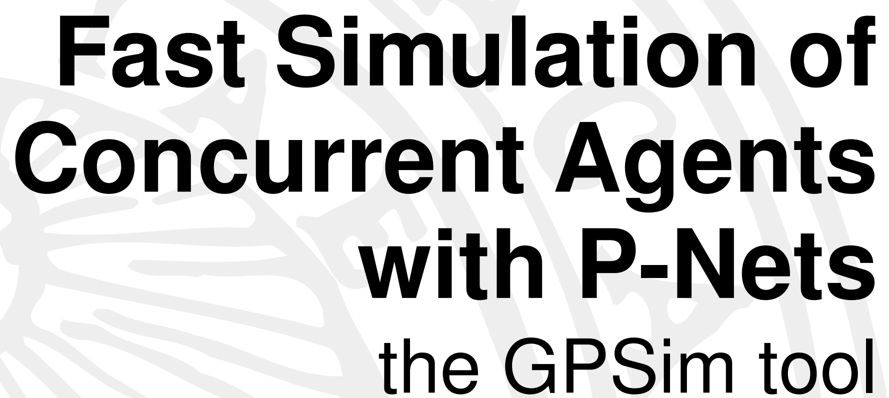

Fast Simulation of Concurrent Agents with P-Nets

the GPSim tool

Timo Qvist

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2005-027
September 2005
ISSN 1404-3203

Abstract

The aim of this thesis was to investigate and enlighten the applicability of P-Nets for simulating large and possibly infinite–control systems. P-Nets are generalized coloured Petri nets [Pet62] which retain detailed information in the tokens regarding firing history and the scope of concurrently executing threads. P-Nets are specified with the help of a process calculi called CCS_k which is a derivation of Milner's CCS [Mil89], and converted to a P-Net representation prior to simulation. A simulation tool named GPSim was implemented using both generic optimization techniques and techniques specific to P-Net structure and semantics. The simulation performance of GPSim was evaluated and compared to two well–known formal verification tools with respectable pedigrees; Bell Labs' Spin and The Concurrency Workbench of the New Century from Stony Brook University.

Thanks

I wish to thank my parents for a truly never ending support and also a very special person who knows who she is.

Contents

1 Introduction **5**
1.1 Goal 6

2 P-Nets **6**
2.1 CCS_k Language 7
2.1.1 CCS_k Semantics and Terminology 7
2.2 P-Net Creation 8
2.2.1 Places 8
2.2.2 Labels 8
2.2.3 Transitions 9
2.2.4 Transition Classification 9
2.2.5 Synchronizations 10
2.3 Firing Policy 11
2.3.1 Enabling Candidates for Transitions 12
2.3.2 Firing Policy Definition 12
2.4 P-Net Imperfections 13
2.4.1 Early Termination 13
2.4.2 Extraneous Synchronization Transitions 14
2.4.3 Redundant Precondition 15
2.5 Guarded P–Net: GP-Net 15
2.5.1 Guarded Summations 15
2.5.2 Redefinition of $translate()$ and $sync()$ 15
2.5.3 Precondition Enhancement 16

3 Petri Net Simulation **16**
3.1 Simulation Example 17
3.2 Discrete Event Simulation for GP–Nets 18
3.3 A Naive DES algorithm 18

4 Optimization **19**
4.1 Four Tasks of Simulation 20
4.1.1 Simulation State and Static Properties 20
4.2 Optimizing the Naive Algorithm 21
4.2.1 The Locality Principle 21
4.2.2 Token History Information 22
4.2.3 Token Creation 22
4.2.4 Caching $maxpref()$ 24
4.2.5 Label Categorization 25
4.2.6 Memory Handling 26

5 Evaluation of Optimization Techniques **27**
5.1 A Token Infinite Net 27
5.2 A Token Finite Net 28
5.3 An Experimental Net 30

6 Case Studies **31**
6.1 CWB–NC Overview 32
6.2 Spin Overview 33
6.3 Case Study Details 33
6.4 A CCS CSMA/CD Protocol 33
6.4.1 Scaling the Protocol 34
6.5 Case Study Comparisons 35
6.5.1 CWB–NC Comparison 35
6.5.2 Spin Comparison 36
6.6 Optimization Summary 36
6.7 Future Optimizations 37

7 GPSim Architecture **38**
7.1 Implementation Language and Methodology 38
7.2 Net Representation 38
7.2.1 The `Place` Object 38
7.2.2 The `Token` Object 39
7.2.3 The `Transition` Object 39
7.2.4 The `Net` Object 39

8 GPSim Usage **39**
8.1 GPSim Options 39
8.2 Input Language 40

9 Appendix **44**
9.1 Original Two Station MAC Protocol 44
9.2 Scaled Three Station MAC Protocol 44
9.3 Adequacy Theorem for P–Nets 45

1 Introduction

Formal methods is an approach that strives to assert a system's correctness by formally proving that it meets the specified requirements. But to do so using standard mathematics is daunting since even a modest concurrent system will challenge most mathematicians. Not only that but it was proven to be impossible to construct a general proof for an arbitrary program already back in the 30's by Alan Turing [Tur36]. The solution is to use a mathematical formalism that, contrary to a generic programming language, only contains the expressive power to model the essential intrinsic behaviour and interaction of components in the system. By limiting the formalism in such a way the models created will be of a class that allow formal proofs to be practical. Of course, if the model doesn't accurately describe the system, any proofs will be useless. There exists a lot of misconceptions about formal methods, most of which are dispelled in [Hal90].

One such mathematical formalism, a well–known language for concurrent systems, is Robin Milner's CCS [Mil89]. CCS is commonly referred to as a process calculus because it uses a simple syntax of mathematical expressions where processes are denoted as operands and operators denote concepts such as actions, choice (nondeterminism), and parallelism. Another well-known formalism, named after its inventor Carl Adam Petri, is Petri nets [Pet62]. Petri nets are a generalization of automata and thus takes a graphical approach to modelling concurrent systems, but the two formalisms complement each other in a constructive way. CCS is suitable for a high–level view of a system, whereas Petri nets are suitable for a low–level view because of their similarity to automata.

Mechanical verification of system properties have been performed by automated tools for quite some time, but contemporary concurrent systems may be arbitrarily large and complex which puts a practical limit on their usefulness. In the event that a model becomes too complex for practical mechanical verification, simulation of the model is a viable alternative for raising the confidence level of the integrity of the model.

Simulation is the process of *executing* the model of a system and gathering statistical information about the system's behaviour. The information gained in such a way gives insight into the operational semantics of the system, such as resource usage for instance. Simulation has many advantages over observing a real physical system, we mention here but a few;

- Simulation may proceed on an abstract mathematical model of the system before actual implementation of the entire system (or recently introduced components).
- Repeated simulation can increase the statistical reliability of the system analysis.
- Simulation can be performed in accelerated time, certain conditions may be reproduced within seconds instead of waiting hours or perhaps days or weeks for a physical system to reach a given state.

Simulation is often used to hash out early bugs in a specification, but may also divulge information about bottlenecks, deadlocks and unfairness; all depending on the thoroughness of the analysis. Traditionally, system testing have

been responsible for finding most errors but today the famous statement by Dijkstra [Dij72]

> *"Program testing can be used to show the presence of bugs, but never to show their absence."*

is generally accepted to hold much truth. Formal methods and simulation are thus validated as alternatives, or more likely complements to testing. Simulation is not the answer to all things however. One should be aware of certain issues regarding the process of simulation before investing time and money. For accurate simulation a correct model is essential, otherwise one cannot expect the output to be accurate either. Simulation will not provide easy answers to complex problems, and it will not solve any problems by itself but most likely only detail their existence. Model design and creation may also require skill and experience as would the actual interpretation of the results of a simulation run. Even so, simulation is used in many engineering practices with much success.

1.1 Goal

This thesis is aimed towards investigating the applicability of simulating large and possibly infinite-controlled systems, specified by a CCS based process calculus called CCS_k [Bal03]. This is achieved by generating a finite p-net [Bal03] for any given set of CCS_k expressions, so that their behaviour is equivalent. We subsequently analyse achievable performance of simulating the p-net with GPSim, an optimized simulation tool developed as a part of the thesis. We compare the performance of GPSim to other (both CCS based and non–CCS based) automated verification tools that also feature simulation modes. A variety of such tools exist, but most lack an optimized simulation mode which decrease the value of a comparison when simulation with such tools would involve various types of I/O (animation for instance). However, two tools were chosen. The first tool chosen was Bell Labs' Spin, which is well–known since over fifteen years back and have won the prestigious ACM Software System Award. Spin was chosen because of its pedigree, and because it has its own (non–CCS based) modelling language called PROMELA, it also features a very fast simulation mode. The Concurrency Workbench of the New Century, abbreviated CWB–NC, is a tool for specifying and verifying finite–state concurrent systems and supports a range of CCS-like modelling languages, however its simulation is basic and was never intended for efficient simulation. It was included in the thesis for background reasons, and because it is implemented in SML of New Jersey in contrast to GPSim which was developed in C++.

2 P-Nets

We shall begin with restating the definition of p-nets originating from [Bal03, Bal04] somewhat differently for (readability and clarity) those readers not from the scientific community. The formal definition stated here is intended to be equivalent but contains a different naming scheme, notation and some additional definitions.

A discovery during the course of this thesis rendered the original p-net definition unsuitable for simulation which gave rise to a new derivation called *guarded*

p–nets. We shall follow the original p-net definition with the derivation of guarded p-nets to clarify the differences between the two.

2.1 CCS$_k$ Language

The language from which a p-net is generated, is a derivation of Milner's CCS in [Mil89]. The following changes apply

- It lacks a relabelling operator.
- Summation uses a binary operator instead of one with potentially infinite arity.

2.1.1 CCS$_k$ Semantics and Terminology

To establish a working terminology we define the following sets derivable from a CCS$_k$ specification.

- $\mathcal{N} = \{x, y, z, \dots\}$ with members referred to as *names*.
- $\bar{\mathcal{N}} = \{\bar{x} | x \in \mathcal{N}\}$ where $\mathcal{N} \cap \bar{\mathcal{N}} = \emptyset$. The members are referred to as *co-names*
- $\mathcal{A} = \mathcal{N} \cup \bar{\mathcal{N}} \cup \{\tau\}$, with members referred to as *actions*
- $\mathcal{C} = \{A, B, \dots\}$ with members referred to as *agent constants*.

The $\bar{\ }$-operator is extended to a self-inverse complementation operator on $\mathcal{A}$ via $\bar{\bar{x}} = x$ and $\bar{\tau} = \tau$. The name x and co-name $\bar{x}$ are said to be *complimentary* and have the *channel* x in common. When a distinction isn't necessary between x, $\bar{x}$ or τ we shall denote by α a member of the set $\mathcal{A}$.

It's assumed that every *agent constant* identifier has a unique and possibly recursive *definition* $A \triangleq P$ and the meaning is that A behaves as its body P. For a p-net we assume an environment of agent constant definitions and we call this environment *finite-control* if there is no occurrence of an agent constant beneath the |–operator (composition). If such an occurrence exist the environment is called *infinite–control*. A distinction is made between these environments because infinite–control usually mean an infinite state space.

The syntax of an agent expression is given by the following grammar

$$
\begin{array}{rcl}
P \triangleq & 0 & \textit{(does nothing)} \\
\mid & \alpha.P & (\textit{perform } \alpha \textit{ then behave as } P) \\
\mid & P + Q & (\textit{behave as one of } P \textit{ or } Q) \\
\mid & P \mid Q & (P \textit{ or } Q \textit{ may proceed independently or synchronize}) \\
\mid & P \backslash a & (\textit{behave as } P \textit{ except restricted on } a) \\
\mid & A & (A \textit{ is a constant and behave as its body})
\end{array}
$$

The following clauses is a standard interleaving structural operational semantics (SOS) for agents which make the above description precise. A transition $P \xrightarrow{\alpha} P'$ means that P can perform α and then behaves as P'.

$$\alpha.P \xrightarrow{\alpha} P \quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'} \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \quad \frac{P \xrightarrow{\alpha} P'}{P\backslash c \xrightarrow{\alpha} P'\backslash c}\ c \notin \{\alpha, \overline{\alpha}\} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ A \triangleq P$$

2.2 P-Net Creation

Informally, given a CCS_k specification of the above form, a p-net is created by applying certain rules to the terms and sub terms of agent constants recursively, a process which may be realized by a LALR parser for the above grammar. This process constructs the entities that a p-net is made of, namely places, labels and transitions:

- $\mathit{Places} = \mathcal{C} \cup \{t \mid t \text{ is a sub term on the right hand side of } A \in \mathcal{C}\}$.
- $\mathit{Labels} = \{\epsilon, a, \bar{a}, \backslash a, < \tau, a >, l/r\}$ where $a \in \mathcal{N}$ and $\bar{a} \in \bar{\mathcal{N}}$.
- let $\mathcal{R}$ denote a set of transitions, then transition creation is a two–step process;
    1. $\mathcal{R}_{partial} = \left\{ \bigcup \{translate(p) \mid p \in \mathit{Places}\} \right\}$
    2. $\mathcal{R}_{complete} = \mathcal{R}_{partial} \cup sync(\mathcal{R}_{partial})$

The creation of places is inherently a recursive process during which the partial set of transitions is constructed but to alleviate definition we have depicted this partial set construction as a sequential application of the function $translate()$. The $translate()$ function is applied to places though its behaviour is dependent on the *term* that the place is associated with, see section 2.2.3. An example p-net is shown on page 10.

2.2.1 Places

Places are the p-net equivalent of the CCS_k terms and sub terms in the specification, for instance the term “a.P” would create two places; one associated with “P” and one with “a.P”. Note also that this is a many-to-one mapping which means that all occurrences of “a.P” maps to the same place. Most definitions of Petri nets include a starting place and how to specify it, but for p-nets we assume the place associated with the first encountered agent constant (in the CCS_k specification) to be the starting place.

2.2.2 Labels

Labels in a p-net are directly derived from the set $\mathcal{N}$ in the CCS_k specification with some additional special labels. These labels are attached to transitions by the $translate()$ function, defined in the following section.

2.2.3 Transitions

Creating the transitions of a p-net is a two–step process. When the specification has been parsed we have a p-net with a complete set of places and labels but only a partial set of transitions. A second step is necessary for completeness, but let us first formally define transitions and the $translate()$ function.

A transition is a pair $(Preset, Postset)$ where

- $Preset \subset Places$, where $1 \leq |Preset| \leq 2$
- $Postset \subset Places$, where $1 \leq |Postset| \leq 2$

The function $translate()$ generates these pairs but also the labels that are associated with certain transitions or arcs within transitions. Labellings are denoted by the symbol $\mapsto$.

$$
\begin{aligned}
translate(A \triangleq P) &\Rightarrow \Big\{\Big(\{A\},\{P\}\Big) \mapsto \epsilon\Big\} && (1)\\
translate(P+Q) &\Rightarrow \Big\{\Big(\{P+Q\},\{P\}\Big) \mapsto \epsilon, \Big(\{P+Q\},\{Q\}\Big) \mapsto \epsilon\Big\} && (2)\\
translate(\alpha.P) &\Rightarrow \Big\{\Big(\{\alpha.P\},\{P\}\Big) \mapsto \alpha\Big\} && (3)\\
translate(P\backslash x) &\Rightarrow \Big\{\Big(\{P\backslash x\},\{P\}\Big) \mapsto \backslash x\Big\} && (4)\\
translate(P \mid Q) &\Rightarrow \Big\{\Big(\{P \mid Q\},\{P \mapsto l, Q \mapsto r\}\Big)\Big\} && (5)
\end{aligned}
$$

Note the use of α in rule 3 which may be either $x \in \mathcal{N}$, $\bar{x} \in \bar{\mathcal{N}}$ or τ. Rule 4 specifies a restriction on *channel* x which means it applies to both names x and co–names $\bar{x}$. Each of these rules create specific transitions that could be classified to be of a certain type. For the definition of p-nets this classification isn't strictly necessary; the original definition in [Bal03, Bal04] does without it. However, in some contexts it's helpful to refer to a transition type in order to establish certain facts without having to restate them every time. For instance, transition label and the size of the $Preset$ and $Postset$. Also, we refer to members of $Preset$ and $Postset$ as pre- and postplaces respectively.

One should also note that labels for composition transitions are different. We make a necessary distinction between the arc going to the left postplace and the arc going to the right postplace with the respective l and r label. In Figure 1 we show the p-net created by an agent constant definition of the form $A \triangleq x.0|((\bar{x}.0|A)\backslash x)$ and this example shows how we label transitions and arcs.

2.2.4 Transition Classification

The function $translate()$ is applied recursively to generate a (partial) set of transitions. We now classify the transitions given by this function, rule by rule as numbered above, into types and denote a type x by $\mathcal{T}_x$.

1. Agent constant definitions creates so called *empty transitions*, denoted by $\mathcal{T}_\epsilon$. The transitions are labelled with the empty label ϵ.
2. The summation operator also creates *empty transitions*, but with the difference of creating two transitions each time this rule is applied.

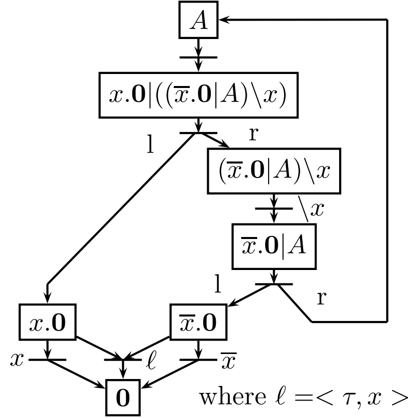

Figure 1: Generated p-net from the expression "$A \triangleq x.0|((\bar{x}.0|A)\backslash x)$"

3. The sequence operator creates so called *prefix transitions*, denoted by $\mathcal{T}_\alpha$. The transitions are labelled with α. When a distinction is necessary we use $\mathcal{T}_x$ for prefix transitions created by a "$x.P$" term, and $\mathcal{T}_{\bar{x}}$ for prefix transitions created by a "$\bar{x}.P$" term.

4. The restriction operator creates so called *restriction transitions*, denoted by $\mathcal{T}_{\backslash x}$. The transitions are labelled with $\backslash x$, where x is the restricted *channel*.

5. The composition operator creates so called *composition transitions*, denoted by $\mathcal{T}_|$. These transitions has one preplace but two postplaces, and have labels l and r on the arc going to the left and right postplace respectively.

2.2.5 Synchronizations

The second step of transition creation provides for possible handshakes in the concurrent model described by the CCS_k specification. The transitions created by this step are aptly named *synchronization transitions*, denoted δ_x. Synchronization transitions are labelled with $< \tau, x >$ where $x \in \mathcal{N}$.

We indicated previously that the function *sync*() is applied to the partial set of transitions created in the first step. However, we redefine the domain of this function to be pairs of prefix transitions with complementary labels. For each pair of complementary prefix transitions (t_1, t_2) the matching *sync*() function (according to transition type) is applied:

$$
\begin{array}{rcl}
sync(\mathcal{T}_x, \mathcal{T}_{\bar{x}}) & \Rightarrow & \Big(\Big\{pre(\mathcal{T}_x), pre(\mathcal{T}_{\bar{x}})\Big\}, \Big\{post(\mathcal{T}_x), post(\mathcal{T}_{\bar{x}})\Big\}, \langle\tau, x\rangle\Big) \\
& or & \\
sync(\mathcal{T}_{\bar{x}}, \mathcal{T}_x) & \Rightarrow & \Big(\Big\{pre(\mathcal{T}_{\bar{x}}), pre(\mathcal{T}_x)\Big\}, \Big\{post(\mathcal{T}_{\bar{x}}), post(\mathcal{T}_x)\Big\}, \langle\tau, x\rangle\Big)
\end{array}
$$

The $pre(t)$ and $post(t)$ gives the (singular) place in the preset and postset of transition t respectively. Note that we use transition classification as specified

in the previous subsection which restricts the pair of source transitions to prefix transitions with *complementary labels*.

Synchronization transitions are different from other transitions in the sense that we need to distinguish between the two preplaces in the preset, and likewise in the postset. We shall use the intuitive terms *left* and *right* to make this distinction. Also note that the preset always contain two places, whereas the postset may contain one if $post(\mathcal{T}_x) = post(\mathcal{T}_{\bar{x}})$.

The two cases above are intended to show, given two complementary transitions, $\mathcal{T}_x$ and $\mathcal{T}_{\bar{x}}$, that; a) only one synchronization transition is created and b) the synchronization transition's left preplace and left postplace correlate to the left argument of the $sync(left, right)$ function. Similarly for the right preplace and right postplace.

2.3 Firing Policy

The firing policy determines the cause and effect of a p-net ; when transitions may fire and the resulting state change. To help define this policy we need additional definitions for concepts like, *firing, marking,* and *tokens.*

Marking. A marking m is a mapping function from a place p to a set of tokens. We use m' to denote the marking after a firing of some transition t. When shown as M we denote the set of all markings $\{m(p) \mid p \in \mathcal{P}\}$, and M' would be the marking after a firing of some transition t.

Firing. A firing was originally defined in [Bal03, Bal04] as the triple (M, t, M') which encapsulates a state change for the entire net. We redefine a firing to be the tuple (t, Ω) where Ω is an *enabling candidate* for transition t.

Enabling candidates. Informally, an enabling candidate, denoted Ω, for a transition t is either a single token η or a pair of tokens (η_l, η_r) depending on the type of t. All transitions except synchronization transitions require a single token whereas a synchronization transition require a pair of tokens from its left and right preplace respectively.

Tokens. Tokens are entities that carry information about their firing history in the form of a string. The string is built from the alphabet $(\mathcal{N} \cup \{l, r\})^*$ where $\mathcal{N} \cap \{l, r\} = \emptyset$. This history accumulates during simulation as the token is part of a *firing* being fired. Henceforth, we shall use Θ to denote a set of tokens and η to denote a singular token.

Token operations. Token string concatenation is denoted by the $\bullet$ symbol, e.g. $\omega c = \omega \bullet c$ for some token string ω where $c \in \mathcal{N} \cup \{l, r\}$. We define the function $strip\mathcal{N}()$ which takes a token string as argument and gives the token string with all occurrences of $a \in \mathcal{N}$ removed. To clarify, for an arbitrary token η, $strip\mathcal{N}(\eta)$ gives a string of the alphabet $\{l, r\}^*$ retaining the relative order between l and r of η. For instance, if $\eta = xlryrlzl$, then evaluating $strip\mathcal{N}(\eta)$ would yield the string $lrrll$.

Marking operations. We denote token addition to a marking m by the $\oplus$ symbol, e.g. $m'(p) = m(p) \oplus \eta$. Conversely, we denote token subtraction from a marking m by the $\ominus$ symbol, e.g. $m'(p) = m(p) \ominus \eta$.

Place extraction. The functions $pre(t)$ and $post(t)$ are mappings from a transition t to its singular place p in the *Preset* or *Postset* of t respectively. If the *Preset* or *Postset* contains two places we shall use $pre_l(t)$ or $pre_r(t)$ to distinguish between the two and similarly for $post(t)$.

Max prefix. The function $maxpref(x, \omega)$ is defined to yield αx for the token string $\omega = \alpha \mathrm{x} \beta$ where $x \notin \beta$. This function yields the empty string ϵ in case $x \notin \omega$.

Note that $\bullet$ has greater precedence than $\oplus$ and also that a token may be seen equivalent to its "string", but the use of "string" doesn't strictly mean a literal character string, but rather a sequence of ordered items.

2.3.1 Enabling Candidates for Transitions

A transition is enabled by certain preconditions holding on the token(s) residing at the place(s) in the preset. All transitions have a base requirement of existence, which means that at least one token must reside on all places in the preset. Below, we specify how to determine, for each transition type, whether a token or token pair is an enabling candidate.

- For empty, restriction, and composition transitions; a token η is an enabling candidate if $\eta \in m(pre(\mathcal{T}))$.
- For prefix transitions $\mathcal{T}_x$ or $\mathcal{T}_{\bar{x}}$; a token η is an enabling candidate if $\eta \in m(pre(\mathcal{T}_\alpha))$ and $maxpref(x, \eta) = \epsilon$ holds.
- For synchronization transitions δ_x; the token pair (η_l, η_r) is an enabling candidate if $\eta_l \in m(pre_l(\delta_x))$ and $\eta_r \in m(pre_r(\delta_x))$ and additionally if
  - $maxpref(x, \eta_l) = maxpref(x, \eta_r)$ and
  - $strip\mathcal{N}(\eta_l) \neq strip\mathcal{N}(\eta_r)$

  both hold.

Let us review this informally. Empty, restriction and composition transitions have no additional precondition other than the existence of tokens on their preplace. Any prefix transition $\mathcal{T}_x$ or $\mathcal{T}_{\bar{x}}$ would require an enabling token whose string (or firing history) doesn't contain x. A synchronization transition requires a token pair, one from each preplace, with equal $maxpref()$ for x and non-equal $\{l, r\}$–order, respectively denoting two tokens in the same scope with respect to x, and two tokens belonging to different concurrent threads.

2.3.2 Firing Policy Definition

With the help of the previous section we can now define a function $pc()$ that implements the behaviour of the previous section. This function yields a token that qualifies as either an enabling candidate on its own or as a member of an enabling candidate based upon a token pair. We shall depict the function as operating on a set of tokens $pc(\Theta) = \eta$, but its behaviour is also dependent on the transition type. First some invariants

- $\forall p \notin pre(t) \cup post(t) \implies m'(p) = m(p)$. Any firing of transition t will leave places outside the pre- and postsets of t unaffected.

- $m'(pre(t)) = m(pre(t)) \ominus pc(m(pre(t)))$. The tokens specified by the $pc()$ function are removed from the places in the preset of t.

The firing policy per transition type is as follows

- For an empty or prefix transition t, the firing policy is

$$m'(post(t)) = m(post(t)) \oplus pc(m(pre(t))).$$

- For a restriction transition t, the firing policy is

$$m'(post(t)) = m(post(t)) \oplus pc(m(pre(t))) \bullet c$$

  where $c \in \mathcal{N}$.

- For a composition transition t, the firing policy is

$$\begin{aligned} \eta &= pc(m(pre(t))) \\ m'(post_l(t)) &= m(post_l(t)) \oplus \eta \bullet l \\ m'(post_r(t)) &= m(post_r(t)) \oplus \eta \bullet r \end{aligned}$$

  for left and right postplace respectively. Note that only one token is actually fired from the preplace, whereas two tokens are added to the postplaces, implying token creation.

- For a synchronization transition t, the firing policy is

$$\begin{aligned} m'(post_l(t)) &= m(post_l(t)) \oplus pc(m(pre_l(t))) \\ m'(post_r(t)) &= m(post_r(t)) \oplus pc(m(pre_r(t))) \end{aligned}$$

  for left and right postplace respectively. Note that if $post_l(t) = post_r(t)$ we add both tokens to the same postplace.

This concludes the definition of original p–nets, for completeness, an adequacy theorem is given in the appendix.

2.4 P-Net Imperfections

We shall now examine the finding of imperfections in the definition of p–nets, and these shortcomings will lead up to the derivation of guarded p–nets.

2.4.1 Early Termination

A problem with p-nets was symptomized by early termination of the simulation. During simulation, a situation could occur when the algorithm deduced (albeit correctly) that no more firings were available, and thus terminated the simulation. The situation could occur when a synchronization transition had (one or more) preplaces that were acting as operands (or part of operands) in

summations. Take the expression $(y.Q + x.P) \mid (\bar{x}.P' + \bar{y}.Q')$ and the associated p-net in figure 2, along with the marking $m(y.Q + x.P) = \{\eta_1 = \text{“}xy''\}$ and $m(\bar{x}.P' + \bar{y}.Q') = \{\eta_2 = \text{“}xy''\}$. Assume restrictions on both x and y. A closer look reveals that the synchronization transition based on x is *dependent* on the choices of both summations. This is all well and true, however the simulation could end up in a state where neither synchronization transition (δ_y isn't shown) is enabled, e.g. $m(y.Q) = \eta_1'$ and $m(\bar{x}.P) = \eta_2'$. The cause for this simulation behaviour is that p-nets have *internal choice* semantics, meaning (in p-net terms) that the empty summation transitions are fired before and regardless of future transitions, e.g. the synchronization transitions. In order to solve this problem, we must introduce *external choice* semantics, and we do that in section 2.5.2.

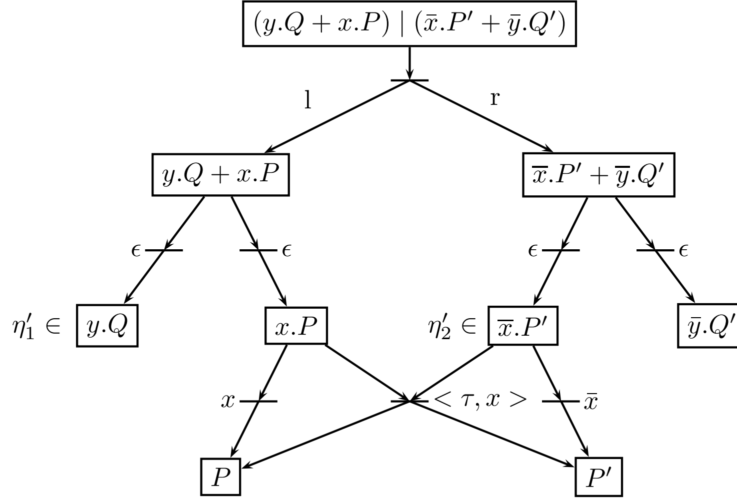

Figure 2: Showing (relevant transitions) in a possible state where no transitions are enabled.

2.4.2 Extraneous Synchronization Transitions

The process of creating synchronization transitions in [Bal03, Bal04] wasn't as restrictive as it could be, which resulted in a correct but suboptimal net. Extraneous transitions were created since the only requirement was that of complementary labels; a restriction on the preplaces didn't exist. A summation expression such as $a.A+\bar{a}.A'$ would generate a synchronization transition based on a. The actual firing of such a transition would be clearly wrong since a single thread cannot synchronize with itself; $a.A$ and $\bar{a}.A'$ doesn't occur in parallel. However, the firing policy was correctly specified so these transitions are never fired, which render them annoying rather than erroneous.

2.4.3 Redundant Precondition

The firing policy determines the preconditions to be satisfied by tokens in order to enable transitions. For a synchronization transition, two preconditions are required to be satisfied. During the thesis, it was found that the second precondition in section 2.3.1, $strip\mathcal{N}(\eta_l) \neq strip\mathcal{N}(\eta_r)$, was redundant. It can be shown that this precondition is always true.

2.5 Guarded P–Net: GP-Net

The motivation behind guarded p–nets, henceforth called GP-Nets, was the finding of the above shortcomings of p–nets. GP-Nets correct these by redefining the summation operator of the input language and the functions $translate()$ and $sync()$.

2.5.1 Guarded Summations

The input language is modified to allow guarded summations only. A guarded summation is of the form

$$\sum \alpha_i.A_i, \text{ where } \alpha \in \mathcal{A} \text{ and } A_i \text{ is an agent constant or sub term.}$$

Thus the arity is potentially infinite but the operands are restricted to be guarded terms. For instance, $P = A + \bar{b}.B$ isn't a guarded summation (A is unguarded) but $P = a.A + c.(\tau.B + \tau.P)$ is. The modified syntax of agent constant definitions is

$$\begin{array}{rcl} P \stackrel{\text{def}}{=} & 0 & \textit{(unmodified)} \\ | & \alpha.P & (\textit{abbreviated}) \\ | & \alpha.P + \beta.Q & (\textit{behave as one of } \alpha.P \textit{ or } \beta.Q) \\ | & P \mid Q & (\textit{unmodified}) \\ | & P \backslash x & (\textit{unmodified}) \\ | & A & (\textit{unmodified}) \end{array}$$

Restricting the input language isn't of great concern since in most applications guarded summations is the only type of summation used.

2.5.2 Redefinition of *translate*() and *sync*()

The $translate()$ function is changed with respect to summations and prefix terms. Referring to the numbered rules in section 2.2.3 on Transitions, rule 2 and 3 are replaced by

$$translate(\sum \alpha_i.A_i) \quad \Rightarrow \quad \left\{ \bigcup \Big(\{\sum \alpha_i.A_i\}, \{A_i\} \Big) \mapsto \alpha_i \right\}$$

which considers the term $\alpha.A$ to be a unary summation. It's important to note that the empty transitions created for p-nets have been removed, and the prefix transitions are now outgoing from the place associated with the summation term. This effectively removes the problems associated with internal choice and

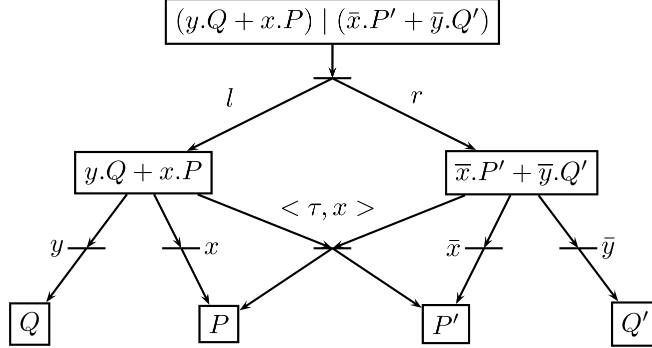

Figure 3: The GP-Net of the previous p–net in figure 2. The synchronization transition based on y is not shown.

early termination because these transitions were the cause of the problems. GP-Nets have *external choice* semantics. In figure 3 we show the GP-Net version of the p-net in figure 2 to indicate the modified structure.

We also modify the *sync*() function by adding a restriction on the preplaces for (complementary prefix) transitions t_1, t_2;

$$\begin{aligned} \forall(t_1, t_2) \quad &\in \quad \{(t_1, t_2) | pre(t_1) \neq pre(t_2)\} : \\ sync(t_1, t_2) \quad &\Rightarrow \quad \Big(\Big\{pre(t_1), pre(t_2)\Big\}, \Big\{post(t_1), post(t_2)\Big\}, \langle\tau, a\rangle\Big) \end{aligned}$$

where a is the common *channel* on which t_1 and t_2 occur.

2.5.3 Precondition Enhancement

Last but certainly not least, we remove the redundant precondition for synchronization transition enabledness which greatly reduce the complexity of the firing policy for synchronization transitions.

3 Petri Net Simulation

Petri net simulation has been around since their birth in the early 60's [Pet62, Rei98]. Simulation theories and accompanying tools exist in abundance. Due to the assortment of Petri net variations, there exists many strategies for simulation but their diverse nature doesn't lend well for a generic implementation for all types of Petri nets. However, what may be universally agreed upon is that basic Petri net simulation may be realized by a sequential discrete event simulation architecture [Eva93, Cao90, Haa89, vH89], where the simulation is performed on one processor as opposed to parallel/distributed discrete event simulation for Petri nets as described in [Cho92, Chi93, Gil98, Nke94].

3.1 Simulation Example

To give a notion of the process of simulation we show a sample simulation run of the example p-net (which also is a valid GP-Net) in figure 1 on page 10. For brevity, we denote a place i as P_i where i is an assigned index as in figure 4. We describe a simulation state, j firings into the simulation, by the marking

$$M_j = \{m(P_1) = \eta_\omega, m(P_3) = ..., m(P_5) = ..., m(P_6) = ...\}$$

where η_ω denote a token with token string ω. We neglect to show the empty marking for places, in this case P_2, P_4, and P_7. One particular simulation run

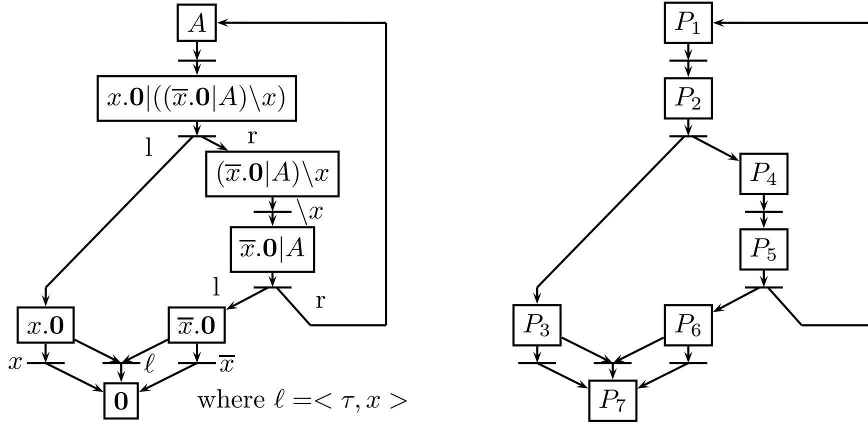

Figure 4: Showing the assigned indices of places on the right.

could behave as follows; stepwise showing seven firings into the simulation

- The initial starting state of the simulation. The marking of the start place P_1 has a single token with an empty string.

$$M_0 = \{m(P_1) = \eta_\epsilon\}$$

- The state after two firings, showing the result of the first composition transition firing. The tokens qualify as enabling candidates for the action transition and restriction transition respectively. The synchronization transition isn't enabled because its right preplace doesn't contain a token.

$$M_2 = \{m(P_3) = \eta_l, m(P_4) = \eta_r\}$$

- After firing the restriction transition note the change in the token string.

$$M_3 = \{m(P_3) = \eta_l, m(P_5) = \eta_{rx}\}$$

- Firing the second composition transition we have a state where the synchronization transition's preplaces both have tokens. However, it's still not enabled because a precondition on the tokens doesn't hold, see section 2.3.1.

$$M_4 = \{m(P_1) = \eta_{rxr}, m(P_3) = \eta_l, m(P_6) = \eta_{rxl}\}$$

A new token has also been spawned and we follow its path onward.

- The first composition transition has fired a second time at six firings into the simulation and three different transitions are enabled.

$$M_6 = \{m(P_3) = \{\eta_l, \eta_{rxrl}\}, m(P_4) = \eta_{rxrr}, m(P_6) = \eta_{rxl}\}$$

  The synchronization transition now has the enabling candidate $(\eta_{rxrl}, \eta_{rxl})$.

- After firing the synchronization transition we have

$$M_7 = \{m(P_3) = \eta_l, m(P_4) = \eta_{rxrr}, m(P_7) = \{\eta_{rxl}, \eta_{rxrl}\}\}$$

  where the tokens in P_7 represents terminated threads.

The behaviour of this specific net is rather simple. Threads represented by tokens on the right side of the net will continuously spawn threads which will later synchronize before termination. The only thread which may terminate alone without synchronization is the first thread in the left side of the net.

3.2 Discrete Event Simulation for GP–Nets

Discrete event architectures has have similar names such as system, scheduler or simulator depending on the application, but for the most part the abbreviation DES is used. For a theoretical approach read [Zei00]. A DES architecture usually include the following components

- A scheduler responsible for selecting events according to some application defined order, chronologically for instance, whereupon they are executed in that order.
- A simulation state which contains the variables of that we wish to simulate.
- and the actual events which operate upon the simulation state maintained by, for instance, a priority queue.

In addition to these, the architecture may include a simulation engine when the complexity increases, and one wishes to separate the *what* to do and the *how* to do it. In this case the events deal with *what* to do and the engine deals with *how* to do it. Basic discrete event simulation for GP-Nets would use firings as the events, the marking M as the simulation state and could use a simple loop for a scheduler. For a simple and correct simulation, the engine component isn't strictly necessary.

3.3 A Naive DES algorithm

We present here a naive algorithm in pseudo–code which we deem to be our starting point regarding optimization. Initially a token would be injected into the starting place of the net, after which the loop is entered. The loop would first derive all possible events from the marking M and stop simulation if none exist. Otherwise it continues by selecting a random event for execution which would effect the state changes of the marking M, according to the firing policy.

```
% initializations
T := <all transitions>
```

```
M := <initial marking>
E := <empty set>
while(true)                % main simulation loop
  foreach t in T           % event derivation
    if enabled(t,M) then   % if enabled
      E := firings(t)      % add to set
    endif
  endfor
  if empty(E) then stop  % stop if no events
  endif
  f := random(E)         % pick random firing
  M := fire(f,M)         % fire and change M
endwhile
```

The above algorithm would be a naive but correct implementation. However it does nothing to reduce the search space for event derivation nor does it use any domain knowledge of the net structure specific to GP–Nets. It was the main purpose of this thesis, to tailor the above naive algorithm to the specific task of simulating GP-Nets in the most optimized and efficient way. We achieved this by reducing the problem size, decreasing (if not minimizing) the search space for event derivation, and by making event execution as efficient as possible. This was made possible by a detailed analysis of the task of simulation, which gave us the insight and domain knowledge needed for the aforementioned optimizations.

4 Optimization

When optimizing a program the chief purpose is to be faster and we generally do that by reducing computation somehow. We may achieve this by removing useless computation, postponing unnecessarily early computation, or perhaps caching results of frequent and costly computations to avoid computing the results every time they are needed. A subtle technique may also be used which involves precomputing values to avoid computation overhead where the results are used.

Removing computation is by far the best optimization even though it may not result in the greatest performance gain. Postponing computation until it's actually needed is useful since the program may branch away from the part of the program that requires the result of that computation, it's then similar to having removed the entire computation. Precomputation has subtle gains which may depend on the compiler and hardware platform, caching on the other hand has obvious benefits.

The above techniques are often described as algorithmic optimizations which from this thesis' perspective may be the most interesting since they pertain specifically to the simulation algorithm. However there are other, equally good, optimizations of a more general category such as memory handling, code inlining and proper use of containers with beneficial asymptotic bounds.

The following sections will contain a description of the optimization techniques utilized in the tool and also point out key properties of the simulation situation that make them both necessary and applicable.

4.1 Four Tasks of Simulation

Taking a closer look at the pseudo-code of the described DES algorithm in section 3.3 we should be able to identify parts of the algorithm which will be crucial to performance. We can narrow the algorithm down to four parts in order of increasing perceived complexity or time contribution:

Picking Picking a *firing* (to fire) should ideally be a constant time operation, or at the very most be linearly dependent on the number of enabled firings. This may rely on the representation and interface between a firing and scheduler.

Firing On the surface, firing is a simple matter. The complexity would be a combination of adding and removing tokens to and from places along with making the modifications (if any) to the token. Addition and removal have well–known complexities for different containers so it would depend on the container usage. Constant or logarithmic complexity can be expected.

Token Creation During simulation tokens may be created by the firing of a composition transition. The firing policy specifies that the two resulting tokens inherit the data of the enabling token with additional data appended to the end. This also implies a dual creation and copying operation since the two tokens are conceptually modified clones of the enabling token. In general, copying operations have linear complexity so this would be a function of token size. Allocation issues for new tokens shouldn't be ignored, recycling of deallocated tokens should be utilized whenever possible.

Enabling Enabling calculation is visibly, by far, the most expensive operation to be performed. Each simulation round of the pseudo–code algorithm, we iterate over all transitions to accumulate a set of enabled transitions (and their firings). Each candidate must be identified in order for all possible firings to be available to the scheduler for picking. This is different per transition type, some require posing queries on each token in the preplace. Synchronization transitions require candidates with token *pairs* which implies a quadratic search for matches.

4.1.1 Simulation State and Static Properties

From a firing policy perspective, the dynamic nature of simulation is manifested by the marking M and is thus considered to represent the simulation state. Theoretically, a previous simulation could be resumed given the same net and a marking. For the sake of understanding simulation we shall also state certain static properties which are emphasized by the restricted dynamic nature of simulation.

The linkage of a GP-Net never change. There are Petri nets which may change linkage, but GP-Nets isn't one of them. This property essentially dumbs down places to be simple placeholders for tokens and outgoing transitions.

The label of a transition never change. This obvious static property has huge implications for enabledness calculation which comprises most of the simulation computation.

The past in a token never change. This is a consequence of the firing policy and • operation which only appends data to the end of the token's string. This important fact in combination with how compositions are fired allows for optimizations like structure sharing and cache techniques. Alternatively, one could say; once a token (or thread) is restricted, it is always restricted.

Number of labels never change This static property allows us to decrease the amount of work to be done in our caching optimizations.

It should be apparent that static properties allows for assumptions to be made which may give rise to optimizations.

4.2 Optimizing the Naive Algorithm

The optimization techniques we describe in this section are made available due to the identified properties of GP-Net simulation described in previous sections. When describing the optimizations we shall briefly note why they are necessary (the preceding sections should give enough detail to derive the specifics), how they work more intimately and also, due to their nature, requirements for being advantageous.

4.2.1 The Locality Principle

Simulation of Petri nets in general exhibit a known locality principle concerning the marking and this principle holds for GP-Nets as well (it follows from the firing policy). The principle states that, only the marking of neighbouring places change due to the occurrence of an event. Alternatively, using GP-Net terminology, the firing of a transition will only effect the *enabledness state* of neighbouring transitions. A transition s is defined to be a neighbour to transition t if it's outgoing from a preplace to t, or outgoing from a postplace to t. The locality principle has three important implications.

1. We may narrow the search space for enabledness calculation to those transitions s that are neighbours to the transition we fire.

2. Firing transition t we know that neighbours that share the same preplace as t can only become *disabled* by this event (since tokens are removed from their preplace) and that neighbours whose preplace is a postplace of t can only become *enabled* by this event (since we are adding tokens to their preplace).

3. We may keep results of enabledness calculation from one round to the next for every transition which is not a neighbour to the transition we fire (they won't be effected).

These implications allow us to decrease the search space for event derivation (postponing computation until actually needed) and for caching enabledness computation. The enabledness state becomes a persistent internal state for each individual transition, rather than a transient state discarded at the end of a simulation round.

Information about enabledness is may thus be carried over the boundaries of the simulation loop, if we take care to rebuild this information *only* for those transitions effected by a firing (because of a neighbour relationship). This would reduce the size of the enabledness calculation significantly (the set of neighbouring transitions is most likely a lot smaller than the total number of transitions), and move it to after a transition has been fired and the marking updated. This would require access to the neighbour relationship for each transition during simulation (it's a static relation), and in order to preserve the result of enabledness calculation, extra data structures for record keeping would have to be maintained for each transition (updated each round for a transition if the neighbour relationship holds).

We may also split transitions into two sets, one for disabled transitions and one for enabled transitions and let the enabledness calculation move transitions in between these sets. This would remove the need to allocate and build a new enabled set as input to the picking algorithm. Large models may generate a great deal of transitions so it's important to choose suitable data structures for these two sets. Ideally, a container with fast deletion and fast insertion should be used. This would render non–random access data structures unsuitable (they have bad asymptotic bounds when searching) which leaves us with either a sorted array or associative data structures, such as a hash map.

4.2.2 Token History Information

The tokens carry information about their firing history and the definition of p-nets and GP-Nets both represents this "history" as a sequence (or string) of characters from the alphabet $(\mathcal{N} \cup \{l, r\})^*$. However, the firing policy allows members of $\mathcal{N}$ to be literal character strings, but performing string allocations and string operations on tokens (such as concatenation, comparison, etc.) during simulation would be a waste of memory, not to mention extremely time consuming. Especially so for recursive nets where token size may grow rapidly.

A simple solution to this apparent problem is to (instead of using literal strings) assign a unique identifier to each member of $\mathcal{N}$ including l and r and then represent the token firing history as a singularly linked list, called *token data list*. New data is added to the head of the list and each node would contain the identifier corresponding to a name. Checking the precondition, for say a restriction, would be reduced to finding a node in the list with the same identifier as the restriction label. Still, this is a linear operation but we shall reduce this further.

4.2.3 Token Creation

Tokens are created when a composition transition fires. The two resulting tokens should inherit the firing history (i.e. token data list) of the enabling candidate, along with adding l and r respectively to the head of the token data list. This operation is depicted in figure 5 with enabling token η = "xyz" and resulting tokens η_l and η_l. What is evident in figure 5 is the redundant information in both η_l and η_r, they contain the same substring "xyz" as is correct and specified by the firing policy. If one assumes that η is not used as one of the resulting tokens η_l or η_r, then allocations were made for two tokens and η's data was copied (linearly) twice.

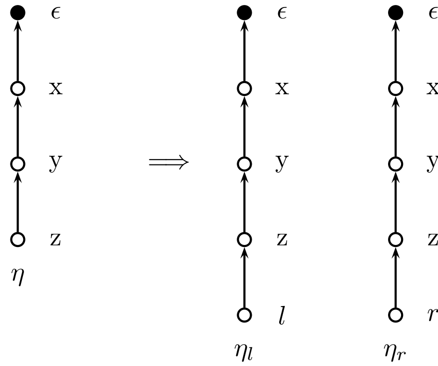

Figure 5: Showing the effect of firing a composition transition. The beginning of the firing history is denoted by ϵ. Note the reverse direction of arrows and order of labels.

We may eliminate one token creation by using η as one of the resulting tokens, say η_l, and add l. We may also realize structure sharing by initializing η_r's token data list with the head of η and then add r, as indicated in figure 6. Structure sharing is possible without conflicts and side-effects for three reasons.

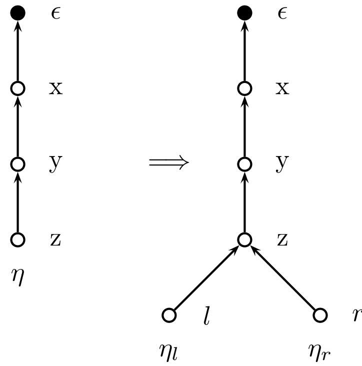

Figure 6: The result of using structure sharing. No copying is performed for the token data list and redundancy is eliminated.

First, the token data list is a singularly linked list in only one "direction". Second, all operations on the token data list, except addition, only requires traversal from head to tail. And third, we know that the "past" in a tokens firing history (i.e., token data list "xyz") is static and will never change for either of the two tokens.

By implementing structure sharing for tokens, the firing history for all tokens in the net (or more appropriately, the marking M), may be visualized by a semi-binary tree. By semi-binary we mean that some nodes only have one child node (due to restriction transitions) and other nodes have two child nodes (due to composition transitions). This is only a conceptual tree, since the tokens

have no notion of sharing whatsoever, but we may use this visualized tree as an alternative way of describing why and how some optimizations work. An example tree is shown in figure 7 and explained further in the next section. We can make two observations about such a tree. First, the number of tokens in the net equals the number of leaves in the tree, which means each leaf node is the head of the token data list for exactly one token. Second, the sequence of nodes generated by traversing from a given leaf node up to the root is equivalent to the token at that node (duly note that this would give the firing history in *reverse* order). It can also be verified, from the nature of how this tree is built, that no two tokens will have the same sequence of l and r nodes in such a chain because that would put the two tokens in the same leaf node. This strengthens (but doesn't formally prove) the claim in section 2.4.3.

4.2.4 Caching $maxpref()$

A synchronization transition δ_x were identified as having the most complex enabledness calculation. It entails a quadratic search of token pairs. Informally, for p-nets the requirements specify that the tokens must belong to different concurrent threads and they must be in the same scope with respect to x, the channel upon which the synchronization occur. However, for GP–Nets, only the latter requirement must be fulfilled, see section 2.5.3.

Given a synchronization transition δ_x occurring on channel x, the requirement for an enabling candidate ($\eta_l \in m(pre_l(\delta_x))$, $\eta_r \in m(pre_r(\delta_x))$) is that $maxpref(x, \eta_l) = maxpref(x, \eta_r)$. If we have a situation as described in figure 7 with the indicated position of η_l, then only the tokens corresponding to the leaf nodes in the encircled subtree satisfy the $maxpref(x, \cdot)$ requirement. It should be observed that these two tokens would also have to reside at the right preplace of the synchronization transition in question, something which can't be derived from the tree.

The enabledness calculation creates all possible pairs of tokens (one from left preplace, the other from the right preplace) and compares the result of the $maxpref(x, \cdot)$ function. The $maxpref(x, \cdot)$ function performs a list traversal of the token data list and returns the first node containing x. It should be apparent that only the tokens in the encircled subtree would qualify.

Thus enabledness calculation for a synchronization transition with m tokens on the left preplace and n tokens on the right preplace would require $m * n$ list traversals to find all possible firings. We can speed this up if we note the fact that enabledness calculation for synchronization transitions only cares about the *first* occurrence (from leaf to root) of some x (and always x because the label argument to $maxpref()$ never change for a given synchronization transition). By utilizing a cache that maps a given token η and name x to the first token data list node of η containing x (or alternatively, to the node of greatest depth containing x on the path from root to leaf node corresponding to η), the $maxpref()$ function would be reduced to constant time lookup instead of linear time lookup. The cache would have to be updated whenever a token is modified by the firing of a restriction transition. Given the fact that number of labels (or names) in a net never change, we may assign a unique identifier in the range [0–n] for each label and implement the cache with a simple array. In contrast to the token data lists, we would need to copy the cache upon token creation, bit by bit, so we should try to reduce the size of the cache. It turns out this is possible but

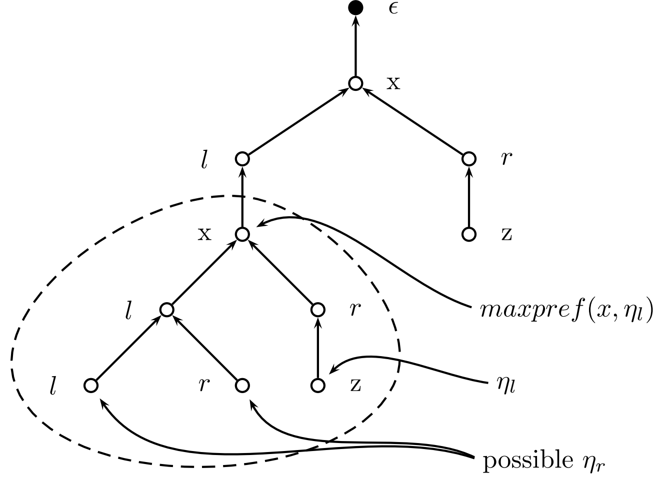

Figure 7: A conceptual semi-binary tree depicting structure sharing.

we need to categorize the labels of a net in order to achieve this.

4.2.5 Label Categorization

The use of certain CCS_k algebra constructions inside recursion have special effects on the net behaviour. Most easily identified is a composition construction inside recursion which removes the upper bound on the number of tokens. However, the fact that we know the upper bound on tokens is hard utilize for optimizations. We present a speculative optimization in section 6.7.

The usage of restrictions in the CCS_k specification does allow us to categorize the labels they create in translation to net components, and subsequently we may optimize transitions based upon a certain category of labels. Lets assume the following environment of (comma separated) agent constants:

$$P = z.(A|B)\backslash x,\ A = (y.P)\backslash y, \text{ and } B = (\bar{y}.P)\backslash y$$

where P is the starting place.

Unrestricted labels. Any unrestricted CCS_k name in a specification will be translated into a label which falls into the category of unrestricted labels. For instance, the CCS_k name z in the above environment.

Globally restricted labels. Any CCS_k name only restricted above recursion *and* composition will be translated into a label which falls into the category of globally restricted labels. For instance, the CCS_k name x in the above environment.

Infinitely restricted labels. Any CCS_k name restricted inside the path of recursion will be translated into a label which falls into the category of infinitely restricted labels. For instance, the CCS_k name y in the above environment.

These categories are distinct and all labels of any CCS_k specification fall into one of these categories.

- If a prefix transition ($\mathcal{T}_x$ or $\mathcal{T}_{\bar{x}}$) is based upon an unrestricted label it will alleviate enabledness calculation since the firing policy impose requirements on the tokens with respect to the label. The firing policy states that an enabling token shouldn't be restricted on the label for prefix transitions, and if the transition is based upon an unrestricted label then we know it isn't. This in effect nullifies the firing policy requirement and makes all tokens, residing at the preplace, qualify.

- Globally restricted labels have similar implications for synchronization transitions as unrestricted labels have for prefix transitions. This might not be obvious, but referring to a tree such as the one in figure 7, we would see all globally restricted labels as chain of single children from the root node down to the first branch made by a composition transition (and only at this level, nowhere else). Though in the case of figure 7 there is only one label which happens to be infinitely restricted (and not globally restricted). A synchronization transition based upon a globally restricted label would have a respective subtree, such as the one encircled in figure 7, containing all the tokens in the net, meaning all possible pairs would qualify for enabling candidacy.

  Furthermore any prefix transition based upon a globally restricted label would never find a token that enables it because all tokens would have this restriction. These transitions may then be culled from the simulation, reducing the workload.

- As for infinitely restricted labels they represent the generic case, of which the two items above are special cases, since no assumptions can be made for the tokens. However, we can reduce the size of the previously mentioned $maxpref()$ cache to only consider infinitely restricted labels, since these are the only labels that may be added to a token (except l and r).

From the categorization of labels we may reduce the total number of transitions to consider during simulation. For prefix transitions and synchronization transitions based upon appropriate label category we may replace the previous enabledness calculation. Prior to this optimization, we maintained record keeping data structures for both transitions types as a result of the locality principle. We may now discard these data structures and the expensive maintenance because all tokens and pairs of tokens are enabling candidates for prefix transitions and synchronization transitions respectively. The "ripple" effect for neighbouring prefix and synchronization transitions (based on appropriate label) are thus reduced greatly since we don't have to update the record keeping data structures by deleting (previously) enabling candidates, or redoing enabledness calculation because a preplace has gained a new token.

4.2.6 Memory Handling

There are three main dynamic data structures in a simulation that needs special care. The token data structure, the node data structure (for token data lists), and the record keeping data structures for those transitions requiring it. The

first two may grow rapidly depending on the net, the third is potentially updated very often. This cause for memory handling issues to be addressed. The main issue to be addressed is slow standard language OS allocation/deallocation routines. These routines needlessly spends time considering memory fragmentation, multi–threading issues and gives no control of spatial locality of consecutive allocations which serves for poor cache hit ratio.

By implementing a memory pool, which preallocates a fixed amount of contiguous memory, we can decrease the number of allocations made to the underlying runtime system. In addition, when tokens are considered to no longer be *live*, by ending up at a zero place, we may recycle the memory of the token by simply releasing it back to the memory pool, instead of expensive deallocation. Specializing the memory pool to only support allocations of a certain size the maintenance routines are simplified and fragmentation issues become trivial.

5 Evaluation of Optimization Techniques

This section shows measurements of three GP-Nets with different characteristics in order to show the performance of GPSim with various optimization techniques turned on or off. We begin each subsection by discussing the properties of the input net (in the form of a CCS_k specification) and which techniques should and shouldn't improve the performance followed by measurements in the form of graphs.

Since all techniques were implemented over an extended period of time and some involved major structural code changes, it wasn't possible to incorporate an on or off capability for each technique. Such a feature would allow for individual speedup measurement for each technique but also create a code management nightmare and quite possibly introduce more bugs. Thus the optimization based on the locality principle and structure sharing are always utilized.

The CCS_k specification used for measurements are shown in GPSim's syntax, refer to section 8.2 for details of the input language.

5.1 A Token Infinite Net

We use a token-infinite net for our first simulation test case, the net from figure 1. This net is a very small recursive net with only seven transitions. It will create a lot of tokens very fast, but it also has a zero place which may be used for recycling non–live tokens. It has no globally restricted labels, only one synchronization transition based on the one and only, infinitely restricted, label.

```
constants A, 0;
actions x;
A = [x.0]|((^x)(['x.0]|A))
```

Given the size of the net, it's expected that a simple data structure for transition sets, like an array, would achieve better performance than a more complex data structure. Measurements support this expectation, so we do not show a curve with the hash map optimization turned on.

Each curve of the graph in figure 8 was constructed by turning on one or more optimizations. Then we performed several simulation runs, increasing the

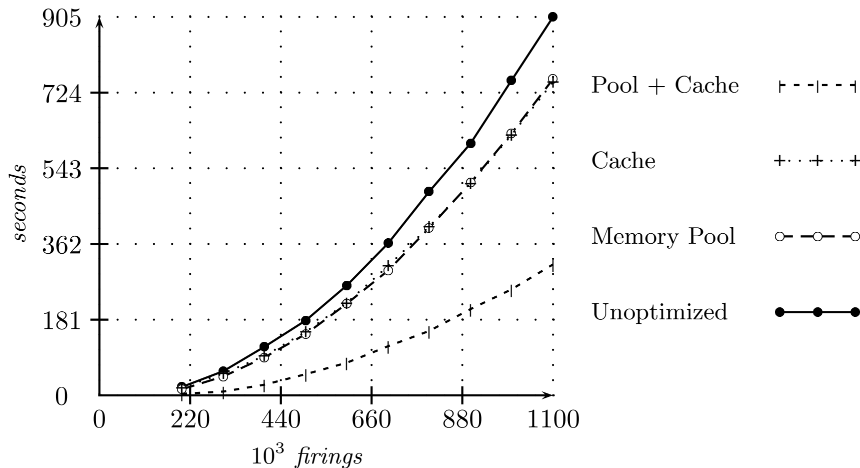

Figure 8: Simulation of a token infinite net.

number of rounds simulated each run, allowing the number of tokens and their size grow. The graph has four curves representing different optimizations turned on or off. What can be seen for this particular net is that the memory pool and cache seems seems to have the same speedup vs the non optimized case. Both curves practically lies on top of each other, and considering that they share no common code puts the observed measurements under suspicion. A thorough investigation of the test case was performed and a more detailed test case was performed as well, indicating no mistake has been made. It appears that the equivalent speedup from the cache optimization and memory pool optimization is a result of the net structure. As we shall see in section 5.3 the cache optimization and memory pool may give widely different speedups for other types of nets. However, one interesting and subtle effect of the memory pool on the cache is shown when both these optimizations techniques are used together. This is somewhat surprising since the cache never performs allocation or deallocation of memory pool objects. The conclusion drawn is that that the improved spatial locality of the memory pool objects improve on the performance of the cache because of better hit ratio in the CPU *hardware* data cache.

One important issue concerning memory use was discovered during measurements. For token-infinite nets the memory consumption may increase radically if the net doesn't contain a zero place where token recycling occurs. There is no other way for tokens to be recycled, since they are considered live in any other case and will remain in memory.

5.2 A Token Finite Net

For this measurement we use as input the specification used in the case study, section 6.4, where we describe this specification in detail. But to emphasize the behaviour of our optimization techniques we only deal with the GP-Net characteristics here as support information for the measurements.

We shall perform the simulation different this time. Instead of increasing the number of firings to simulate, we increase the actual size of the net (in transitions), a process also described in section 6.4. We stop the simulation once a specified stop criteria has been met. The result of the measurements are shown in two separate graphs.

The net created from the specification is token-finite with a constant token size. Additionally, all labels may be categorized as either unrestricted or globally restricted. We may expect the memory pool speedup to be small since tokens are constant in both number and size, the latter reason also giving us cause to expect the cache to be of little use. Since we increase the net size in transitions, we expect the choice of data structure for the transition set to be of major importance. We also expect label categorization to play a large role since this may remove transitions that will never fire, in addition to speeding up enabledness calculation for synchronization transitions. The graph in figure

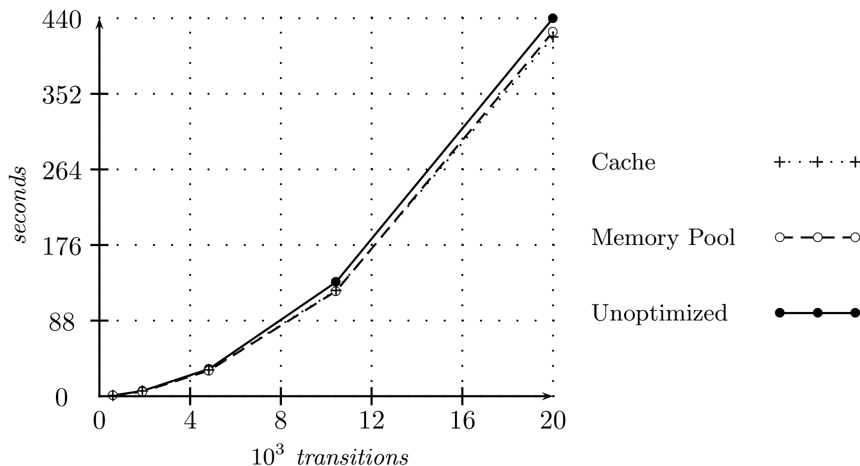

Figure 9: Simulation of token finite net.

9 shows two optimized curves that behave as expected for the optimizations turned on. The memory pool speedup is marginal, as is the speedup for the cache. One might ask why we get a speedup at all from the cache. This is because label categorization is not turned on, which means the cache will have entries for all label used in restrictions (which, by default, are considered to be infinitely restricted). The entries in the cache are inserted once, and the cache is never updated again, but it is still *queried* during simulation, which accounts for the speedup. It can be shown when label categorization is turned on, the cache optimization gives no benefit for this particular net.

The graph in figure 10 shows four optimized curves plus the unoptimized curve for reference. It's easy to verify that the choice of data structure for the transition set is of greatest importance, the curve outperforms the label categorization optimization. The curve for label categorization shows that about a quarter of the transitions have been shaved off, which most likely is the cause for its performance. However it's obvious that, even though it's faster than the

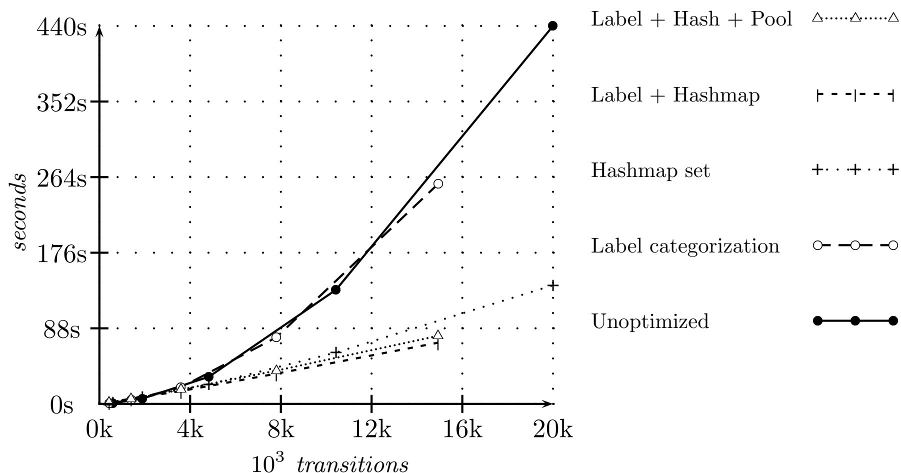

Figure 10: Simulation of token finite net.

unoptimized case, they share a similar derivative. The two other curves indicate increased speedup when combining the label categorization and hash map but a small penalizing overhead when the memory pool is also turned on.

5.3 An Experimental Net

The nets simulated so far has been both token-finite and infinite with constant or variable token size and ranged from small to large nets. None of these nets represents a worst-case scenario regarding enabledness calculation for synchronization transitions due to a relatively small number of tokens appearing at the preplaces of synchronization transitions. An artificial, token-infinite net, was constructed with many synchronization transitions where the preplaces of synchronization transitions would fill up with possible candidate pairs quickly. This net is a modified version of the case study net and is in all ways a valid net, but doesn't share any of the original semantics.

```
constants P,M,MAC,MACs,Mc,MACx,MACy;
actions s,er,c,br,md,r,e,b,collision;
P = (^c)(^b)(^br)(^e)(^er)(^md)(MAC|MAC|M)
MAC = [~.MACx+~.MACx+~.MACx+~.MACy]
MACy = MACx|MACx
MACx = [s.MACs+br.[c.MAC+er.[r.MAC+s.r.MACs]+s.[c.MACs+er.r.MACs]]]
MACs = ['b.[c.MACs+'e.[c.MACs+md.MAC]]+br.[c.MACs+er.[c.MACs+r.MACs]]]
M = [b.[b.Mc+'br.[b.Mc+e.[b.Mc+'er.'md.M]]]]
Mc = ['c.'c.M]
```

The net isn't dynamic in all ways because it doesn't have any infinitely restricted labels, but can easily be modified with dummy restrictions to achieve growth of token size as well. However our purpose with this net is to measure a worst–case scenario with quadratic complexity for synchronization transitions, and the cache optimization (which is aimed to alleviate simulation of nets with large tokens) does not effect this search space.

That being said, this net is very dynamic with regards to the necessary record keeping of enabling candidates. So the memory pool is expected to be efficient, contrary to the cache optimization. A simple array for data structure is most likely the optimal choice, and since the label categorization optimization is aimed at enabledness calculation we expect it to show a positive speedup.

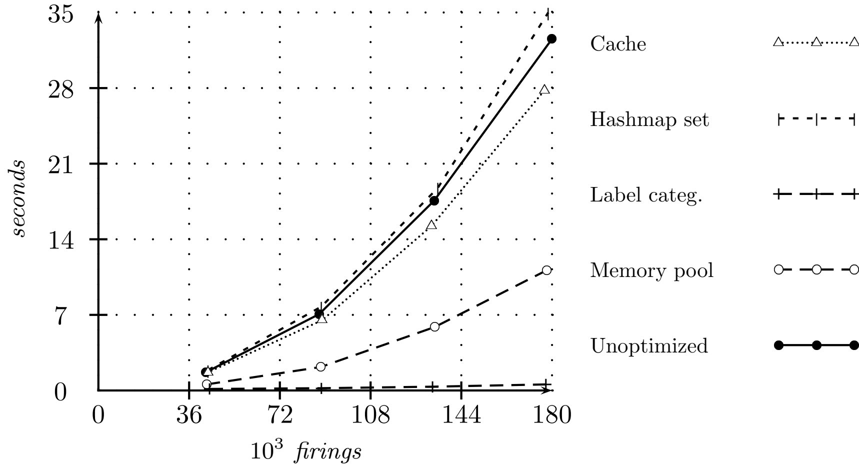

Figure 11: Simulation of net with quadratic search space.

The graph in figure 11 has four curves for the individual optimizations turned on, and one curve for the unoptimized case. We see that a hash map is the wrong choice since we're only dealing with a 74 transition net. We see that the cache provides a small speedup, this for the same reason as for the measurements of figure 9. The memory pool optimization does a great job because of the growing simulation state, not only tokens and their size but mostly for record keeping structures for synchronization transitions. But the label categorization outperforms all other optimizations. It is quite hard to spot, being almost parallel to the x–axis one might doubt its correctness. The explanation for its speedup is twofold. First, the net is comprised mostly of synchronization transitions with a reduced enabledness calculation. These transitions can assume that all token pairs from their preplaces qualifies as enabling candidates because they have equal $maxpref()$ for the label which the synchronization transition is based upon (a globally restricted label). Second, the previous assumption makes record keeping of any kind unnecessary and this reduce memory consumption by several magnitudes, shown in figure 12.

6 Case Studies

In this section we compare GPSim with two well–known programs for model checking and verification of concurrent systems. The most challenging tool of these was Spin, its long history of development and use have resulted in a very qualitative tool which also features a fast simulation mode (if not considered

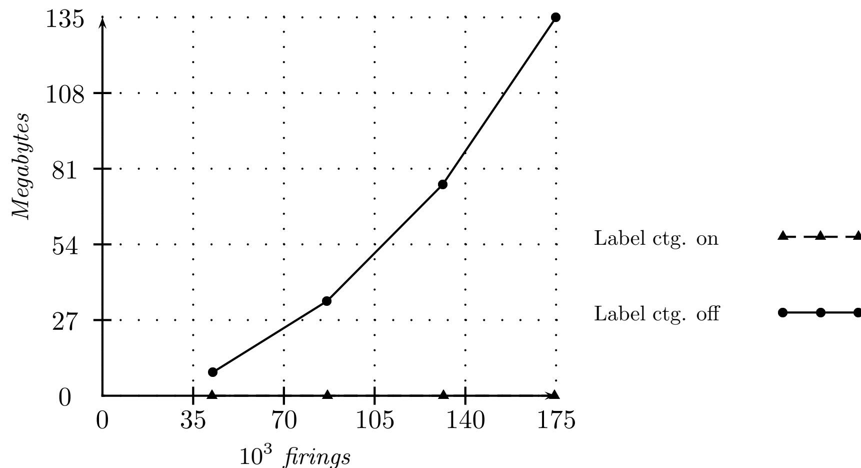

Figure 12: Showing effect of label categorization on memory allocations.

the fastest available). The Concurrency Workbench of the New Century, abbreviated CWB–NC, on the other hand features only a basic simulation mode in addition to verification functionality though it is still considered a respectable tool in the field. CWB–NC was chosen because it is implemented in SML of New Jersey in contrast to both Spin and GPSim (C and C++ respectively) and it also supports a wide range of languages, including CCS. The author is not an expert on either tool so a brief overview of the tools are presented in the next two sections and following sections deals with the comparisons.

6.1 CWB–NC Overview

The following paragraph is an excerpt from CWB–NC's official homepage at `www.cs.sunysb.edu/~cwb/`

> *"The Concurrency Workbench of the New Century (CWB-NC) provides users with a number of different techniques for specifying and verifying finite-state concurrent systems. The tool combines support for several different system-design notations with decision procedures for a number of refinement relations and for determining if systems satisfy temporal formulas. The design of the tool facilitates customization, and a related tool, the Process Algebra Compiler, may be used to specialize the CWB-NC to new design notations and to add new features to existing ones. Since the original release in 1996, the CWB-NC has been acquired by 350 different groups and has been applied to several large-scale case studies in the areas of communications protocols and process-control systems."*

6.2 Spin Overview

The following paragraph is an excerpt from Spin's official root web page at `spinroot.com`

> *"Spin is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM."*

Spin has its own high level language called PROMELA, which is a non–deterministic language loosely based upon Dijkstra's guarded command language notation and borrowed notation from Hoare's CSP for I/O operations.

6.3 Case Study Details

For the case study we chose a simple CCS model for a two station MAC–sublayer communications protocol from the paper by Joachim Parrow [Par87]. This CCS specification is a finite-control system and has a scalable structure meaning that we may increase the net size by adding stations to this specification using a special purpose utility capable of generating an n-station protocol.

For a comparison of simulation performance (measured in time) between widely different tools to be fair, we need a way to make sure they simulate the same problem size. This was the case for the comparison between GPSim and Spin. Spin uses its own high level language, which technically is not a process algebra and we could not, with great accuracy, map the internals of Spin simulation to Petri net terminology (for instance, the concept of transitions). We needed a solution to stop the simulation once a criteria was met, specifying this criteria in at the event level allowed us to curb the simulation equally for both tools, regardless of how they simulate internally. The criteria was specified in the input specification, CCS_k for GPSim and PROMELA for Spin, as a firing counter and an assert respectively, thus the simulation was stopped once a number of successful *sends* (from any one station to another) had been correctly *received*.

6.4 A CCS CSMA/CD Protocol

The original protocol for two stations was a broadcast protocol where the stations should provide error free bidirectional communication to the upper layers of the protocol stack. The communications medium acts as a semaphore which the stations would try to acquire exclusive access to by sending their message. A collision could occur if exclusive access wasn't granted; the stations would synchronize back to a previous state where the choice was between resending the message or receiving a message. The specification of the original protocol is shown below in the syntax of GPSim and the original protocol from [Par87] is shown in the appendix, section 9.1.

```
constants P,MAC1,MAC1X,MAC2,MAC2X,M,MX;
```

```
actions s1,s2,r1,r2,e1,e2,b1,b2,br1,br2,er1,er2,c1,c2;
P = (^er1)(^c1)(^b2)(^br2)(^e2)(^er2)
    (^c2)(^e1)(^br1)(^b1)(MAC1|M|MAC2)
MAC1 = ['s1.MAC1X+br1.[er1.[r1.MAC1+'s1.r1.MAC1X]+'s1.er1.r1.MAC1X]]
MAC2 = ['s2.MAC2X+br2.[er2.[r2.MAC2+'s2.r2.MAC2X]+'s2.er2.r2.MAC2X]]
MAC1X = ['b1.[c1.MAC1X+'e1.MAC1]+br1.er1.r1.MAC1X]
MAC2X = ['b2.[c2.MAC2X+'e2.MAC2]+br2.er2.r2.MAC2X]
M = [b1.[b2.MX+'br2.[b2.MX+e1.[b2.MX+'er2.M]]] +
     b2.[b1.MX+ 'br1.[b1.MX+e2.[b1.MX+'er1.M]]]]
MX = ['c1.'c2.M+'c2.'c1.M]
```

6.4.1 Scaling the Protocol

A small utility was programmed that would reproduce the protocol specification for n stations instead of two. A similar utility to produce an n–station protocol for the PROMELA language was implemented, the minor changes needed for CWB–NC was made by hand. During this phase we diverted from broadcast semantics and instead introduced unicast semantics so that a sending station names the receiving station, but not vice versa. This of course changes the semantics of the protocol but for our comparisons it doesn't matter since we're only interested in the *performance* of the simulation and not any statistical information about the concurrent system.

The move from broadcast to unicast, in addition to going from two stations to n stations, required some modifications of the original protocol with the most significant changes made to collision handling. Previously, once a station had started to receive a message it would complete the receive before trying to send a pending message. This can be verified in the above specification by noting that a station, say `MAC1`, after synchronizing with the medium `M` on the channel `br1` the possible sequences of synchronizations are `er1.r1.MAC1`, `er1.'s1.r1.MAC1X`, or `'s1.er1.r1.MAC1X`. Thus the sender never check for complete delivery of a message once it's past a certain point (the only other station is busy receiving, a collision can't occur). No such assumption can be made in our unicast n-station protocol since a collision may now involve three stations; one sender, one receiver and a second sender (possibly to a fourth station). So modifications were made so that a sender will synchronize with the medium once a message has been completely received before the sender may proceed with another operation. It was also necessary for stations to check for collisions more often. A modified two station protocol is shown below and a parameterized version of a three station net is shown in the appendix, section 9.2.

```
constants P,M,MAC1,MACs12,MAC2,MACs21,Mc12;
actions s12,s21,er1,c1,br1,md1,r1,er2,c2,br2,md2,r2,b12,e12,b21,e21;
P = (^er1)(^c1)(^br1)(^md1)(^er2)(^c2)(^br2)(^md2)
    (^b12)(^e12)(^b21)(^e21)(MAC1|MAC2|M)
MAC1 = ['s12.MACs12+br1.[c1.MAC1+er1.[r1.MAC1+'s12.r1.MACs12] +
                         's12.[c1.MACs12+er1.r1.MACs12]]]
MAC2 = ['s21.MACs21+br2.[c2.MAC2+er2.[r2.MAC2+'s21.r2.MACs21] +
                         's21.[c2.MACs21+er2.r2.MACs21]]]

MACs12 = ['b12.[c1.MACs12+'e12.[c1.MACs12+md1.MAC1]] +
          br1.[c1.MACs12+er1.[c1.MACs12+r1.MACs12]]]
MACs21 = ['b21.[c2.MACs21+'e21.[c2.MACs21+md2.MAC2]] +
          br2.[c2.MACs21+er2.[c2.MACs21+r2.MACs21]]]
```

```
M = [b12.[b21.Mc12+'br2.[b21.Mc12+e12.[b21.Mc12+'er2.'md1.M]]]+
     b21.[b12.Mc12+'br1.[b12.Mc12+e21.[b12.Mc12+'er1.'md2.M]]]]
Mc12 = ['c1.'c2.M+'c2.'c1.M]
```

6.5 Case Study Comparisons

In section 5.2 we measured the performance of different optimization techniques for this specification and the measurements in figure 10 indicate that for this particular net we should turn on label categorization and select a hash map as our data structure for the transition sets.

6.5.1 CWB–NC Comparison

When measuring simulation performance for the CWB–NC tool we were unable to make the process automated via the script functionality. It seemed that entering the (interactive) simulation mode of the tool halted further reading of the script, so commands appearing in the script-file, directing the tool to simulate a given number of firings, were ignored and script-file processing resumed once simulation mode was exited. We were thus forced to manually type the command for simulating the intended number of firings and the command to exit simulation mode (which would be buffered in the shell), whereupon the tool would report estimated time of simulation. This manual delay was estimated to lie within the range of one to two seconds, so we increased the simulation size (in firings) in order to measure large values which would render the manual delay insignificant. We also performed the measurements three times in order to get an average, the number of firings simulated was 400000. It should also be noted that we used a compiled SML–NJ executable for these simulation runs, and the "caching" option was turned on.

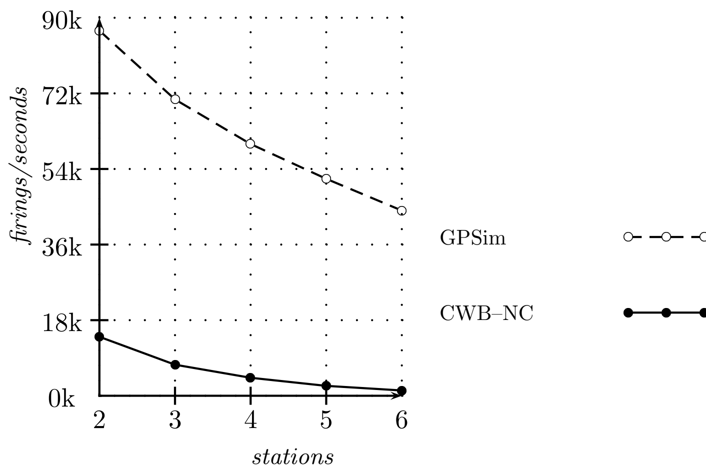

Figure 13: GPSim performance vs CWB–NC

In figure 13, we show the measured performance of CWB–NC in firings per second for each net size (number of stations) and the comparative measurements

for GPSim. One may derive two facts from the measurements. First, our GPSim clearly outperforms CWB–NC (which was to be expected). Second, our GPSim scales remarkably well indicated by halving its performance going from two stations to six. CWB–NC, in contrast, slows down about tenfold. This is also not very surprising. In retrospect the comparison is rather unfair because CWB–NC was never intended for optimized simulation and the inherent sluggishness of compiled SML code (though this may be under debate) but still gives an impression of the performance possible with GPSim.

6.5.2 Spin Comparison

The comparison with Spin use the PROMELA models generated by the utility program. The simulation proceeds until the given number of sends have been received, which is specified to be 2000 in the respective input files for GPSim and Spin. GPSim simulated with the same optimizations turned on as in the previous section, the hash map and label categorization. In figure 14 we can see the GPSim outperforming Spin steadily as the number of stations grow which both effect the net size and number of tokens. Spin version used for measurements was 4.2.1.

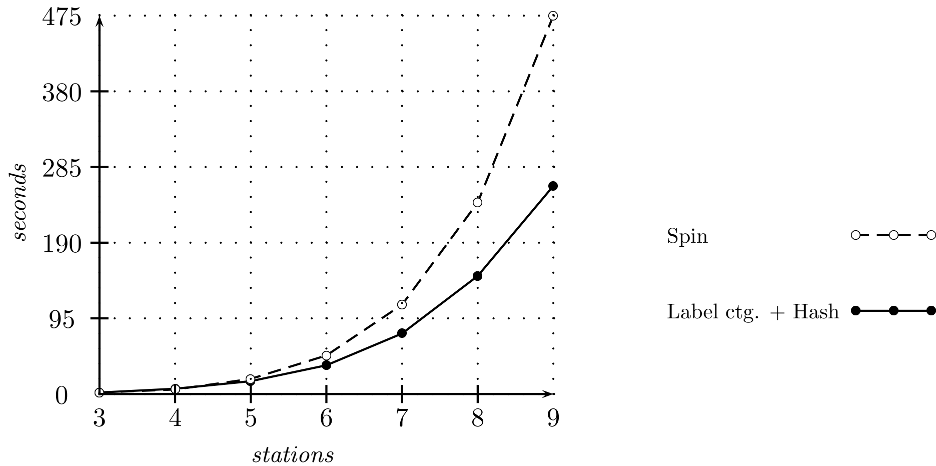

Figure 14: GPSim performance vs Spin

It should be mentioned that a measurement for the experimental net in section 5.3 was performed for Spin as well. Spin has an internal limit to the number of concurrent threads, namely 255. Though we do not support this claim by an actual graph, we maintain that simulating this net with Spin results in performance going downhill very fast as the number of threads increase. Spin was outperformed by GPSim by an even greater degree for this net.

6.6 Optimization Summary

From the measurements in the previous sections, we may draw some conclusions about each technique.

Memory Pool. The memory pool is a crucial technique for nets that are dynamic in nature, i.e. where the simulation state grows over time. It also may effect other parts of the simulation due to a better spatial locality of the memory pool objects.

Cache. The cache effects nets with synchronization transitions based on infinitely restricted labels and its speedup will increase as token size grows. It benefits greatly from the memory pool optimization.

Array vs Hash map. This is a simple choice between array and hash map as the data structure for the transition set. The hash map shouldn't be used for small nets and conversely when the number of transitions reach a certain threshold it's imperative to use a data structure with fast lookup operations.

Label categorization. Label categorization takes advantage of the CCS_k specification form and in return the net structure. It uses this domain knowledge in order to reduce computation for synchronization transitions when the firing policy requirements are fulfilled by all potential enabling candidates. It also reduces the total number of transitions in the net because it identifies transitions which would never fire in any simulation state and promptly removes them. This optimization doesn't attack the problem with quadratic search space for synchronization transitions per se, but it identifies when that search is unnecessary for individual transitions thus not only reducing computation, but also memory consumption by several magnitudes and this last fact might make it the most important optimization implemented.

On the negative side, of these four optimization techniques not one of them gives an appreciable speedup for all types of nets. The exception to this rule should be the memory pool, however its simplistic design and its use for only dynamic objects of the simulation results in the observed behaviour.

6.7 Future Optimizations

GPSim is a generic simulator for GP-Nets and use generic domain knowledge assumptions about GP-Nets to optimize simulation, very little input-specific information is used because of the shallow net analysis. Because the current implementation couples parsing, net creation, and simulation there are input-specific quantities that are difficult to realize into optimizations. A future implementation could take a non-generic approach, where the front-end would parse the net and analyze it thoroughly. The output would be a net description and representation in the form of source code. This output would be compiled and linked with a simulation library, and the compiled program would be a simulation tool for the specific net. Such an approach could make a lot of choices regarding internal data structures of *individual* entities in the net by generating the appropriate net representation.

One such net quantity which may benefit from such an approach is the token bound at individual places. An arbitrarily large model would most likely have subnets where only a few (or just one) token circulate, and other subnets where no such bound exist. The current generic implementation of places doesn't

take such characteristics into account; places which will, at most, have only one token still use a sorted vector and associated operations. The opposite situation, where places with no token bound and a high likelihood of high token count, also use a sorted vector which may not be the optimal choice. Other input-specific properties might be taken advantage of using this approach. Of course, this approach requires compilation for each different net which can be regarded as time consuming. However, it is speculated that the process of analysing the concurrent system will be comprised of many simulation runs, significantly more time consuming than the one time compilation.

7 GPSim Architecture

GPSim architecture is simple since the basic application is also rather simple. The complexity arise from the optimizations made to the algorithm because they use domain knowledge which isn't obvious from the problem specification. GPSim is comprised of a parser and a net representation. The main application file knits the functionality of these together creating GPSim.

7.1 Implementation Language and Methodology

Many other simulation tools are written in high-level languages such as SML which trade performance for applicability and versatility, however C++ was chosen as implementation language for several reasons. Foremost because of the performance of compiled native code, also because of a large library of utility components in the Standard Template Library which also has known asymptotic bounds. It was also the authors preferred language in contrast to only rudimentary knowledge of SML, and wouldn't incur a learning penalty from the start.

GPSim consists of a parser and a representation of the net generated by the parser. These two are tied together by the main application file which use the interface to the net for simulation. The net representation has a set of obvious but few components which resulted in a small number of base classes where we chose to forego the usual object-oriented source code candy (such as inheritance, polymorphism and overloaded operators) as much as possible. For performance reasons no error checking routines exist in the simulation code.

7.2 Net Representation

The basic types in the net representation are the basic entities in a Petri net. We have `Place` objects, `Token` objects, and `Transition` objects from which a number of specialized objects are derived. The net simulated is represented by a `Net` object which keeps track of the previously mentioned objects and also provide an interface for simulation.

7.2.1 The `Place` Object

A `Place` object is a container for outgoing `Transition` objects and `Token` objects. A `Place` object takes care of notifying neighbouring `Transition` objects that a `Transition` object has been fired. This is one application of the locality

principle and is an integral part of the enabling calculation process once a transition has been fired. `Transition` objects are notified differently whether any of their preplaces have had a token removed or added. A `Place` object which represents a zero constant in the CCS_k specification also deletes any `Token` objects added to it.

7.2.2 The `Token` Object

The `Token` object is also a container which encapsulates the firing history of a token in the form of a list. Each `Token` object has a position in the GP-Net indicated by a `Place` object member variable. `Token` objects also maintain an individual cache for fast lookup of $maxpref()$ information.

7.2.3 The `Transition` Object

A `Transition` object represents, in combination with the `Place` objects, the linkage of the net simulated. Each `Transition` object maintains a `Place` object variable for each of its preplaces and postplaces. During simulation, `Transition` objects are fired and modifies the marking of the net by removing and adding tokens from the preplaces to the postplaces respectively, along with any modifications to their firing history. Subsequently, tasks are delegated through the preplaces and postplaces to neighbouring `Transition` objects so they may recalculate their enabling status and update their (if any) record keeping data structures for enabling candidates. This process may move `Transition` objects in between the previously mentioned sets in section 4.2.1 for enabled and disabled transitions.

7.2.4 The `Net` Object

The `Net` object, of which only one exists, represents the GP-Net and contains all the entities of previous type which comprise the net and the simulation state. The `Net` object provides a function for single stepping the simulation one round which amounts to one firing, thus allowing a loop to move the simulation state forward in time.

8 GPSim Usage

A normal use-case of GPSim is to write a CCS_k specification in a text editor and save it as a file with an arbitrary name (a filename extension is not mandatory). This file is given as input to GPSim at the command prompt of a shell with appropriate command line options which defines both actions and feedback of GPSim. GPSim will issue error messages with line numbers if the input file isn't syntactically correct but no semantical analysis of the input is performed.

8.1 GPSim Options

GPSim accepts a few command line options with a dash (`-`) prefix, the exception being the filename of the input file which may appear anywhere on the command line. To display the command line options, GPSim may be executed with a question mark appropriately prefixed by a dash, and a brief description of all

options should be displayed along with any compile-time options turned on. The following table explains the available options.

Option	Description
-?	Show command line syntax
-p	Parse the input file only and show some net information
-i	Interactive mode, enables the debug prompt during simulation
-f<limit>	Set max number of simulation rounds (firings)
-s<limit>	Override stop criteria limit specified in the input file
-r<seed>	Set the seed to the random generator used for simulation
-d<flag>	Set one debug flag, either in hexadecimal or string form
-D<flags>	Set one or more debug flags in hexadecimal form

The feedback of GPSim is determined by the debug flags which are mostly used in conjunction with the interactive mode. Debug flags may be set individually as either a hexadecimal integer prefixed with 0x or equivalently as a string. The following strings may be used with the "d" flag.

ACTION	Show prefix ($\mathcal{T}_x$) transitions firing
COMPLEMENT	Show prefix ($\mathcal{T}_{\bar{x}}$) transitions firing
SYNC	Show synchronization transitions firing
RESTRICTION	Show restriction transitions firing
COMPOSITION	Show composition transitions firing
EMPTY	Show empty transitions firing
SELECTION	Show firing selection
ENABLED	Show enabled transitions
TOKENS	Show token positions at beginning of each round
STATS	Display simulation statistics and information

Setting the STATS flag makes GPSim display, in addition to other information, both the debug flags and random seed used for a simulation so the user may issue the same debug flags with the -D option and repeat a simulation verbatim. The interactive mode is intended for debugging purposes but may be used for single stepping through a simulation. Typing h and pressing Enter will give a brief help on commands. GPSim will stop the simulation once any of the given stop criteria have been fulfilled.

8.2 Input Language

The input language is comprised of three sections. First comes a mandatory declarative section for the agent constants and names of in the model. Second is an optional declarative section where each declared name is attributed to be a member of the set of unrestricted labels or the set of globally restricted labels. Any name not declared in this way is by default attributed to be infinitely restricted. Last comes the agent constant definitions. For instance, the example net in figure 1 would have a specification such as

```
constants A, 0;
actions x;
A = [x.0]|((^x)(['x.0]|A))
```

where the two first statements are the mandatory declarative statements, the optional section isn't necessary (`x` is infinitely restricted) and then the agent constant definition of `A` which is deemed the starting place. The input language of GPSim adhere to the following grammar.

```
       ccsk  :  decls defs
      decls  :  cdecl adecl optional
      cdecl  :  constants strlist
      adecl  :  actions strlist
   optional  :  odecl optional ‖ ϵ
      odecl  :  fdecl ‖ rdecl ‖ sdecl
      fdecl  :  unrestricted strlist
      rdecl  :  restricted strlist
      sdecl  :  stop strlist = integer
    strlist  :  string ; ‖ string , strlist
       defs  :  defs agentconst ‖ ϵ
 agentconst  :  agentid = agent
      agent  :  ( agent ) ‖ agent | agent ‖ summation ‖
                agentid ‖ (^ nameid ) agent
  summation  :  [ stlist ]
     stlist  :  stlist + sterm ‖ sterm
      sterm  :  nameseq summation ‖ nameseq agentid
    nameseq  :  name . nameseq ‖ ϵ
       name  :  nameid ‖ ' nameid ‖ ∼
```

Keywords are shown in `teletype` and grammar reduction rules in *italic*. Punctuation and separation symbols are syntactically important. The ‖ symbol separates different reduction choices for a rule and ϵ denotes an empty reduction choice.

The keywords `restricted` and `unrestricted` means that labels created from the given names are defined as globally restricted or unrestricted respectively, other labels are, by default, infinitely restricted. If label categorization is not turned on, these optional statements have no effect.

The `stop` statement tells the simulation to count the number of times a transition based on a label created from any of the given names is fired, and stop the simulation once the counter reaches the limit (denoted by *integer*).

The move to GP-Nets from p-nets required a structural change in both the grammar and the parser code to handle the new summations when creating the GP–Net. A design flaw was inadvertently introduced into the grammar, mostly due to a lack of structurally different test cases when implementing the change in the parser (and probably lack of sleep as well). This flaw makes the parser reject agent constants with a composition operator inside a summation. This means that terms such as "`[x.(X|Y) + z.Z]`" is illegal on the right–hand side of an agent constant definition. It should be stressed that this flaw does not reduce the value of simulations performed by GPSim, nor does it decrease the expressiveness of the language. One may avoid the problem by introducing a new agent constant used inside the summation and whose definition is the composition. Because of this work–around, which produce a semantically equivalent specification, it was given a rather low priority on the to-do list and has survived to this date.

References

[Bal03] *Baldamus, M.*, *Mayr, R.* and *Schneider, G.*. A backward/forward strategy for verifying safety properties of infinite-state systems. Technical Report ISSN 1404-3203, Uppsala University, Department of Information Technology, Uppsala, Sweden [dec 2003].

[Bal04] *Baldamus, M.*, *Mayr, R.* and *Schneider, G.*. Finite-control petri net semantics for infinite-control processes. Technical report, Uppsala University, Department of Information Technology, Uppsala, Sweden [2004].

[Cao90] *Cao, X.-R.* and *Ho, Y.-C.*. Models of discrete event dynamic systems. *IEEE Control Syst. Mag.*, 10(4):69–76 [June 1990]. NewsletterInfo: 38.

[Chi93] *Chiola, G.* and *Ferscha, A.*. Distributed simulation of timed petri nets: Exploiting the net structure to obtain efficiency. In *Marsan, M., A.*, ed., *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA*, volume 691, pp. 146–165. Springer-Verlag [1993].

[Cho92] *Chou, C.-C.*. *Parallel simulation and its performance evaluation.*. Doctoral Thesis, University of Iowa [January 1992]. InternalNote: Submitted by: hr.

[Dij72] *Dijkstra, E.*, *Dahl, O.* and *Hoare, C.*. *Structured Programming*. Academic Press [1972].

[Eva93] *Evans, J. B.*. The devnet: a petri net for discrete event simulation. *Lecture Notes in Computer Science; Advances in Petri Nets 1993*, 674:91–125 [1993].

[Gil98] *Gile, M. R.* and *DiCesare, F.*. A portable framework for distributed simulation of colored stochastic petri nets. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'98), 11-14 October 1998, San Diego, CA*, pp. 61–65 [1998].

[Haa89] *Haas, P. J.* and *Shedler, G. S.*. Stochastic petri net representation of discrete event simulations. *IEEE, Transaction on Software Engineering*, 15(4):381–393 [1989]. NewsletterInfo: 33,35 InternalNote: auch +f: 35.585.

[Hal90] *Hall, A.*. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19 [1990]. ISSN 0740-7459.

[Mil89] *Milner, R.*. *Communication and Concurrency*. Prentice Hall International (UK) Ltd [1989].

[Nke94] *Nketsa, A.*. Conservative distributed simulation of petri nets. *Computer and Mathematics with Applications*, 27(9-10):45–51 [1994].

[Par87] *Parrow, J. G.*. Verifying a csma/cd protocol with ccs. Technical Report ECS-LFCS-87-18, Computer Science dept., University of Edinburgh [1987].

[Pet62] *Petri, C. A.*. *Kommunikation mit Automaten.*. Bonn: Institut fr Instrumentelle Mathematik, Schriften des IIM Nr. 2 [1962].

[Rei98] *Reisig, W.* and *Rozenberg, G.*. Informal introduction to petri nets. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491 [1998].

[Tur36] *Turing, A.*. On computable numbers, with an application to the entscheidungs problem. In *Proceedings of the London Mathematical Society, Series 2, 42*, pp. 230–265 [1936].

[vH89] *van Hee, K. M.*, *Somers, L. J.* and *Voorhoeve, M.*. A formal framework for simulation of discrete event systems. In *Murray-Smith, D. et al.*, eds., *ESC 89. Proceedings of the 3rd European Simulation Congress, 1989, Edinburgh, UK*, pp. 113–116. SCS Eur., Ghent, Belgium [1989]. NewsletterInfo: 35.

[Zei00] *Zeigler, B. P.*. *Theory of Modeling and Simulation*. 2nd Edition, Academic Press [2000]. ISBN 3-540-62752-9.

9 Appendix

9.1 Original Two Station MAC Protocol

The original MAC protocol is shown in figure 15 with the indices of the MAC stations suppressed and where "?" denotes a input action and "!" denotes an output action.

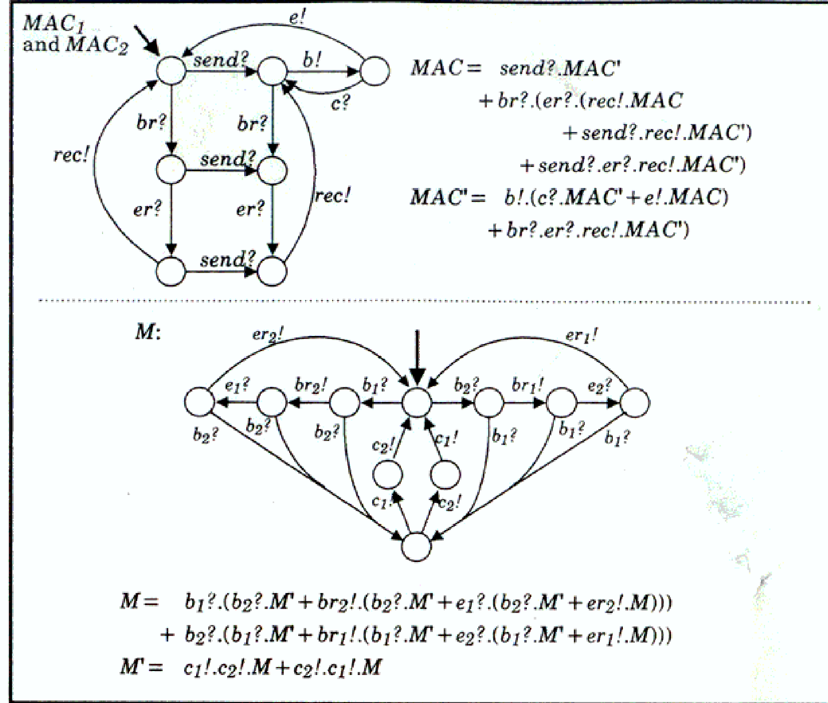

Figure 15: Original MAC sublayer protocol figure borrowed from [Par87]. Permission granted by Joachim Parrow.

9.2 Scaled Three Station MAC Protocol

Below is a parameterized version for a three station model. The initial and sending states for station `MAC1` is shown in full. Other stations are similar, the only difference being in dices. The declarative sections are not shown.

```
% Start of 3-station protocol
P = (^Eur1)(^c1)(^br1)(^md1)(^er2)(^c2)(^br2)(^md2)(^er3)
    (^c3)(^br3)(^md3)(^b12)(^e12)(^b13)(^e13)(^b23)(^e23)
    (^b21)(^e21)(^b31)(^e31)(^b32)(^e32)(MAC1|MAC2|MAC3|M)

% Initial state for MAC<1>
MAC1 = ['s12.MACs12 + 's13.MACs13 + br1.[c1.MAC1 +
        er1.[r1.MAC1 + 's12.r1.MACs12 + 's13.r1.MACs13] +
        's12.[c1.MACs12 + er1.r1.MACs12] + 's13.[c1.MACs13 +
```

```
        er1.r1.MACs13]]]
% Sending states for MACs<1,2>. 1 sending to 2.
MACs12 = ['b12.[c1.MACs12 + 'e12.[c1.MACs12 + md1.MAC1]] +
           br1.[c1.MACs12 + er1.[c1.MACs12 + r1.MACs12]]]
% Sending states for MACs<1,3>. 1 sending to 3.
MACs13 = ['b13.[c1.MACs13 + 'e13.[c1.MACs13 + md1.MAC1]] +
           br1.[c1.MACs13 + er1.[c1.MACs13 + r1.MACs13]]]

% More MAC<n> where n != 1, similar different indices
MAC<n> = [ ... ]
% More MACs<n,m> where n != 1 and m != n, similar different indices
MACs<n,m> = [ ... ]

% Medium
% Successful send is: b12.'br2.e12.'er2.md1.M
% Showing handling when station 1 sends to 2
M = [b12.[b23.Mc12 + b21.Mc12 + b31.Mc13 + b32.Mc13 +
    'br2.[b31.Mc123 + b32.Mc123 +
     e12.[b31.Mc123 + b32.Mc123 +
    'er2.'md1.M]]] +
% station 1 sends to 3, similar
     b13.[ ... [ ... [ ... ]]] +
% station 2 sends to 1, similar
     b21.[ ... [ ... [ ... ]]] +
% and so forth...
     ... ]
Mc12 = ['c1.'c2.M + 'c2.'c1.M]
Mc13 = ['c1.'c3.M + 'c3.'c1.M]
Mc23 = ['c2.'c3.M + 'c3.'c2.M]
Mc123 = ['c1.'c2.'c3.M + 'c1.'c3.'c2.M + 'c3.'c1.'c2.M +
         'c3.'c2.'c1.M + 'c2.'c1.'c3.M + 'c2.'c3.'c1.M]
```

9.3 Adequacy Theorem for P–Nets

The following paragraph, item 1 and 2, and Theorem 3, is taken from the paper in [Bal03] on page 8 and is restated here for completeness of the definition of p–nets.

> *"The intuition behind p–net firings is to represent what could perhaps be called a kind of sub–atomic behavior, that is, behavior one level below the operational transition semantics. Specifically, net markings correspond to sub–atomic states and net firings to sub–atomic steps, where there are branches due to summation, forks due to parallel composition, backward jumps due to agent constants and so on. The behavior can be regarded as sub–atomic since in general several firings of the eventual p–net are required to represent one SOS transition. What is crucial for us is that p–net firings coincide with the standard transition semantics in the sense that both generate the same observable traces up to firings labeled with ϵ. Theorem 3 states that formally. As a preliminary, given any agent P, let $init_P$ be the*

marking that maps (i) P to the (singleton) multiset containing the empty token and (ii) all other places/agents to the empty multiset. Moreover:"

1. The set $TTrace(P)$ of (possibly infinite) *transition traces* of P is given by

   $$TTrace(P) = \{\alpha_1\alpha_2 \ldots \mid \text{there are } P_0, P_1, \ldots \text{ so that } P_0 = P \text{ and } P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots\}.$$

2. The set $FTrace(P)$ of (possibly infinite) *firing traces* of P is given by

   $$FTrace(P) = \{\lambda_1\lambda_2 \ldots \mid \text{there are } m_0, m_1, \ldots \text{ so that } m_0 = init_P \text{ and } m_0 \xrightarrow{\lambda_1} m_1 \xrightarrow{\lambda_2} \ldots\}.$$

Theorem 3. For any agent P, the sets $TTrace(P)$ and $FTrace(P)$ coincide up-to the removal of all τ's from the traces in $TTrace(P)$ and of all τ's and all ϵ's from the traces in $FTrace(P)$.

Recent technical reports from the Department of Information Technology

2005-007 Linda Brus: *Nonlinear Identification of an Anaerobic Digestion Process*

2005-008 Linda Brus: *Nonlinear Identification of a Solar Heating System*

2005-009 Claes Olsson: *Disturbance Observer-Based Automotive Engine Vibration Isolation Dealing with Non-Linear Dynamics and Transient Excitation*

2005-010 Pär Samuelsson, Björn Halvarsson, and Bengt Carlsson: *Cost-Efficient Operation of a Denitrifying Activated Sludge Process - An Initial Study*

2005-011 Per Carlsson and Arne Andersson: *A Flexible Model for Tree-Structured Multi-Commodity Markets*

2005-012 Milena Ivanova and Tore Risch: *Customizable Parallel Execution of Scientific Stream Queries*

2005-013 Håkan Zeffer, Zoran Radovic, and Erik Hagersten: *Flexibility Implies Performance*

2005-014 Richard Gold and Mats Uddenfeldt: *Daigan: Constructing Proxy Networks with SelNet*

2005-015 Håkan Zeffer, Zoran Radovic, Martin Karlsson, and Erik Hagersten: *TMA: A Trap-Based Memory Architecture*

2005-016 Håkan Zeffer and Erik Hagersten: *Adaptive Coherence Batching for Trap-Based Memory Architectures*

2005-017 Tobias Bandh: *Evaluation of Authentication Algorithms for Small Devices*

2005-018 Eva Olsson, Niklas Johansson, Jan Gulliksen, and Bengt Sandblad: *A Participatory Process Supporting Design of Future Work*

2005-019 Torsten Söderström: *Computing the Covariance Matrix for PEM Estimates and the Cramer-Rao Lower Bound for Linear State Space Models*

2005-020 Stefan Engblom: *Computing the Moments of High Dimensional Solutions of the Master Equation*

2005-021 Mei Hong, Torsten Söderström, and Wei Xing Zheng: *Accuracy Analysis of Bias-Eliminating Least Squares Estimates for Identification of Errors-in-Variables Systems*

2005-022 Torbjörn Wigren: *MATLAB Software for Recursive Identification and Scaling Using a Structured Nonlinear Black-box Model – Revision 2*

2005-023 Per Lötstedt and Lars Ferm: *Dimensional Reduction of the Fokker-Planck Equation for Stochastic Chemical Reactions*

2005-024 Torsten Söderström, Erik K. Larsson, Kaushik Mahata, and Magnus Mossberg: *Approaches for Continuous-Time Modeling in Errors-in-Variables Identification*

2005-025 Paul Pettersson and Wang Yi (eds.): *Pre-Proceedings of the 3rd International Conference on Formal Modelling and Analysis of Timed Systems*

2005-026 Erik Abenius, Fredrik Edelvik, and Christer Johansson: *Waveguide Truncation Using UPML in the Finite-Element Time-Domain Method*

2005-027 Timo Qvist: *Fast Simulation of Concurrent Agents with PNets - the GPSim Tool*

UPPSALA
UNIVERSITET

September 2005
ISSN 1404-3203
http://www.it.uu.se/