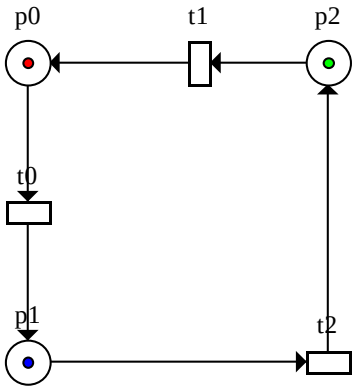


Figure 12.7: A Coloured Petri net.

y directions, and have a letter appended after the name to allow easier user recognition. A net containing coloured tokens is found in Figure 12.7, and the net unfolded by DaNAMiCS is shown in Figure 12.8. We see that there are unconnected places which have no effect on the analysis but are generated automatically by the unfolding algorithm. Only the necessary transitions are generated, however, corresponding to each possible mode.

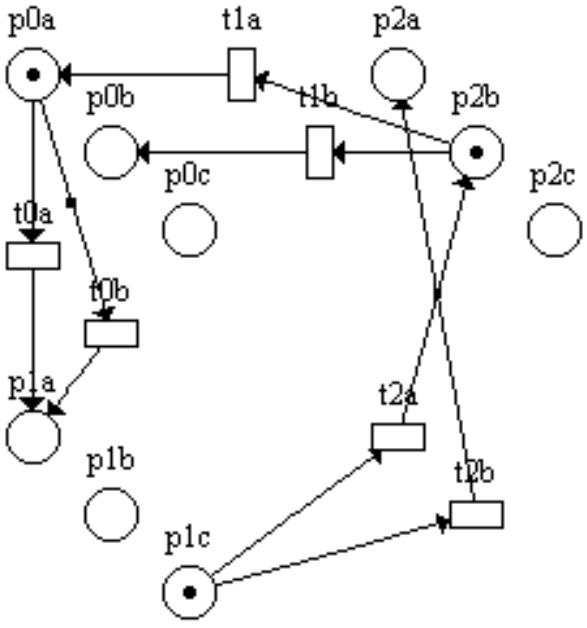


Figure 12.8: The net in Figure 12.7 unfolded by DaNAMiCS.

The algorithm to unfold a Petri net is shown in Figure 12.9:

begin unFold $(CPN(P, T, I^-, I^+, M_0))$
$\forall p \in P, c \in C(p)$ create a place (p, c) in the new Ordinary Petri net
$\forall t \in T, c' \in C(t)$ create a transition (t, c') in the new Ordinary Petri net
set the arc $I^-((p, c)(t, c'))$ to weight $I^-(p, t)(c')(c)$
set the arc $I^+((p, c)(t, c'))$ to weight $I^+(p, t)(c')(c)$
set the initial marking $M_0(p, c) = M_0(p)(c), \forall p \in P, c \in C(p)$
end

Figure 12.9: The Algorithm for unfolding CPNs.

The choice to unfold the net before analysis was made so that we could reuse the code that performed the correctness analysis. This code was specific to the Petri nets without coloured tokens. DNAmaca would need the net to be unfolded for DaNAMiCS to pass the description vector of the net through to it. The only exception when we do not unfold the net is when we are providing interactive animation for the user.

12.7 Animation

The user driven animation of the coloured net is similar to that of ordinary nets. The user selects an enabled transition. If there is only one enabled mode in that transition then that mode is fired automatically. If multiple modes are enabled then the user needs to select which of the modes he or she would like it to fire in. This dialog is shown in Figure 12.10.

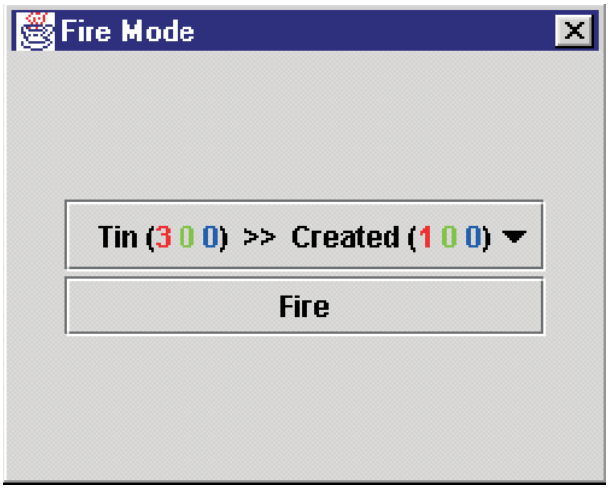


Figure 12.10: The section of a mode in an enabled transition.

The dialog to select an enabled transition from displays all the enabled modes. The user selects the desired mode which is then fired on the user's command. The net is then updated to reflect the changes caused by firing this transition in this mode.

12.8 Future work

There are many interesting improvements that could be made to the implementation of Coloured Petri nets in DaNAMiCS. These would all allow the model to be designed on a level closer to the real-world problem.

The first and most obvious extension would be to allow the guard function to be any function. The action of this transition need not coincide with the enabling function. Thus for example a transition could be enabled with three blue tokens on place p_1 but only consume two of the blue tokens. This would require the development of a concise easily understandable notation for defining these expressions. The point of CPNs is to make specification of the Petri net easier. These extensions do have the potential for making the specification of the model unwieldy and difficult to manipulate.

These extensions, however, do exceed the scope of the DaNAMiCS project, but as we have mentioned, DaNAMiCS has been designed expressly to allow these kinds of extensions.

Chapter 13

Testing

13.1 Overview

Testing is an important part of program development process because it provides:

- a means to find errors in the program implementation. Regular testing throughout the development leads to a system which is more robust and is functionally correct.
- feedback which guides the development process. If end users are involved with program testing during the development process, their feedback will ensure that the end system meets their needs. Getting potential users of DaNAMiCS involved in program testing was a very important aspect of the DaNAMiCS project.

13.2 Functional testing

Black-box or functional testing, tests a system's capabilities. This is important, since the capabilities of the system define its usefulness, and its limitations, to end-users.

Functional testing is crucial for a modeling and analysis tool such as DaNAMiCS. The analysis of the models must be correct, since these results are used to guide the implementation of real systems. For example, suppose that a communication protocol was analysed. If the analysis function incorrectly returned that the protocol was deadlock-free, this would have serious ramifications when the protocol was actually implemented for use in a distributed environment.

In order to carry out functional analysis of a system, a set of inputs and the corresponding set of outputs must be known. When testing the analysis functions of DaNAMiCS, the correct results could sometimes be obtained by analysis using DNAnet. Clearly, with the extended class of Petri nets that DaNAMiCS could handle, this was often not possible. In these cases, the results had to be obtained manually, before comparing them with the analysis results from DaNAMiCS. Additionally, some results were compared with those published in papers on Petri net theory.

13.2.1 Testing Invariant analysis

The invariant algorithm, described in Chapter 10, consists of two phases. The first of the two phases is so efficient that the second phase is rarely even executed. We had to select our test nets carefully in order to ensure that the entire algorithm was tested.

The algorithm was first tested on a basic selection of nets and the results compared with those produced by DNAnet. For example, the classic problem of the readers and writers was tested. The Petri net for the solution to the Readers/Writers problem is shown in Figure 13.1.

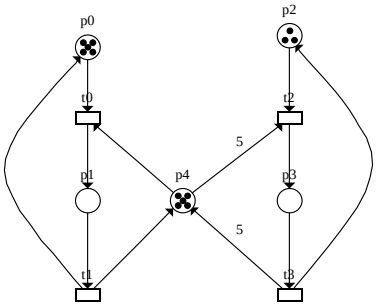


Figure 13.1: The Readers/Writers problem to be solved by Invariant analysis.

The net in Figure 13.1 is clearly covered by P- and T-Invariants. The results of the analysis can be seen in the DaNAMiCS screen-shot in Figure 13.2. We see that DaNAMiCS correctly solves the net by finding that it is covered by P- and T-Invariants. Notice also, that the equations generated automatically by DaNAMiCS are correct. This net was not sufficient to test the invariants, and thus a wide selection of nets were solved, and the results proved correct by DNAnet, and sometimes by hand.

Figure 13.2: Screen-shot of the results of the Petri net in Figure 13.1.

Since DaNAMiCS includes inhibitor arcs, we needed to test that these additions did not flaw the invariant analysis. In Figure 13.3 we find the solution to the Readers/Writers problem using inhibitor arcs. DaNAMiCS correctly finds this net to be covered by positive P- and T-Invariants.

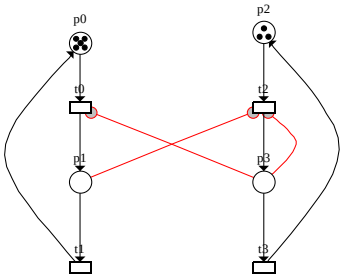


Figure 13.3: Inhibitor arcs, solved with Invariant analysis.

13.2.2 Testing Coverability analysis

The coverability graph generation algorithm in DaNAMiCS was completely new, and thus very thorough testing was required. There were few previous results available to test DaNAMiCS against, thus the results had to be tested by hand. In addition to this, DaNAMiCS can produce many different categories of results. We will not provide exhaustive examples of these here, but we will list them, and highlight important issues.

Firstly, if a net is not live, then either it contains a deadlock, or its coverability graph contains final components which cannot enable all the transitions. In the first case, DaNAMiCS prints out the sequence of transitions which lead to the deadlock. As an example consider the simple Petri net in Figure 13.4. This net clearly contains a deadlock and DaNAMiCS prints:

```
The net is not live,
It contains a deadlock via the following sequence:
```
t_0, t_1, t_2.

which is correct.

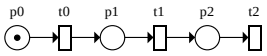


Figure 13.4: A net which can only deadlock.

In the case where the net contains final components which cannot enable all the transitions, as in the net in Figure 13.5, DaNAMiCS yields the following correct information:

```
The net is not live,
Final component 1 can enable:
```
t_2; t_3
```
but not:
```
t_4; t_5; t_6; t_7.

The net in Figure 13.5 clearly contains a livelock, which the results reflect. The heuristics to create this and other nets were obtained from an understanding of Petri nets and the functioning of the algorithm. We tried to create nets that would test all possible results.

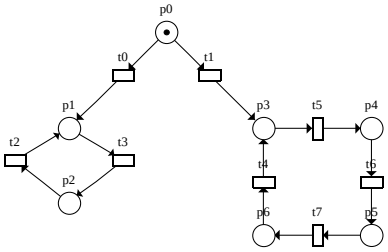


Figure 13.5: A net caught in livelock.

If the net is unbounded, DaNAMiCS yields information on which sequence of transitions, which, when repeated, will cause a place to become unbounded. As an example, the net in Figure 13.6 is clearly unbounded with place p_3 being defective. DaNAMiCS displays the following results:

The net has unbounded Places:
p_3
p_3 becomes unbounded by firing:
t_0
and then firing the following repeatedly:
t_1, t_2, t_3.

which is clearly correct. Heuristics were also used to create this example.

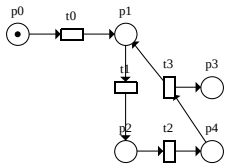


Figure 13.6: p_3 in unbounded in this net.

Since the addition of inhibitor arcs directly affects places becoming unbounded, we will show an example of where DaNAMiCS is unable to decide whether a place is unbounded or not. In this case, DaNAMiCS returns that the suspected places are *possibly unbounded*. The example in Figure 13.7 shows such a net. The results returned by DaNAMiCS, as expected, are:

The net has possibly unbounded places,
The possibly unbounded places are:
p_2; p_0

Note that DaNAMiCS did not find p_1 as possibly unbounded, this is because the algorithm terminated before it could detect this places. It is important to remember that this was implemented to ensure that DaNAMiCS never makes incorrect guesses about the correctness of the net. Tests were also made to ensure

that the cases where DaNAMiCS stops only when chains in the coverability graph reach a certain length, or where it stops only when it runs out of memory, were correctly performed.

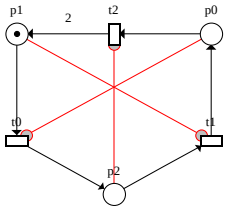


Figure 13.7: A net with possibly unbounded places.

Tests on the existence of home states were made and the results compared with that of DNAnet. Finally, the two new features added, safeness and persistence, had to be tested based on heuristics and results found in the literature [16]. The net shown in Figure 13.8 is clearly safe, and DaNAMiCS prints the following results:

```
The net is Safe.
```

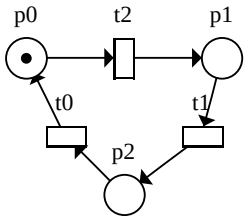


Figure 13.8: A safe net.

DaNAMiCS also gives information of the persistence of the net. The net shown in Figure 13.9 is taken from [16] and the net is persistent which is correctly decided by DaNAMiCS. The net shown in Figure 13.10 is clearly not persistent and DaNAMiCS prints the following results:

```
The net is not Persistent
Firing transition t_0 in marking (100) disables the following transitions:
t_1
Firing transition t_1 in marking (100) disables the following transitions:
t_0
```

Thus we see that DaNAMiCS consistently gives the correct results as it as been tested on a wide variety

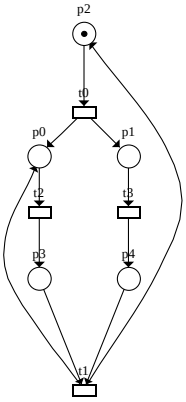


Figure 13.9: A persistent net.

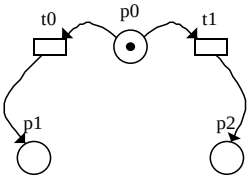


Figure 13.10: This net is not persistent.

of Petri nets. Certain errors were reported by users in this area which illuminated logical flaws in the algorithms. Most testing, however, was based on literature and heuristics.

13.2.3 Testing Simulation

Simulation testing was performed with the aid of DNAnet. A series of tests on differing nets was performed. The nets to be simulated were created in DNAnet, simulated, and then imported into DaNAMiCS where the results were compared. Five Petri nets were simulated by both programs and the results checked. The five Petri nets used are shown in Figure 13.11.

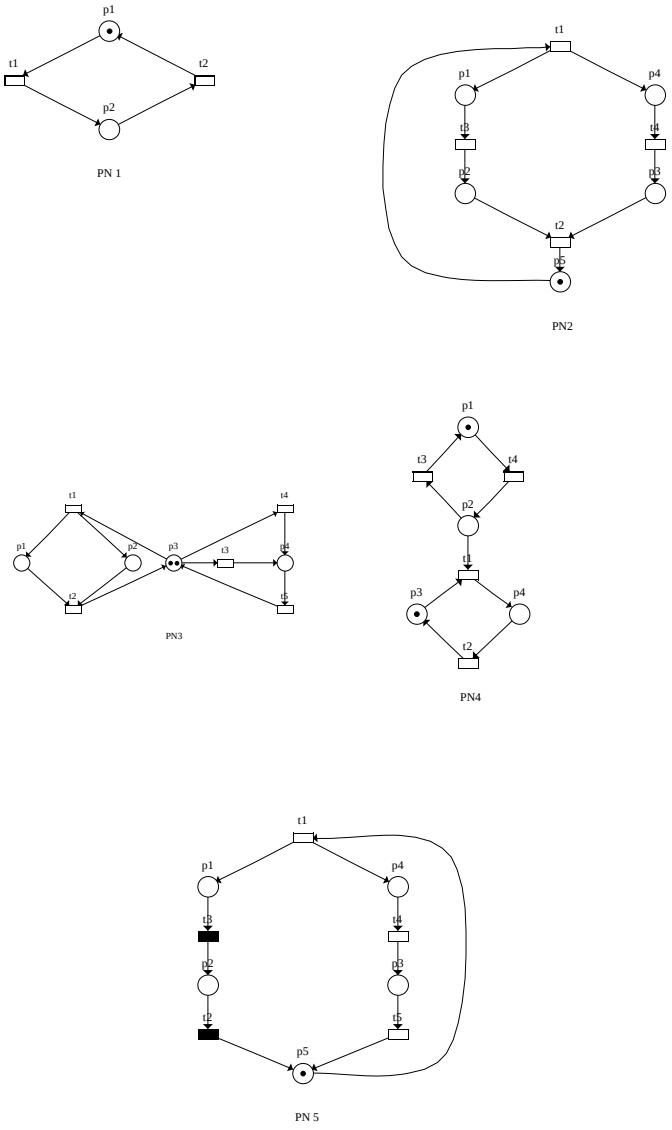


Figure 13.11: The five Petri nets used in the simulation tests.

We see, from Table 13.1, that the results, while not exactly the same, are very similar and thus we can safely

	DNAnet		DaNAMiCS	
	Throughput of t_1	Mean Tokens of p_1	Throughput of t_1	Mean Tokens of p_1
PN1	0.5	0.5	0.5	0.5
PN2	0.28	0.29	0.29	0.28
PN3	0.36	0.45	0.35	0.45
PN4	0.25	0.5	0.25	0.5
PN5	0.97	0	0.99	0

Table 13.1: Showing the differences in simulation results between DNAnet and DaNAMiCS

assume that the simulation is working correctly. It is also important to remember that simulation results will vary from run to run, and this should also be taken into account.

13.2.4 Testing Steady state analysis

In order to perform steady state analysis, a model file must first be created. DNAmaca analyses this model and produces the performance results. So, testing steady state analysis meant testing whether these model files were created correctly, rather than testing the network connection between DaNAMiCS and server - which falls under interface testing.

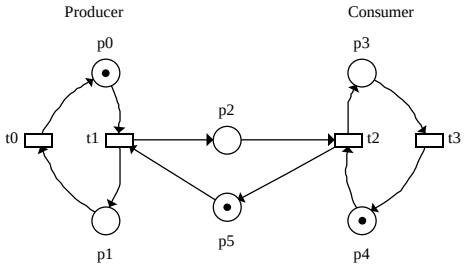


Figure 13.12: One of the Petri nets used in testing steady state analysis.

As a sample test, Figure 13.12 shows a Petri net which models the producer/consumer problem. In this net, it is obvious that the consumer will consume as much as the producer produces. DNAmaca produced the following *transition throughput*results when run on the model file that DaNAMiCS created:

```
Count Measure : Throughput for transition t3

  mean                  3.5714285714e-01


Count Measure : Throughput for transition t2

  mean                  3.5714285714e-01
```

```
Count Measure : Throughput for transition t1

  mean                 3.5714285714e-01


Count Measure : Throughput for transition t0

  mean                 3.5714285714e-01
```

The transition throughput shows that, on average, **t1** and **t2** fire the same number of times. This is the expected result, and was confirmed by DNAnet's analysis, which is:

```
Count Measure 'Throughput for transition main.t1'

  mean                 3.5714285714e-01

Count Measure 'Throughput for transition main.t2'

  mean                 3.5714285714e-01

Count Measure 'Throughput for transition main.t3'

  mean                 3.5714285714e-01

Count Measure 'Throughput for transition main.t4'

  mean                 3.5714285714e-01
```

Steady state analysis was tested with many examples in this way. This ensured that the performance results obtained by DaNAMiCS were correct.

13.3 Structural testing

Structural testing is also known as *white-box* testing. This kind of testing relies on knowledge of the algorithms and program code to determine the test cases. Since the algorithms in DaNAMiCS were often developed incrementally, structural testing was used to test the newly added sections after they were coded.

For example, the invariant analysis algorithm contains sections that are not always executed. After the invariant analysis algorithm was coded, structural testing ensured that sample Petri nets were found that that caused those sections of the algorithm to run.

13.4 Component testing

There were many occasions where certain components of DaNAMiCS were independent of each other. These components were developed and tested in isolation, before being integrated into the rest of the

DaNAMiCS program.

For example, in order to conserve network bandwidth, a class which used the Java **GZipStream** was created. This class was tested to ensure that it could compress and decompress correctly by using UNIX's **gzip** and **gunzip** on sample files that were created. Only once its functionality was proven, was it integrated into DaNAMiCS for use on network packets.

13.5 Interface testing

Interface testing occurred when various components were integrated to form larger sub-systems of DaNAMiCS. Since these components were already tested, errors in the new sub-system were unlikely to be caused from faulty components, but rather through misuse of the component.

An example of this was with the development of the client and server classes for the steady state analysis. These classes were tested independently of each other for functional correctness. Once they were working as stand-alone classes, they then had to communicate via network packets. Interface testing highlighted errors introduced by the byte ordering differences between C++ and JAVA for certain functions that inserted data into the packets.

13.6 User testing

Involving potential users in program testing was a very important aspect of the DaNAMiCS project. Their input influenced the project specification, the look and feel of DaNAMiCS and helped test the program's capabilities and correctness.

Since DaNAMiCS was supposed to provide a better Petri net tool than DNAnet, the users of DNAnet were used as a test base for DaNAMiCS. These users had already made suggestions for improvement and, in fact, this is why the DaNAMiCS project was originally proposed. Their suggestions formed the basis of the functional requirements for DaNAMiCS. Since the users of DNAnet proposed these additional capabilities, they could test whether DaNAMiCS met their objectives or not.

A mailing list was set up that contained the known users of DNAnet - about one hundred and fifty in total. They were contacted early in the project to find out exactly what it was that they wanted to see in a new modeling tool such as DaNAMiCS. The response showed that users wanted:

- support for analysis of GSPNs. This was already provided by DNAnet.
- a convenient way to view results. Currently, many of the modeling tools write results to file. These results then have to be viewed externally, which is very inconvenient.
- subnet support. DNAnet provided subnets, and users found that feature to be very useful.
- support for inhibitor arcs.
- support for coloured tokens.

As can be seen in Section 13.6.1, the users certainly feel that DaNAMiCS has met these objectives.

The users were kept informed of the development status of DaNAMiCS via the DaNAMiCS web site. Once development had reached a sufficient stage, a beta version of DaNAMiCS was made available for download. The mailing list was used to remind DNAnet users that the DaNAMiCS tool was now available for download. This proved to be an invaluable means of feedback, and was a very useful way of testing DaNAMiCS. For example, after some of the responses, changes were made to the toolbar and the results viewer. Also, certain bugs in the analysis were pointed out which revealed errors in the implementation of the analysis algorithms. These were promptly corrected and the updated version was made available for download.

13.6.1 User response

Most of the problems that users had after downloading DaNAMiCS was actually getting DaNAMiCS to run. This was caused by the users having an old version of JAVA, or, by users who tried to run the JAVA class files as executables. The web-page was then updated to more explicitly define the system requirements and instructions needed to execute DaNAMiCS.

The only negative responses had to do with the implementation of coloured tokens. Some users did not like the fact that coloured tokens were displayed as the colours ”red”, ”green” and ”blue”. They would have preferred coloured tokens with distinguishing labels such as ”CC”, ”CR” and so on. Despite that, they felt that coloured Petri nets were a very useful feature in DaNAMiCS.

The user response to DaNAMiCS has been very good. Users are impressed with the extended classes of models that can be created and analysed, and with the way that the results are displayed. In this respect, DaNAMiCS has certainly achieved its objectives.

Chapter 14

Conclusion

14.1 DaNAMiCS

The DaNAMiCS project succeeds in its aim to provide a flexible system in which to design Petri nets that model real-world problems. With the use of inhibitor arcs, we allow any possible system to be modeled and increase our modeling power to that of a Turing machine.

We present the users with a wealth of newly implemented features, such as the ability to model with coloured tokens, and the ability to place capacities on the number of tokens in a place. We allow the user hierarchical modeling techniques with subnets to decrease the complexity of the Petri nets and increase abstraction.

In addition to this, we increase the analysing power of DaNAMiCS to cope with the new additions. We present novel work on the formulation of an algorithm to generate coverability graphs for Petri nets with the above extensions. Although it is proven that coverability analysis is undecidable in the general case, the new algorithm can correctly analyse a much wider class of Petri nets. Finally, we improve the analysis process by providing the user with an intuitive means to quickly analyse his or her results.

Future users are involved in the development of DaNAMiCS through a mailing list, and via the DaNAMiCS website. The mailing list is an important means of obtaining user feedback, and the web site allows users to download the current version of DaNAMiCS. Allowing users to use the system as it is being developed provides an excellent way to test DaNAMiCS. This results in DaNAMiCS being well-suited to users' needs.

DaNAMiCS has met, and exceeded all of its goals, and is a promising new modeling tool.

14.2 Future Work

It as already been mentioned that work on Coloured Petri nets could be useful. This would enhance the flexibility of the modeling tool as a whole. Further work could include the addition of high-level Petri nets which are an extension to Petri nets and allow very powerful, complex modeling.

A possible future system could be a DaNAMiCS server with optimised C++ code to analyse Petri nets from different clients. The analysis functions could be greatly accelerated if implemented in C++ and run on an independent system.

There are many different classes of Petri net available for modeling purposes. These can be very specialised and therefore are not necessarily useful to all users. However, some users may find them invaluable. For

this reason future work could involve the implementation of more problem specific net classes. One of the benefits of these classes is that they allow the use specialised analysis techniques to obtain detailed information on the various properties of the Petri nets.

14.3 Concluding Remarks

DaNAMiCS has been a rewarding project. Although the learning curve for DaNAMiCS was very steep, it is felt that the improvement over DaNAMiCS's predecessor is significant. DaNAMiCS should serve as a useful teaching tool as all the basic features of Petri nets are available, in addition to various advanced features in the models and the analysis techniques.

Acknowledgments

DaNAMiCS would not have become a reality without the help of the following people:

- Our supervisor, Prof P.S. Kritzinger who gave the project direction and encouraged us to experiment with many new ideas.
- Dr A.C.M. Hutchison, who inspired us, and gave up much of his time to assist us with this and other documents.
- William Knottenbelt, whose long-distance advice, bug-testing, ideas, and suggestions were invaluable.
- Simon Buffler, who gave up valuable time to proof-read this document.
- Heinz Kabutz, whose prompt reponse, ideas , and bug-testing were very useful.
- All the users, around the world, who downloaded the original versions of DaNAMiCS and gave comment.
- Our DNA Laboratory colleagues, Elton Saul and Matthew Berry, whose wit and brilliant ideas made the early mornings in the lab worthwhile.
- The Honours Class of 1998, for their friendship and entertaining lunch-time debates.
- Our friends and family:
  - Ian's parents (for putting up with him for another year), James, and Natalie, for their constant caring and support,
  - Marla, for forcing deserts and cappuccinos upon Byron; Miriam, for never caring too much; and Terry, for his uncanny ability to point out how hopeless the situation really is.
  - Michael's parents and close friends.

Appendix A

DaNAMiCS File Format Grammar

A.1 Petri net Files

The following grammar is the file format for Petri net files saved by DaNAMiCS. The convention used for this grammar is a mixture of *Context Free Grammar* and *Regular Expression* syntax. For example, **PlaceEntry*** means that the non-terminal PlaceEntry can appear zero or more times in a row. Non-terminal symbols are enclosed in quotes e.g. ”Places (” is a string of non-terminals.

```
PetriNet            :  "#PetriNet"
                       PlacesList
                       TransitionsList
                       SubnetsList
                       ArcsList
                     | "#ColouredNet"
                       NumColours
                       CPlacesList
                       CTransitionsList
                       CSubnetsList
                       CArcsList

PlacesList          :  "Places [" PlaceEntry* "]"

TransitionsList     :  "Transitions [" TransitionEntry* "]"

SubnetsList         :  "Subnets [" SubnetEntry* "]"

ArcsList            :  "Arcs [" ArcEntry* "]"

CPlacesList         :  "CPlaces [" CPlaceEntry* "]"

CTransitionsList    :  "CTransitions [" CTransitionEntry* "]"

CSubnetsList        :  "CSubnets [" SubnetEntry* "]"
```

```
CArcsList           :  "CArcs [" CArcEntry* "]"

PlaceEntry          :  "(" Xpos "," Ypos ")"
                       Initial
                       Current
                       Limit
                       IPstatus
                       Name

TransitionEntry     :  "(" Xpos "," Ypos ")"
                       Orientation
                       Type
                       WeightRate
                       IPstatus            Name

SubnetEntry         :  "(" Xpos "," Ypos ")" FileName Name

ArcEntry            :  "Edge (" Weight Type HandleList?  FromEntry ToEntry ")"

FromEntry           :  "SubnetFrom:" Name
                    |  "SubnetFrom:" SubnetName FromEntry

ToEntry             :  "SubnetTo:" Name
                    |  "SubnetTo:" Name ToEntry

Xpos                :  integer

Ypos                :  integer

HandleList          :  "(" Xpos "," Ypos ")"
                    |  "(" Xpos "," Ypos ")" HandleList

CPlaceEntry         :  "(" Xpos "," Ypos ")" Initial+ Limit+ IPstatus Name

CTransitionEntry    :  "(" Xpos "," Ypos ")"
                       Orientation
                       Type
                       IPstatus
                       Name
                       NumModes
                       Modes

Modes               :  "Modes " ModeEntry+ ""

ModeEntry           :  "mode entry (" NumModes IntList "rate" WeightRate ")"
```

```
CArcEntry          :  "Edge ("
                      NumModes
                      IntList+
                      Type
                      HandleList?
                      FromEntry
                      ToEntry ")"

IntList            :  int
                   |  int IntList

NumModes           :  "numModes" int

Initial            :  integer

Current            :  integer

Limit              :  integer

IPstatus           :  isPip
                   |  notPip

Name               :  string

FileName           :  string ".bim"

SubnetName         :  string

Orientation        :  "vertical"
                   |  "horizontal"

Type               :  0
                   |  1

WeightRate         :  real

Weight             :  int

NumColours         :  int
```

A.2 Results Files

The grammar for the DaNAMiCS results files is presented here. The grammar can easily be extended to cope with additions. The basic methodology used in the creation of the results grammar was to allow it to be

extensible and flexible, such that the various modules can be removed or added at will. The results are taken from the four different results viewing panes and written to four individual files. The grammar is written as follows:

```
Results              :  Specific_Results
                     |  Results

Specific_Results     :  Invariant_Results
                     |  Coverability_Results
                     |  Simulation_Results
                     |  Performance_Results

Invariant_Results    :  T_Statistics
                     |  P_Statistics
                     |  T_Invariants
                     |  P_Invariants
                     |  Equations
                     |  Incidence_Matrix

T_Statistics         :  ``\begin{TStatistics}'' INFO_TEXT ``\end{TStatistics}''

P_Statistics         :  ``\begin{PStatistics}'' INFO_TEXT ``\end{PStatistics}''

T_Invariants         :  ``\begin{TInvariants}'' INFO_TEXT ``\end{TInvariants}''

P_Invariants         :  ``\begin{PInvariants}'' INFO_TEXT ``\end{PInvariants}''

Equations            :  ``\begin{Equations}'' INFO_TEXT ``\ end{Equations}''

Incidence_Matrix     :  ``\begin{Matrix}'' INFO_TEXT ``\ end{Matrix}''

Coverability_Results :  Liveness
                     |  Boundedness
                     |  General
                     |  Graph

Liveness             :  ``\begin{Liveness}'' INFO_TEXT ``\end{Liveness}''

Boundedness          :  ``\begin{Boundedness}'' INFO_TEXT ``\end{Boundedness}''

General              :  ``\begin{General}'' INFO_TEXT ``\end{General}''

Graph                :  ``\begin{Graph}'' INFO_TEXT ``\end{Graph}''

Simulation_Results   :  Throughput
                     |  MeanTokens
```

```
Throughput          :  ``\begin{Throughput}'' INFO_TEXT ``\end{Throughput}''

MeanTokens          :  ``\begin{Throughput}'' INFO_TEXT ``\end{Throughput}''

Performance_Results :  Throughput
                    |  MeanTokens
                    |  Distribution

Distribution        :  ``\begin{Distribution}'' INFO_TEXT ``\end{Distribution}''
```

Where “INFO_TEXT” is any string of text.

Appendix B

DNAmaca Syntax

B.1 The Interface Definition Language for DNAmaca

The following symbols are used in the definition:

`{ X }*`	denotes one or more occurrences of X
`|`	separates alternatives

As in TEX, comments begin with `%`; the remainder of the input line is ignored.

The grammar for the language is:

```
model_description = \model {
  {
    state_vector | initial_state | transition_declaration |
    constant | help_value | invariant | state_output_function |
    primary_hash_function | secondary_hash_function | additional_headers
  }*
}

state_vector = \statevector{
  { <type> <identifier> {, <identifier> }*; }*
}

type = basic C/C++ variable type;
identifier = valid C/C++ identifier;

initial_state = \initialstate {
  { <assignment> }*
}

assignment = C/C++ assignment to elements of the state vector
```

```
transition_declaration = \transition{<identifier>}{
  \condition{<boolean expression>}
  \action{ { <assignment> }* }
  \rate{<real expression>} | \weight{<real expression>}
  \priority{<non-negative integer>}
}

boolean expression = C/C++ boolean expression
real expression = C/C++ real expression
assignment = C/C++ assignment

constant = \constant{<identifer>}{value}

help_value = \helpvalue{<type>}{<identifier>}{<expression>}

invariant = \invariant{<expression>}

state_output_function = \output {
  { <statements> }*
}

statements = C++ statements to output elements of the state vector

primary_hash function = \primaryhash {
  <C++ function body returning an integer from 0 to 16383>
}

primary_hash function = \secondaryhash {
  <C++ function body returning a 32-bit integer>
}

additional_headers = \header {
  <C++ include statements and/or class definitions>
}

solution_control = \solution {
  { \method{gauss | grassman | gauss_seidel | sor | bicg | cgnr |
       bicgstab | bicgstab2 | cgs | tfqmr | ai | air | automatic} |
    \accuracy{<real>} |
    \maxiterations{<long int>} |
    \relaxparameter{<real> | dynamic} |
    \startvector{ <filename> } |
    \reportstyle{full | short | none} |
    \reportinterval{<long int>}
  }*
}
```

```
\performance_measures = \performance {
  { state_measure | count_measure }*
}

state_measure = \statemeasure{identifier}{
  \estimator{ {mean | variance | stddev | distribution}* }
  \expression{<real_expression>}
}

count_measure = \count_measure{identifer}{
  \estimator{mean}
  \precondition{<boolean expression>}
  \postcondition{<boolean expression>}
  \transition{ all | {<identifier>}* }
}

output_options = \outputoptions {
  { \statelist{<filename>} |
    \steadystatevector{<filename>} |
    \transitionmatrix{<filename>} |
    \performanceresults{<filename>}
  }*
```

B.2 Sample model file produced by DaNAMiCS for DNAmaca

```
\model {
    \statevector{
        \type{long} {p0, p1}
    }
    \initial{
        p0 = 0;
        p1 = 0;
    }
    \transition{t0} {
        \condition{1
        }
        \action{
            next->p0 = p0 + 1;
        }
        \rate{1.0}
    }
    \transition{t1} {
        \condition{p0 > 0
        }
        \action{
```

```
            next->p0 = p0 - 1;
            next->p1 = p1 + 1;
        }
        \weight{1.0}
    }
    \transition{t2} {
        \condition{p1 > 2
        }
        \action{
            next->p1 = p1 - 3;
        }
        \weight{1.0}
    }
}
\performance{
    \statemeasure{Mean tokens on place p0} {
        \estimator{mean variance distribution}
        \expression {p0}
    }
    \statemeasure{Mean tokens on place p1} {
        \estimator{mean variance distribution}
        \expression {p1}
    }
    \statemeasure{Enabled probability for transition t0} {
        \estimator{mean}
            \expression{ 1 }
    }
    \statemeasure{Enabled probability for transition t1} {
        \estimator{mean}
            \expression{ (p0 > 0) ? 1 : 0 }
    }
    \statemeasure{Enabled probability for transition t2} {
        \estimator{mean}
            \expression{ (p1 > 2) ? 1 : 0 }
    }
    \countmeasure{Throughput for transition t0} {
        \estimator{mean}
        \precondition{1}
        \postcondition{1}
        \transition{t0}
    }
}
```

Appendix C

Project Planning

C.1 Overview

The DaNAMiCS project was implemented by three Honours students and required careful planning to coordinate the various activities. A plan for the project was drawn up very early in the project, which made estimates of the times to be spent on each aspect of the project. We made use of the *workpackage detail* planning structure and variance reports were compiled periodically for our supervisor to check the progression of the project. In total, approximately 1400 person hours were spent on DaNAMiCS in design, implementation, and documentation.

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** -	**Task** -
Title DaNAMiCS			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 1222	**Cost** -	**Start Date** -	**Fin. Date** 24/10/98

Brief Description
To design a Petri Net editing tool for use in the modelling of concurrent systems.

Detailed Description

Project

- 1A: 1: Interface
- 1B: 2: Object-Oriented Design
- 1C: 3: Petri Net Implementation
- 1D: 4: Logic Implementation
- 1E: 5: Documentation

Outputs:

- A Petri Net Editor
- A Petri Net Simulation Tool
- A Petri Analyser for Markovian Analysis
- a Petri Net Data Results Viewer

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 1	**Task** -
Title DaNAMiCS - Interface			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 160	**Cost** -	**Start Date** -	**Fin. Date** 24/10/98

Brief Description

Design and implement and interface for the tool. We will use iterative design with help from actual end-users to design the optimum interface.

Detailed Description

Element 1

- 1A: 1: Editor
- 1B: 2: Results Viewer

Outputs:

- A Petri Net interface suitable for construction, simulation and analysis
- A good visualisation tool for the analysis of Petri Nets

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 1	**Task** A
Title DaNAMiCS - Interface - Editor			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 90	**Cost** -	**Start Date** 6/05/98	**Fin. Date** 31/5/98

Brief Description
Design and write the widgets and dialogs for the interface

Detailed Description

Task 1A

- 1AA: 1: Canvas with splinse and Petri Net objects
- 1AB: 2: The Floating Button Bar and dialogs
- 1AC: 3: Menu System with sub-menus
- 1AD: 4: Testing

Outputs:

- A Petri Net interface suitable for construction, simulation and analysis

Staff

- (bc)
- (id)
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 1	**Task** B
Title DaNAMiCS - Interface - Results Viewer			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 70	**Cost** -	**Start Date** 10/05/98	**Fin. Date** 31/08/98

Brief Description
Write a useful tool, determined by the end users for visualising Petri Net data

Detailed Description

Task 1B

- 1BA: 1: Panel Design
- 1BB: 2: Scripting Language for multiple results processing
- 1BC: 3: Visualisation techniques explored
- 1BD: 4: Verification

Outputs:

- A results visualisation component

Staff

- bc
- id
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** -
Title DaNAMiCS - OO Design			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 28	**Cost** -	**Start Date** 05/05/98	**Fin. Date** 25/05/98

Brief Description
Perform full object-oriented design on the DaNAMiCS project showing interactions and attributes.

Detailed Description

Element 2

- 2A: 1: Design the classes for the underlying Petri Net structure
- 2B: 2: Design the classes for performing the mathematical operations on the Petri Net
- 2C: 3: Design the classes for assessing correctness
- 2D: 4: Design the classes for calculating performance
- 2E: 5: Verify this design

Inputs:

- The existing DNAnet structure

Outputs:

- The system object design

Staff

- bc
- id
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** A
Title DaNAMiCS - OO Design - Structure			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 8	**Cost** -	**Start Date** 05/05/98	**Fin. Date** 25/05/98

Brief Description
Design the structural objects

Detailed Description

Task 2A

- 2AA: 1: Design the classes for the places
- 2AB: 2: Design the classes for the transitions
- 2AC: 3: Design the classes for arcs

Inputs:

- The existing DNAnet structure

Outputs:

- The structural object design

Staff

- bc
- (id)
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** B
Title DaNAMiCS - OO Design - Mathematics			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 2	**Cost** -	**Start Date** 20/05/98	**Fin. Date** 21/21/98

Brief Description
Design the math objects

Detailed Description

Task 2A

- 2AA: 1: Design the matrices and other necessary mathematical structures

Inputs:

- The existing DNAnet structure

Outputs:

- The math object design

Staff

- bc
- id
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** C
Title DaNAMiCS - OO Design - Correctness			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 5	**Cost** -	**Start Date** 05/05/98	**Fin. Date** 09/05/98

Brief Description
Design the correctness classes

Detailed Description

Task 2C

- 2CA: 1: Design the classes for a reachability tree
- 2CB: 2: Design the classes for the invariant analysis

Inputs:

- The existing DNAnet structure

Outputs:

- The correctness object design

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** D
Title DaNAMiCS - OO Design - Performance			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 5	**Cost** -	**Start Date** 10/05/98	**Fin. Date** 20/05/98

Brief Description
Design the performance classes

Detailed Description

Task 2D

- 2DA: 1: Design the classes for performing simulations
- 2DB: 2: Design the classes for the Markov Chain analysis

Inputs:

- The existing DNAnet structure

Outputs:

- The performance objects design

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 2	**Task** E
Title DaNAMiCS - OO Design - Testing			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 8	**Cost** -	**Start Date** 23/05/98	**Fin. Date** 25/05/98

Brief Description
Design the performance classes

Detailed Description

Task 2E

- 2EA: 1: Verify the entire object structure

Inputs:

- The recently produced object structure

Outputs:

- A valid, viable design for implementation

Staff

- bc
- (id)
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 3	**Task** -
Title DaNAMiCS - Implementation			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 352	**Cost** -	**Start Date** 20/05/98	**Fin. Date** 31/08/98

Brief Description
Following the object design we implement all aspects of the project

Detailed Description

Element 3

- 3A: 1: Write the Petri Net structure
- 3B: 2: Design and write the Inhibitor arcs
- 3C: 3: Design and write the Coloured tokens
- 3D: 4: Test the design

Inputs:

- The verified object structure

Outputs:

- A fully functional implementation of the system with regard to a the above components

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 3	**Task** A
Title DaNAMiCS - Implementation - Structure			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 80	**Cost** -	**Start Date** 20/05/98	**Fin. Date** 30/06/98

Brief Description
Implement the basic but essential components of the Petri Net

Detailed Description

Task 3A

- 3AA: 1: Places: Tokens and orientation
- 3AB: 2: Transitions: Enabling, firing, and orientation
- 3AC: 3: Arcs: orientation

Inputs:

- The object structure for the petri net objects

Outputs:

- The underlying petri structure implemented

Staff

- (bc)
- (id)
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 3	**Task** B
Title DaNAMiCS - Implementation - Inhibitor Arcs			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 80	**Cost** -	**Start Date** 31/06/98	**Fin. Date** 31/07/98

Brief Description
Design and Write the inhibitor arcs

Detailed Description

Task 3B

- 3BA: 1: Background reading
- 3BB: 2: Design the basic structure
- 3BC: 3: Integrate this into the existing structures

Inputs:

- The basic petri net object structure

Outputs:

- An extended petri net implementation to handle inhibitor arcs

Staff

- (bc)
- id
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 3	**Task** C
Title DaNAMiCS - Implementation - Coloured Tokens			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 96	**Cost** -	**Start Date** 31/06/98	**Fin. Date** 31/07/98

Brief Description
Design and Write the inhibitor arcs

Detailed Description

Task 3C

- 3CA: 1: Background reading
- 3CB: 2: Design the basic structure
- 3CC: 3: Integrate this into the existing structures

Inputs:

- The basic petri net object structure

Outputs:

- An extended petri net implementation to handle coloured tokens

Staff

- bc
- (id)
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 3	**Task** D
Title DaNAMiCS - Implementation - Testing			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 96	**Cost** -	**Start Date** 30/06/98	**Fin. Date** 31/07/98

Brief Description
Test all the implemented aspects

Detailed Description

Task 3D

- 3DA: 1: Test the basic structure
- 3DB: 2: Test the inhibitor arcs
- 3DC: 3: Test the coloured tokens

Inputs:

- The complete structural implementation

Outputs:

- A fully functional implementation of the system with regard to a the above components

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 4	**Task** -
Title DaNAMiCS - Logic			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 440	**Cost** -	**Start Date** 15/07/98	**Fin. Date** 31/09/98

Brief Description
This is the design of all the logical elements of the system

Detailed Description

Element 4

- 4A: 1: Correctness
- 4B: 2: Simulation
- 4C: 3: Markov Chain analysis
- 4D: 4: Test the above elements

Inputs:

- The basic object structure

Outputs:

- A system that can support the necessary functional requirements

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 4	**Task** A
Title DaNAMiCS - Logic - Correctness			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 180	**Cost** -	**Start Date** 15/07/98	**Fin. Date** 31/08/98

Brief Description
This is the design of the correctness elements of the system

Detailed Description

Task 4A

- 4AA: 1: Write the coverability graph
- 4AB: 2: Write the matrices and functionality for the P- and T-invariants
- 4AC: 3: Liveness functions
- 4AD: 4: Boundedness functions

Inputs:

- The basic object structure

Outputs:

- A system that can support correctness

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 4	**Task** B
Title DaNAMiCS - Logic - Simulation			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 120	**Cost** -	**Start Date** 15/07/98	**Fin. Date** 31/08/98

Brief Description
This is the design of the correctness elements of the system

Detailed Description

Task 4B

- 4BA: 1: Functions to actuall perform the simulation (state space exploration)
- 4BB: 2: Calculation of transition throughput
- 4BC: 3: Calculation of the mean number of tokens per place
- 4BD: 4: Trace the simulation

Inputs:

- The basic object structure

Outputs:

- A system that can perform simulations on Petri nets

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 4	**Task** C
Title DaNAMiCS - Logic - Markov Chain Analysis			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 100	**Cost** -	**Start Date** 20/08/98	**Fin. Date** 15/09/98

Brief Description
This is the design of the performance assessment of the Petri nets by means of Markov Chain analysis

Detailed Description

Task 4C

- 4CA: 1: Background reading into Markov Chain analysis
- 4CB: 2: Write the client/server DaNAMiCS/DNAMaca interaction
- 4CC: 3: Interpret the data received from DNAmaca
- 4CD: 4: Extensive network strain tests

Inputs:

- The basic object structure
- William's (or Mark's) Markov Chain analyser (perhaps both)

Outputs:

- A system that can provide exact performance measurements

Staff

- bc
- (id)
- (mn)

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 4	**Task** D
Title DaNAMiCS - Logic - Testing			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 40	**Cost** -	**Start Date** 15/08/98	**Fin. Date** 31/09/98

Brief Description
Test all the logical elements of the system

Detailed Description

task 4D

- 4DA: 1: Test Correctness
- 4DB: 2: Test Simulation
- 4DC: 3: Test the Markov Chain analysis with special reference to the network stability

Inputs:

- The logical elements coded for syntatic functionality

Outputs:

- A fully functional, stable system that can support the necessary functional requirements

Staff

- bc
- id
- mn

WORKPACKAGE WORK DETAIL		**Project** 1	**Element** 5	**Task** -
Title DaNAMiCS - Documentation			**Issue No.** 3	**Date** 29/04/98
Manager Prof. Kritzinger	**Man Hours** 272	**Cost** -	**Start Date** 01/10/98	**Fin. Date** 24/10/98

Brief Description
Write the documentation for the system and the project

Detailed Description

Element 5

- 5A: 1: 10 page Short Paper
- 5B: 2: Project Poster
- 5C: 3: Web-Presence
- 5D: 4: User Manual
- 5E: 5: 100 page Project report

Inputs:

- Necessary background reading
- The DaNAMiCS application

Outputs:

- Complete, understandable documentation and theses

Staff

- bc
- id
- mn

Bibliography

[1] T. Agerwala and M. Flynn. Comments on Capabilities, Limitations and 'Correctness' of Petri nets. *Proceedings of the First Annual Symposium on Computer Architecture*, pages 81–86, July 1973.

[2] H. Alaiwan and G. Memmi. A proposed algorithm for finding positive solutions of homogenous linear equations. *Rev. Tech. Thompson-CSF*, 14:125–135, 1982.

[3] K. Barkaoui and M. Minoux. A Polynomial-time Graph Algorithm to Decide Liveness of some Basic Classes of Bounded Petri Nets. In *Application and Theory of Petri Nets*, pages 62–75, 1992.

[4] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets, An Introduction to the Theory*. 1995.

[5] E. Best. Structure Theory of Petri Nets: The free choice hiatus. *Advances in Petri Nets*, 1:168–205, 1986.

[6] S. Christensen and L. Petrucci. Modular State Space Analysis of Coloured Petri Nets. In *Application and Theory of Petri Nets*, pages 201–217, 1995.

[7] M. D'Anna and S. Trigila. Concurrent System Analysis: An optimized algorithm for finding net invariants. *Computer Communications*, 11:215–220, 1988.

[8] E. Dijkstra. *Cooperating Sequential Processes*. 1968.

[9] P. Kemper F. Bause and P. Kritzinger. Absract Petri Net Notation. *Petri Net Newsletter*, 49, 1996.

[10] K. Jensen. *Coloured Petri Nets, Analysis Methods*. Springer - Verlag, 1992.

[11] K. Jensen. *Coloured Petri Nets, Basic Concepts*. Springer - Verlag, 1992.

[12] William J. Knottenbelt. Generalized Markovian Analysis of Timed Transition Systems, 1996.

[13] S. Kosaraju. Limitations of Dijkstra's Semaphore Primitives and Petri nets. *Operating Systems Review*, 7(4):122–126, October 1973.

[14] C. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Application and Theory of Petri Nets*, pages 278–297, 1995.

[15] J. Martinez and M. Silva. A Simple and Fast Algorithm to Obtain all Invariants of a Generalised Petri Net. In *Application and Theory of Petri Nets, Selected Papers from the First and Second European Workshop on Application and Theory of Petri Nets*, 1981.

[16] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of IEEE*, April 1989.

[17] S. Patil. Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes. *Computation Structures Group Memo 57*, February 1971.

[18] J.L. Peterson. Petri Nets. *ACM Computing Surveys*, (9):242–252, September 1977.

[19] J.L. Peterson. *Petri net Theory and the Modeling of Systems*. 1981.

[20] W. Reisig. Petri Nets. An Introduction. *EATCS Monographs on Theoretical Computer Science*, 4, 1985.

[21] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. 1994.

Index