HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports ISSN 0783–5396
No. **29**; March 1994 ISBN 951-22-2028-8

THE MODELLING AND ANALYSIS OF A HIGH SPEED DATA BUS WITH PREDICATE/TRANSITION NETS

Tino Pyssysalo

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, Finland

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
SF–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **29**; March 1994

ISSN 0783–5396
ISBN 951-22-2028-8

The Modelling and Analysis of a High Speed Data Bus with Predicate/Transition Nets

TINO PYSSYSALO

Abstract: In this work predicate/transition nets (Pr/T-nets) are used in the modelling and analysis of the medium access control algorithm of a parallel high speed data bus, Frame Synchronized Ring (FSR-bus). Two Pr/T-net models of the bus are presented. One models the basic operation of the control algorithm. The other is an extended model, in which nodes can send longer frames and the reception of a frame can fail.

The analysis of the nets is based on the reachability analysis, which is made by a highly advanced Pr/T-net reachability analysis tool, `PROD`. The state space of the reachability graph grows very fast with respect to the number of nodes in the ring. The state space explosion is tried to be reduced with the stubborn set and symmetry methods. The stubborn set method reduces the number of reachable states almost 70%. Symmetry method is not so efficient. These results show, how effective these methods can be.

Analytical models and simulations have been used in the analysis of the FSR-bus. With these methods it has not been possible to prove, whether the bus can deadlock or whether the bus is fair. In this work these properties are successfully proved with `PROD`.

Other important results concern the performance of the FSR-bus in terms of waiting times of the nodes. The longest waiting path and the situation, in which the waiting occurs, are examined with `PROD`. It is found that in some circumstances a node has to wait quite long to get access to the bus, which is undesirable in real time applications.

Keywords: Concurrent and distributed systems, protocol verification, predicate/transition nets, reachability analysis tools, Frame Synchronized Ring.

Printing: TKK Monistamo; Otaniemi 1994

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
SF–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358-0}$ 4511
Telex: 125 161 htkk sf
Telefax: +358–0–465 077
E-mail: lab@hutds.hut.fi

Contents

1 Introduction

The need of computation power is continuously increasing. In large database systems, multimedia, and real time applications the increasing computation demands are achieved by distributing computation tasks among several communicating processors. An example of such a *distributed system* is a telecommunication protocol, where each individual node of a network obeys some communication rules, so that the whole network of processors performs the computation task together.

It is very difficult to verify the operation of distributed systems. Such systems rarely have only one execution path, but there are many possible operation sequences, which the system can execute. For example it is impossible to determine the order of actions of individual processors. Distributed systems can be modelled with formal abstract models, with which the essential behaviour of a system can be more easily verified.

Reachability analysis is an efficient method of verification for distributed systems. In the reachability analysis a *reachability graph* of an abstract model of the system is generated. The graph consists of all the reachable states of the system and actions leading from one state to another. Using the reachability analysis it is easy to check, if certain properties hold in the system or find for example every terminal state of a system. Unfortunately, reachability graphs of many practical systems are intractably large (*state space explosion*). The number of reachable states grows very fast with respect to some system parameter, for example a number of nodes in a communication network. Many methods have been proposed, with which a partial reachability graph can be used in the verification of a system.

One example of a distributed system is the *Frame Synchronized Ring* (*FSR-bus*) [3, 4, 6, 7]. The FSR-bus is a high throughput data bus for high speed multiprocessor applications, which include high speed packet switching, real time video compression, multimedia coding, and processing of security algorithms. The FSR-bus has been developed in Telecommunications Laboratory of Technical Research Centre of Finland since 1990. The main goal of the research program is to develop a cost-effective and efficient interconnection network that could be used in many high speed applications.

In this work the *medium access control algorithm* (*MAC algorithm*) of the FSR-bus is modelled and analysed with *predicate/transition nets* (*Pr/T-nets*). The validation and performance analysis of the MAC protocol has been studied with analytical models and simulations. However, the complexity of the protocol makes analytical approaches inaccurate. With simulation the correctness, fairness, and performance of the FSR-bus can be examined in a natural way. The choice between different operation sequences is typically random, so the errors found with simulation may be hard to locate and repeat. Furthermore, there is a possibility that certain types of errors are not found at all due to randomness. With simulations and analytical models it has not been possible to prove that the FSR-bus does not deadlock or that the MAC algorithm is fair. Pr/T-nets offer an additional way to study the correctness and fairness of the FSR-bus. The usefulness of Pr/T-nets lies in the ease of human comprehension of graphic description and sophisticated theory behind the nets.

A highly advanced reachability analysis tool, `PROD` [2, 14], is used in generating the reachability graphs of predicate/transition nets. `PROD` has been developed by a group of researchers of Digital Systems Laboratory and students of Helsinki University of Technology. Many reduction methods to reduce the size of the state space have been implemented in `PROD`, so reachability graphs of large systems can be generated easily and fast with it. `PROD` includes a graph query program `probe`, with which queries about the properties of the system can be made. With `probe` it is possible to search for nodes or event sequences satisfying some given property.

The goals of this work include the following.

- An abstract Pr/T-net model of the FSR-bus is developed. With reachability analysis and `PROD` terminal nodes of the system are searched for. It is proved that the FSR-bus cannot deadlock. The number of nodes in the FSR-bus is initially two and is increased, until the reachability graph contains so many states that the disk space (290 Mbytes) runs out.

- Because the size of the reachability graph grows very fast with respect to the number of nodes in the bus, some methods are used to reduce the size of the reachability graph. These methods include *stubborn set method*, *symmetry method* and bit packing. Using these methods the size of the reachability graph is tried to be reduced, so that the number of nodes in the bus could be increased in the analysis. It is examined, how effectively these methods work in practice, i.e., how much space and time they need to generate the reduced reachability graphs.

- It is studied, whether the fairness of the FSR-bus can be proved with `probe` queries. If it is found possible, the queries are constructed and proofs are given. If the FSR-bus is fair, the nodes that are ready to send get always access to the bus after a finite waiting time.

- If the fairness of the FSR-bus can be proved, actual waiting paths of the nodes are examined. It is tried to find the situation, in which a node has the longest waiting time from the moment it gets ready to send to the moment it gets access to the bus. In addition, it is studied, how long a node has to wait in the worst case, before it can issue a service request, if the bus is occupied and the node is ready to send.

The correctness of the FSR-bus has not been proved yet. With simulations no error situations have been found, but there is still a possibility that the bus does not operate in a proper way. If the correctness of the FSR-bus, i.e., the bus is fair and does not deadlock, can be proved with `PROD`, the result will have a great significance. The bus is useless in most applications, to which it was originally developed, if the medium access control algorithm includes errors. The results concerning the waiting times of the nodes are important too. If long waiting times are usual in the bus that is used in real time applications, lots of data will be lost or at least delayed.

In Section 2 it is discussed, how concurrent and distributed systems can be modelled and analysed. The syntax, semantics, and some examples of the abstract model used in this work, Pr/T-nets, are given. Stubborn set method and symmetry method that are used to avoid the state space explosion are presented. A simple example of their usefulness is given.

The reachability analysis tool `PROD` and its features are presented in Section 3. It is explained, how to model and analyse Pr/T-nets with `PROD`. The basics of the `probe` graph query language are presented and examples of its usage are given.

Frame Synchronization Ring is presented in Section 4. The structure and operation of the bus are explained. The flow diagram of the medium access control algorithm of the bus is shown.

The Pr/T-net model of the FSR-bus is developed in Section 5. Two different models are developed. The first one is basically similar to the MAC protocol of the FSR-bus. The second model is an extension of a basic model. It includes credit size, some error situations, and retransmissions of packets.

The analysis methods and results are presented in Section 6.

A source listing of a C program that finds the longest path from a set of paths generated by `PROD` is seen in Appendix A. In Appendix B a `PROD` net description of the predicate/transition net model of the Frame Synchronized Ring is given. The net description of the extended model of the bus is given in Appendix C.

2 Modelling and Analysing Concurrent and Distributed Systems

Concurrent and distributed systems consist of many communicating processes or processors. Frame Synchronized Ring bus (FSR-bus) is one example of a distributed system, because the data flow control of the bus is distributed among the nodes connected to the bus. When the number of processing units increases, it becomes too complex to verify the operation of these systems without any abstract formal models and analysis tools. Several methods have been developed, with which it is easy to model concurrent and distributed systems and verify, if certain properties hold in abstract models of these systems. For example process algebras, net theory and specification languages are widely used in modelling and analysing distributed systems.

In this work predicate/transition nets (Pr/T-nets) are used in modelling and analysis of a medium access control algorithm of a distributed data bus. The usefulness of Pr/T-nets lies in the ease of human comprehension of graphic description and sophisticated theory behind the nets, which enables powerful simulation and interpretation methods.

2.1 Syntax of Predicate/Transition Nets

An introduction to the formalism of predicate/transition nets is now given. The following definitions are based on the definitions in [1] and [10].

A structure is a tuple of objects, $\mathcal{R} = (D; f_1, ..., f_k; R_1, ..., R_n)$, where D is a non-empty set of *individuals* called the *domain* or *carrier* of $\mathcal{R}$, the f_i are *functions* in D and the R_j are *relations* in D.

An example of a structure is the structure of whole numbers, which is denoted by $\mathcal{Z} = (\mathcal{Z}; +, *, ...; =, \neq, >, \geq, ...)$. Note that the domain of the structure is often denoted by the same letter as the structure. In this work no other structures than the structure of whole numbers are used.

Definition 2.1 *For each $n \geq 0$ a set of n-ary* operators *(function symbols) and a set of n-ary* predicates *(relation symbols) of the structure $\mathcal{R}$ form the* vocabulary *of the* first-order language *L that consists of two kinds of expressions,* terms *and* formulae. *A set of* well-formed *terms is denoted by L_t and a set of well-formed formulae by L_f. In addition, there is a set of variable symbols, V, disjoint from operators and predicates. Expressions are built in the following way.*

L_t :

- *A variable is a term.*

- *If $f^{(n)}$ is a n-ary operator and $v_1, ..., v_n$ are terms, then $f(v_1, ..., v_n)$ is a term.*
- *No other expression is a term.*

L_f :

- *if v_1 and v_2 are terms, then $v_1 = v_2$ is an* atomic *formula.*
- *If $P^{(n)}$ is a n-ary predicate and $v_1, ..., v_n$ are terms, then $Pv_1...v_n$ is an atomic formula.*
- *If p_1 and p_2 are formulae, then $\neg p_1$ and $(p_1 \vee p_2)$ are formulae.*
- *If x is a variable and p is a formula, then $(\exists x)p$ is a formula.*
- *No other expression is a formula.*

Definition 2.2 *If L is a first-order language, a class $LC^{(n)}$ of* symbolic sums *of tuples of length n for each $n \geq 0$ is defined in the following way.*

- *The constant 0 is in $LC^{(n)}$.*
- *If $v_1, ..., v_n$ are terms, then the n-tuple $\langle v_1, ..., v_n \rangle$ is in $LC^{(n)}$.*
- *If l_1, l_2 are in $LC^{(n)}$, then $(l_1 + l_2)$ is in $LC^{(n)}$.*
- *If l is in $LC^{(n)}$ and z is a non-negative integer, then zl is in $LC^{(n)}$.*
- *No other expression is in $LC^{(n)}$.*

A set of symbolic sums of an arbitrary set A is denoted by $LC_\Sigma(A)$. The set of all finite tuples of an arbitrary set A is denoted by $A^* = \bigcup_{n \geq 0} A^n$.

Now the definition of predicate/transition nets is given.

Definition 2.3 *Let L be a first-order language. A* predicate/transition net (Pr/T-net) *is a 3-tuple $N_L = \langle N, A, M_0 \rangle$, where*

- *N is an underlying* directed net, *$N = \langle S, T; F \rangle$, where S is a set of* places, *T is a set of* transitions *($S \cap T = \emptyset$), and $F \subseteq (S \times T) \cup (T \times S)$ is a* flow relation *from places to transitions and conversely.*
- *A is the* annotation *of N, $A = (A_N, A_S, A_T, A_F)$, where*
  - *1. $A_N = \mathcal{R}$ is a first-order structure for L, called the* support *of N.*

  - *2. A_S is a bijection between the set of places and the set of variable predicates that assigns to each place an* arity. *The arity is the size of the tuples in a place.*
  - *3. $A_T : T \to L_f$ assigns to each transition a well-formed formula. A formula is called a* firing condition *(*selector *or* gate of the transition*). All the variables that are outside the scope of quantifiers $\exists$ and $\forall$ must appear in some symbolic sum that A_F assigns to the arc that hits the transition.*
  - *4. A_F assigns to each arc a symbolic sum of tuples of terms of L_t. The arities of all the symbolic sums of tuples in an arc must equal to the arity of a place, which the arc hits.*

- *$M_0 : S \to LC_\Sigma(A_N^*)$ is the* initial marking *of the net N. It assigns to each place a symbolic sum of tuples of the carrier of the structure A_N. The size of the tuples must equal to the arity of a place.*

Graphically places are represented as circles, transitions as boxes and the flow relation by directed arcs between the respective circles and boxes. Tuples in the places determine the arity of a place. The first-order logical formulae of the selectors are written inside the boxes. Symbolic sums, for example $\langle a, b\rangle + \langle a, y\rangle$, are written inside the circles, when they represent markings or near the arcs, when they represent arc expressions.

An example of a Pr/T-net is seen in Figure 1. In Figure 1 a Pr/T-net model of an incorrect solution to the dining philosophers problem of three philosophers is shown. In this problem there are some philosophers sitting and thinking around a table. Eventually each of the philosophers will become hungry. In order to eat a philosopher needs two forks. Between every two philosophers there is a fork, so every philosopher can not eat simultaneously. Every philosopher has access only to the fork on his left hand side and the one on his right hand side. The problem is to find a protocol, which every philosopher obeys and according to which thinking and eating alternate smoothly. The problem is similar to that, where some processes use some scarce resources so that every process can not have access to the resources simultaneously.

The incorrect solution seen in Figure 1 goes as follows. Initially every philosopher is thinking. The symbolic sum of tuples of arity one, $\langle 0\rangle + \langle 1\rangle + \langle 2\rangle$, in places `thinking` and `forks` indicates that initially every philosopher is thinking and no forks have been taken. Because every philosopher is initially thinking, places `eating` and `withLeft` are empty.

When a philosopher gets hungry, he first takes a fork on his left hand side, unless another philosopher has taken it before him. The symbolic sum $\langle x\rangle$ near the arcs that hit the transition `takeLeft` indicates that a philosopher number x tries to get a fork number x. After the philosopher has successfully got the fork, he tries to get the fork on his right hand side (a fork number $(x+1)$ mod 3). When he has got two forks, he begins to eat. After eating, a philosopher puts two forks he has used back onto the table and begins to think again. The modulo operator is denoted by "|".

1-tuple $\langle x \rangle$ inside the circles of the net represents the action of the philosopher number x. If the philosopher x is eating, place `eating` contains 1-tuple $\langle x \rangle$ (and possible some other tuples). Note that empty symbolic sums in a marking and tautologies in a selector (for example $x = x$ in a transition `takeLeft` in Figure 1) are usually omitted in the graphical presentation of a Pr/T-net.

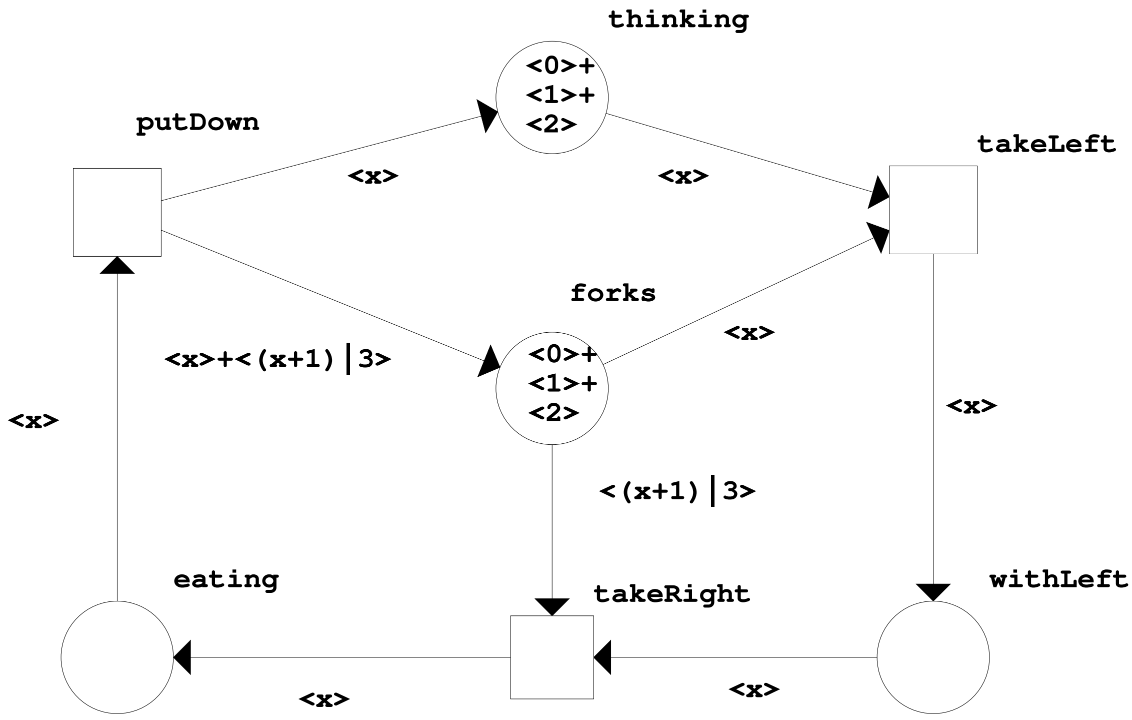

Figure 1: Pr/T-net of the dining philosophers problem of three philosophers.

2.2 Analysis of Predicate/Transition Nets

Reachability analysis is a powerful method of verification for concurrent and distributed systems. With reachability analysis it is easy to check, if a system has any terminal nodes or if certain properties hold in a system. Reachability analysis is based on generating the reachability graph for a formal model of a system (for example a Pr/T-net). The reachability graph consists of a set of vertices V and a set of edges E. Vertices are all the states reachable from the initial state of the system and the edges are the actions leading from one state to another. In this work the semantics for the Pr/T-nets is defined by a reachability graph with single transition steps. That means interleaving semantics is used.

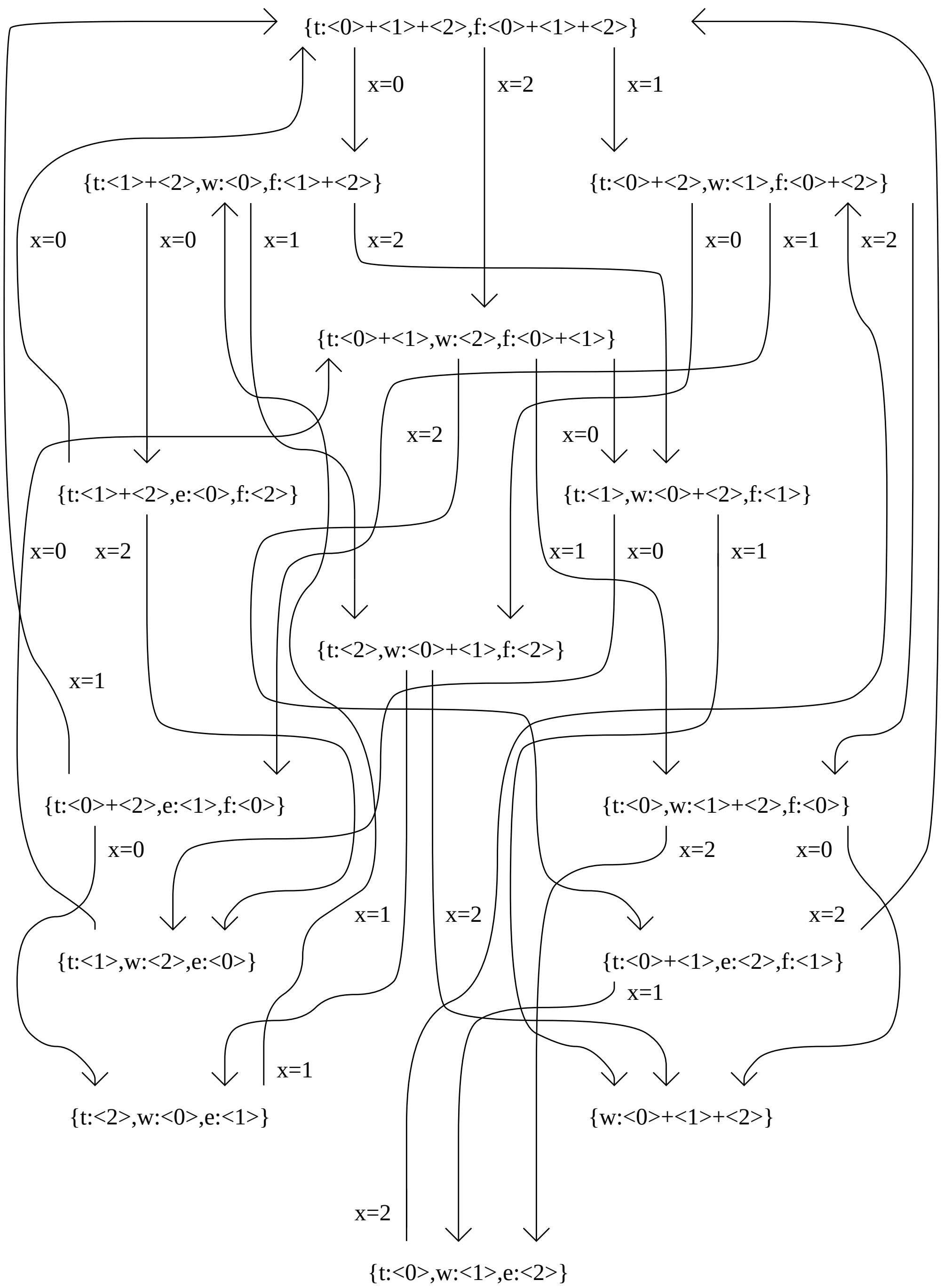

Figure 2: Reachability graph of the Pr/T-net of the dining philosophers problem.

The following definitions concerning the reachability graph of the Pr/T-nets are based on the definitions in [1] and [10].

If a formula E has variables outside the scope of quantifiers $\forall$ and $\exists$, it is called *open* and otherwise *closed* or a *proposition*. The occurrences of variables in a single term are always outside the scope of quantifiers and such a term is always open. Closed terms are *compound names* of individuals. Variables that are outside the scope of quantifiers are called *free* and can be replaced by terms in expressions.

Definition 2.4 *Let $\alpha = x_1 \leftarrow v_1, ..., x_n \leftarrow v_n$ be a* substitution, *where x_i are variables and v_i are terms. If E is an expression, $E : \alpha = E : \{x_1 \leftarrow v_1, ..., x_n \leftarrow v_n\}$ designates the result, when every free occurrence of x_i is substituted by v_i in an expression E. $E : \alpha$ is called the α-*instance *of E.*

Any closed expression E has an *interpretation* $\mathcal{R}(E)$ that is determined by the interpretation of its subexpressions: terms, operators, predicates and constants. The structure $\mathcal{R}$ assigns to each closed term an individual of $\mathcal{R}$ and to each closed formula the truthvalue (true or false).

For example in the structure $\mathcal{Z}$, the interpretation of an constant 2 is $\mathcal{Z}(2)$, which is 2 and the interpretation of an expression $(2+3=4)$ is $\mathcal{Z}(2+3=4)$, which is false.

Definition 2.5 *If every operator f^m and every predicate P^n of first-order language L denotes an m-ary function of the structure $\mathcal{R}$, designated by $f_{\mathcal{R}}$, and n-ary relation of the structure $\mathcal{R}$, designated by $P_{\mathcal{R}}$, respectly, a structure $\mathcal{R}$ is called a* structure *for L or a L-*structure.

Definition 2.6 *Let L be a first-order language, $\mathcal{R}$ a L-structure, and ϕ a formula of L. ϕ is said to be* valid *in $\mathcal{R}$, if and only if each closed instance of ϕ is true in $\mathcal{R}$. That is denoted by $\mathcal{R} \models \phi$.*

For example in the structure of whole numbers $\mathcal{Z} \models (2+3 \leq 8*6)$.

Definition 2.7 *If $\mathcal{R} \models (A_T(t)) : \alpha$ (an expression, which is the result, when all the free variables in the firing condition of the transition t are substituted by α, is true in the L-structure of $\mathcal{R}$) and the closed arc expression $A_F(s,t)$ and $A_F(t,s)$ have an interpretation, the α-instance of the transition t is called* feasible.

Let α be a substitution $x \leftarrow 1$. Then the α-instance of a transition `takeLeft` of the Pr/T-net in Figure 1 is feasible, because $\mathcal{Z} \models (x = x) : (x \leftarrow 1)$. Furthermore, all arc expressions of those arcs that hit the transition, have an interpretation, which is $\{\langle 1 \rangle\}$.

Definition 2.8 *Let N be a Pr/T-net, M and t its marking and its transition respectly and α a substitution of t. At marking M a feasible α-instance $t : \alpha$* is enabled, *denoted by $M[t : \alpha\rangle$, if*

$$\forall s \in S[\mathcal{R}(A_F(s,t):\alpha) \subseteq \mathcal{R}(M(s)) \wedge \mathcal{R}(A_F(t,s):\alpha) \cap \mathcal{R}(M(s)) = \emptyset]. \quad (1)$$

In Figure 1 a feasible α-instance $(x = x) : (x \leftarrow 1)$ of the transition `takeLeft` is enabled in the initial marking, because for every input place of the transition $\{\langle 1\rangle\} \subseteq \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle\}$ and for every output place of the transition $\{\langle 1\rangle\} \cap \emptyset = \emptyset$. For other places equation (1) holds, because an empty set is a subset of an empty set and intersection of an empty set and an arbitrary set is an empty set.

Definition 2.9 *Let N be a Pr/T-net, M and M' its markings, t its transition, and $t:\alpha$ a feasible α-instance of t. An α-instance $t:\alpha$ enabled at M can* fire *to marking M', denoted by $M[t:\alpha\rangle M'$, if*

$$\forall s \in S\ \mathcal{R}(M'(s)) = \mathcal{R}(M(s)) \setminus \mathcal{R}(A_F(s,t):\alpha) \cup \mathcal{R}(A_F(t,s):\alpha). \quad (2)$$

If the enabled α-instance $(x = x) : (x \leftarrow 1)$ of the transition `takeLeft` is fired in the initial marking of the net in Figure 1, a new marking is reached, at which $\langle 1\rangle$ is removed from places `thinking` and `forks` and added to the place `withLeft`.

Now a reachability graph of a Pr/T-net can be defined.

Definition 2.10 *The* reachability graph *of a Pr/T-net is a 2-tuple $\langle V, E\rangle$ where V is the set of* vertices *and E is the set of* edges*. The initial marking $M_0 \in V$. If $M \in V$ and $M[t:\alpha\rangle M'$, then $M' \in V$ and $\langle M, t:\alpha, M'\rangle \in E$. There are no other elements in V and E.*

The deadlock or the terminal node in the reachability graph is defined as follows.

Definition 2.11 *A marking M is* dead*, if $\forall t \in T$ none of the feasible α-instance $t:\alpha$ is enabled ($\neg M[t:\alpha\rangle$). The vertex representing the dead marking in the reachability graph is called a* terminal node *or a* deadlock*.*

In Figure 2 the reachability graph of the Pr/T-net in Figure 1 is seen. The graph has 14 vertices and 27 edges. Note the terminal node, where every philosopher has taken the fork from his left hand side. So the marking in the terminal node is $\{$`withLeft:` $\langle 0\rangle + \langle 1\rangle + \langle 2\rangle\}$. None of the transitions is enabled, because places `forks` and `eating` are empty.

2.3 State Space Explosion

The main problem with reachability analysis is state space explosion. The size of the state space may grow faster than exponentially with respect to the size of a system configuration. If the main purpose of the analysis is to search for terminal nodes, it is not necessary to generate the complete state space. Many methods have been proposed to avoid generating the complete state space. Two methods used in this work are symmetry method [5, 10] and Antti Valmari's stubborn set method [10, 12, 14]. These methods generate a reduced reachability graph of place/transition nets (P/T-nets), which are low-level Petri nets. Every Pr/T-net can be represented by an equivalent P/T-net. When the symmetry method or the stubborn set method are applied to Pr/T-nets, a Pr/T-net is unfolded into a P/T-net, either method is applied to a P/T-net, and the state space information of the reachability graph of the P/T-net is folded back into the Pr/T-net level. In Section 3 it is explained, how a Pr/T-net is unfolded into a P/T-net.

2.3.1 Stubborn Set Method

In the stubborn set method, the number of the states inspected during the generation of the state space is tried to be reduced, while still finding all terminal states. The definition of stubborness is given in [12]. Stubborn set is a subset of enabled transitions in a marking. Rather than firing all enabled transitions, only those transitions that belong to the stubborn set are fired. So if the stubborn sets include only a few of all enabled transitions, firing of the transitions of a set does not generate as many successor markings as firing of all enabled transitions. In that way the state space is tried to be reduced.

The idea behind the stubborn set method is the following. In a marking M the set of transitions T is tried to be partitioned into two sets T_s (stubborn set) and $\overline{T}_s$ (difference of the set of all transitions of the net and the stubborn set). In T_s there must be at least one enabled transition t and any path $\sigma \in (\overline{T}_s)^*$ must be enabled in the marking M. Sets should be constructed so that the transition t and the path σ commute in such a way that from the marking M it is possible to get to the marking M' by first firing the transition t and then all the transitions in the path σ or by first firing all the transitions in σ and then firing the transition t. If furthermore it can be guaranteed that firing any path $\sigma \in \overline{T}_s^*$ cannot enable any transition in the set T_s, there is no need to generate the internal markings on path from M to M' where all the transitions in the path σ are fired first and then the transition t is fired. If a terminal node can be found by generating such a path, it can be found by firing some enabled transition $t \in T_s$.

The efficiency of the stubborn set method depends, how effectively stubborn sets can be computed in markings of a net. Several algorithms to compute the set of enabled transitions of a stubborn set have been developed. Faster algorithms do not reduce the state space as much as slower algorithms, but they compute the stubborn sets more quickly. In this work the effective reduction of state space is preferred to the short computation time. That is why the so called *deletion algorithm* [13] is used. The deletion algorithm finds the minimal stubborn set, in the sense that no proper subset of its enabled transitions can be the set of all enabled transitions of any stubborn set, in time $\mathcal{O}(\mu\rho|T|^2)$, where T is the set of transitions ($|T|$ is the number of transitions in a set T), μ is the maximum number of input places of a transition and ρ is the maximum number of adjacent transitions of a place [13]. Because deletion algorithm finds the minimal stubborn set, the reachability graph should be small.

Stubborn set method does not always work efficiently. The reachability graph of the Pr/T-net in the Figure 1 generated by using deletion algorithm in computing the stubborn sets has 14 vertices and 21 edges. Only the number of edges could be reduced in this case.

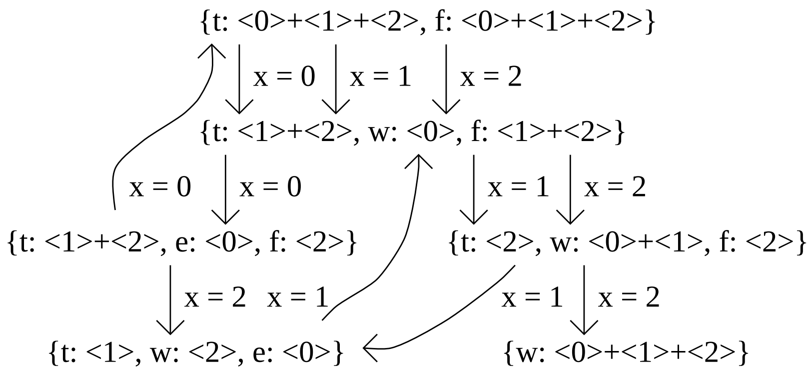

Figure 3: Reachability graph of the Pr/T in Figure 1 generated by using the symmetry method.

2.3.2 Symmetry Method

In symmetry method one marking in the reachability graph represents an equivalence class of markings. The equivalence classes are generated by an equivalence relation $\sim$ that is called a symmetry. The size of the reachability graph can often be significantly reduced, because many concurrent and distributed systems are fully symmetrical. However the time needed to generate the graph may not decrease at all, since the time saved in generating the smaller reachability graph is lost in computing the symmetries.

The reachability graph of the Pr/T-net of dining philosophers generated by using the symmetry method is seen in Figure 3. The number of vertices and edges is clearly smaller than in the complete state space reachability graph in Figure 2. In [10] five different types of symmetries among markings have been presented: identity, shift, permutation, related and product. In addition to the identity symmetry, two symmetries have been used in the graph of Figure 3. The first one is a shift, in which a philosopher and a fork number n are described by a philosopher and a fork number $(n+1)$ *mod* n and the other is a shift, where a philosopher and a fork number n are described by a philosopher and a fork number $(n+2)\ mod\ n$. For example all the markings $\{$`t:` $\langle 1\rangle + \langle 2\rangle$, `w:` $\langle 0\rangle$, `f:` $\langle 1\rangle + \langle 2\rangle\}$, $\{$`t:` $\langle 0\rangle + \langle 1\rangle$, `w:` $\langle 2\rangle$, `f:` $\langle 0\rangle + \langle 1\rangle\}$, $\{$`t:` $\langle 0\rangle + \langle 2\rangle$, `w:` $\langle 1\rangle$, `f:` $\langle 0\rangle + \langle 2\rangle\}$ belong to the same equivalence class, which the marking $\{$`t:` $\langle 1\rangle + \langle 2\rangle$, `w:` $\langle 0\rangle$, `f:` $\langle 1\rangle + \langle 2\rangle\}$ represents.

3 Reachability Analysis Tool PROD

The generation of the reachability graph of a Pr/T-net can be easily automated. Advanced automatic analysis tools generate a graph consisting of thousands of reachable markings in a few second. With good analysis tools it is possible to interpret the state space. Terminal nodes, event sequences that lead from the initial state to the terminal state, and paths satisfying a given property can be searched for.

3.1 General about PROD

PROD [14, 2] is a highly advanced reachability analysis tool for predicate/transition nets. It has been developed by a group of researchers of Digital Systems Laboratory and students of Helsinki University of Technology since 1989. PROD is continuously developed and new features are added to it. PROD has a graphical Pr/T-net editor, with which nets can be drawn easily. PROD automatically generates the reachability graph of the Pr/T-net drawn with the editor. Both symmetry method and stubborn set method are implemented in PROD, which makes it a powerful analysis tool. PROD has a rich query language for inspecting the reachability graph. With query language states or paths satisfying given properties can be found. PROD is one of the most advanced Pr/T-net analysis tools in the world.

3.2 Using PROD

PROD consists of a graphical Pr/T-net editor `pred`, net description language preprocessor `prpp`, a program computing the *strongly connected components* of the reachability graph called `strong`, a graph query program `probe`, and a batch program `prod`. PROD is the name of the whole analysis tool, `prod` is a part of it.

A predicate/transition net that is analysed is drawn with `pred`. An example net of a dining philosophers problem is seen in Figure 4. The Pr/T-net is compiled into the net description language of PROD, which is the C preprocessor language extended with net description directives. The net description of the Pr/T-net in Figure 4 is seen below. Places and their initial markings, transitions and their input and output flow, and firing conditions (gates) of the net can be found in the net description. Empty markings and tautologies in gates are not seen in the net description. The high and low limits of the tuples of the places must be given. `<.0..(N-1).>` is an abbreviation of the symbolic sum `<.0.>+<.1.>+<.2.>`, when N is three. The number of philosophers `N` can be set to an arbitrary integer value without changing the whole net description.

The net description is compiled into an executable program. When a program is executed, it generates the reachability graph of the net. With command line options stubborn set method or symmetry method can be applied. With `strong` the strongly connected components of the graph can be computed and with `probe` queries about the properties of the graph can be made.

```
#define N 3
#place thinking lo(<.0.>) hi(<.(N-1).>) mk(<.0..(N-1).>)

#place withLeft lo(<.0.>) hi(<.(N-1).>)

#place eating lo(<.0.>) hi(<.(N-1).>)

#place forks lo(<.0.>) hi(<.(N-1).>) mk(<.0..(N-1).>)

#trans takeRight
in{
    withLeft: <.x.>;
    forks: <.(x+1)%N.>;
}
out{
    eating: <.x.>;
}
#endtr

#trans putDown
in{
    eating: <.x.>;
}
out{
    thinking: <.x.>;
    forks: <.x.>+<.(x+1)%N.>;
}
#endtr

#trans takeLeft
in{
    thinking: <.x.>;
    forks: <.x.>;
}
out{
    withLeft: <.x.>;
}
#endtr
```

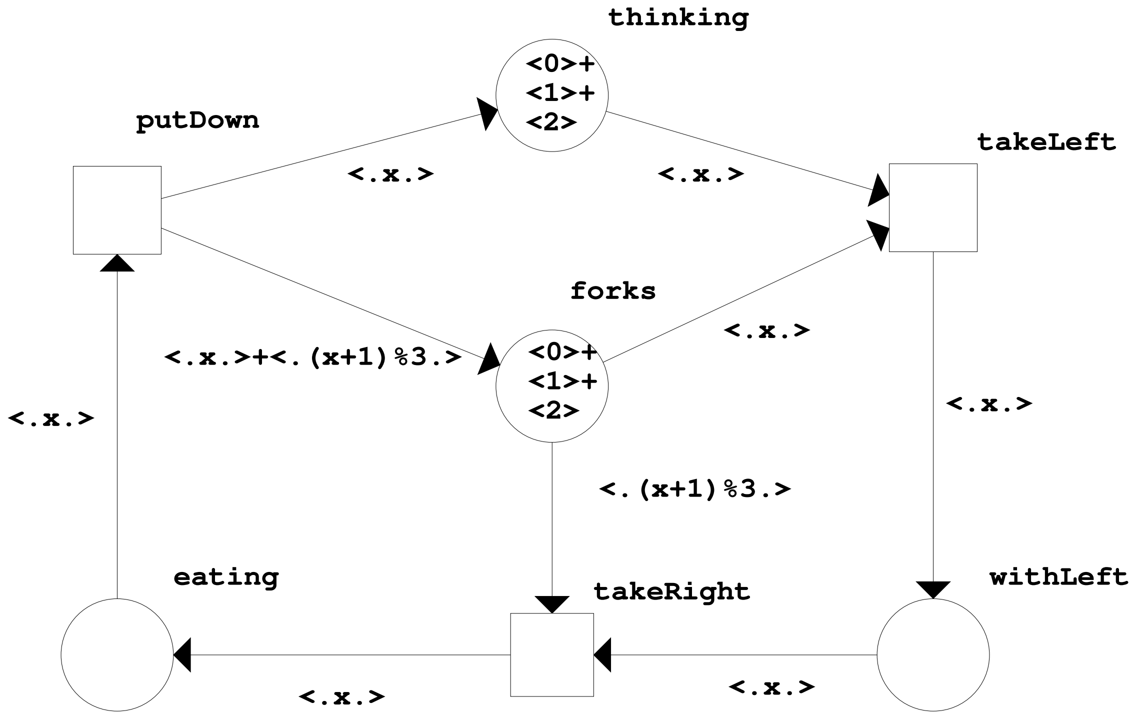

Figure 4: Pr/T-net of the dining philosophers problem drawn with `pred`.

3.2.1 Reachability Graph Generation

There are different alternatives, how a reachability graph of a Pr/T-net can be generated with `PROD`. It is possible to generate the complete reachability graph, stubborn set method with *incremental* and deletion algorithms, or symmetry method can be applied. The incremental algorithm is faster than the deletion algorithm, however the stubborn sets produced by it may contain unnecessarily many enabled transitions.

Before `PROD` uses stubborn set or symmetry method, it unfolds the Pr/T-net into a place/transition net (P/T-net). `PROD` applies the methods to the P/T-net and `probe` folds the information of the reachability graph of the P/T-net back to the Pr/T-net level. Predicate/transition nets can be thought of as an compressed representation of equivalent P/T-nets. In P/T-nets markings are represented by tokens, which do not have any internal structure. Every arc of a P/T-net has an arc weight and every place has a capacity. Arc weight indicates, how many tokens are swallowed from a place or produced to a place, when a transition fires. Capacity is the maximum number of tokens that a place can have. At marking M the transition t is enabled, if and only if in all input places s of t the number of tokens is at least the arc weight of an arc leading from s to t and in all output places s' of t the number of tokens is at most the difference between the capacity of place s' and the arc weight of the arc leading from t to s'.

A Pr/T-net is unfolded to a P/T-net in the following way.

- For every place of the Pr/T-net as many places are generated as is the number tuples in that place. In the example net every place can have three tuples, the arity of which are 1. So the unfolded Pr/T-net has $3 * 4 = 12$ places.

- For every feasible α-instance of every transition of the Pr/T-net, a transition and arcs are generated. The symbolic sums determine the arc weights. There are three substitutions ($\alpha_1 : x \leftarrow 0$, $\alpha_2 : x \leftarrow 1$, $\alpha_3 : x \leftarrow 2$), which can be applied to every transition of the Pr/T-net. So the number of transitions in the P/T-net is $3 * 3 = 9$. A flow from places to transitions and conversely consists of 1-tuples, which are substituted by a constant 0, 1, or 2. In the P/T-net a constant can be replaced by a single token, because the number of the philosopher is determined by the transitions the philosopher fires. So every arc weight is 1.

- The initial marking of the Pr/T-net determines the number of tokens in places of the P/T-net. In the initial marking of the P/T-net, places that indicate that initially all philosophers are thinking and none of the forks is in use, have a token.

- Such transitions and places of the P/T-net are removed, to which there is not a directed path from a place that has one or more tokens in the initial marking or from a transition that has no input places.

In Figure 5 the unfolded Pr/T-net of the dining philosophers problem is seen. The number of places and transitions has been tripled with respect to the number of places and transitions of the Pr/T-net.

3.2.2 Query Language

The reachability graph of the Pr/T-net generated by `PROD` can be examined with `probe`. Below a few examples of simple `probe` queries about the dining philosophers net are seen.

The prompt of `probe` consists of the number of current node followed by `#`. A node in the reachability graph is a reachable marking of the net. It is possible to go to any node in the graph with the command `goto` and to look, what marking the node represents with the command `look`. The first node is of course the initial marking of the net, where every philosopher is thinking and all three forks are available. With `succ` command the enabled transitions and successor nodes of the current node are seen. There are three possibilities to choose a successor arrow in the initial node. Below arrow 2 is chosen, so `x` is substituted by constant 2 and the transition `takeLeft` is fired. After firing the current marking changes.

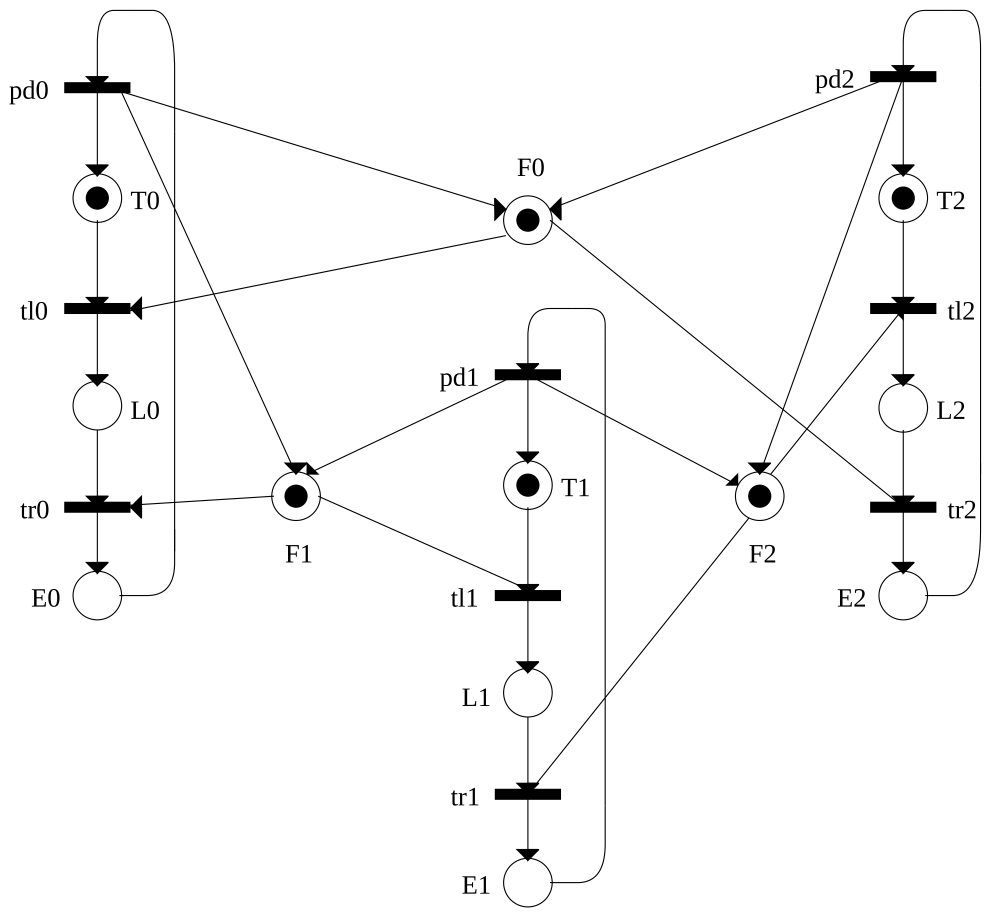

Figure 5: P/T-net of the dining philosophers problem.

```
0#look
Node 0, belongs to strongly connected component $$1
  thinking: <.0.> + <.1.> + <.2.>
  forks: <.0.> + <.1.> + <.2.>
------------------------------------------------
0#succ
Arrow 0: transition takeLeft, precedence class 0
  x = 0
to node 1
------------------------------------------------
Arrow 1: transition takeLeft, precedence class 0
  x = 1
to node 2
------------------------------------------------
Arrow 2: transition takeLeft, precedence class 0
  x = 2
```

```
to node 3
-------------------------------------------------
Node 0 has 3 successor arrows
-------------------------------------------------
0#next 2
3#look
Node 3, belongs to strongly connected component $$1
  thinking: <.0.> + <.1.>
  withLeft: <.2.>
  forks: <.0.> + <.1.>
-------------------------------------------------
```

With `statistics` useful information about the reachability graph can be seen. Number of nodes is the number of generated nodes in the graph. Every reachable marking is not necessarily generated, because the graph generation can be interrupted and partial reachability graphs can be examined too. A *real arrow* corresponds to a fired instance of a non-fact transition. A node is terminal, if it does not have any successor arrows. A *fact arrow* corresponds to an instance of a *fact transition*. Fact transitions are used to present undesirable situations, so enabled instance of a fact transition is undesirable. In the Pr/T-net of dining philosophers no fact transitions are used. A node has been completely processed, if all its immediate successor arrows are created. A graph is strongly connected, if and only if there is a path from each node to all other nodes. From terminal node there is not a path to any node, so the number of strongly connected components is 2 in the example net. A terminal strongly connected component is *trivial*, if and only if it is a terminal node or the whole graph.

`showends $0` returns the marking representing the terminal node.

```
0#statistics
Number of nodes: 14
Number of (real) arrows: 27
Number of terminal nodes: 1
Number of fact arrow source nodes: 0
Number of fact arrows: 0
Number of nodes that have been completely processed: 14
Number of strongly connected components: 2
Number of nontrivial terminal strongly connected components: 0
0#showends $0
Node 12, belongs to strongly connected component $$0
  withLeft: <.0.> + <.1.> + <.2.>
-------------------------------------------------
1 path end nodes
-------------------------------------------------
```

```
0#succ 12
Node 12 has 0 successor arrows
-----------------------------------------------
```

With query command nodes and paths satisfying given properties can be searched for. In the first query below probe searches for nodes, in which place eating is empty. The second query is similar to the first one, except a set is not built (volatile has been added into the query). Queries can be extended to multiple lines by typing "\" at the end of each line. The verbosity level of the query can be chosen by options mute, verbose, and superverbose. Finally all such paths from the initial node to the terminal node, in which the place eating is empty, are searched for. The paths consist of markings and arrow numbers between them.

```
0#query node eating == empty
   0,         1,         2,         3,         5,         6,
   8,        12
8 paths
Built set $2
-----------------------------------------------
0#query mute volatile node eating == empty
8 paths
0#query dpath(eating == empty, false) $0
   0  [0>    1  [1>    5  [1>   12,        0  [0>    1  [2>
   6  [1>   12,        0  [1>    2  [1>    5  [1>   12,
   0  [1>    2  [2>    8  [1>   12,        0  [2>    3  [1>
   6  [1>   12,        0  [2>    3  [2>    8  [1>   12
6 paths
Built set $3
```

Infinite event sequences can be detected with dspan and dpath queries. The query dspan(ϕ, true) false returns paths, in every node of which formula ϕ is true and which have an end cycle. dpath(ϕ, true) false returns all such paths. dspan ignores alternative routes, but is guaranteed to detect all those cycles that dpath detects. Below paths, which contain a cycle and in which a philosopher number 1 is continuously thinking, are detected with dspan and dpath. field[0] refers to the first field in the tuple, which is the number of the philosopher in the example.

```
0#query dspan((((thinking) : (field[0] == 1)) != empty), \
true) false
   0  [0>    1  [0>    4  [0>    0,         0  [0>    1  [0>
   4  [1>   10  [0>    3  [0>    9  [0>    0,         0  [0>
   1  [0>    4  [1>   10  [0>    3  [1>    6  [0>   10
3 paths
```

```
Built set $4
----------------------------------------------
0#query mute dpath((((thinking) : (field[0] == 1)) != empty), \
true) false
7 paths
Built set $5
```

To get more information about `PROD` or `probe`, see [2].

4 Frame Synchronized Ring

The Frame Synchronized Ring (FSR-bus) [3, 4, 6, 7] is a high throughput parallel data bus, which is developed for high speed multiprocessor applications in a research project in the Telecommunications Laboratory of the Technical Research Centre of Finland. In the future real time network applications demand more and more computation power from the software and hardware operating with the applications. To achieve the increasing computing demands many processors have to be connected together and the computation have to be parallelized. The FSR-bus is a high throughput solution with the aid of which the processing bottleneck is tried to be removed. Areas, to which the FSR-bus will be applied, include high speed packet switching in computer networks, multimedia coding, real time video compression, and real time processing of security algorithms.

4.1 Structure of the Bus

The FSR-bus belongs to the category of slotted rings with destination release policy. The FSR-bus consists of a number of nodes connected together with a parallel data bus. In addition to the data bus, three control lines connect the nodes together: bus clock, frame synchronization line, and request line. The basic structure of the FSR-bus is seen in Figure 6 [6]. The nodes consist of a ring interface with receive and transmit buffers. Peripherals that use the bus are attached to the nodes through a interface control unit. Interface control units are implemented using programmable gate arrays. All the signals in the data bus and control lines are synchronized by the bus clock. The bus clock signal is fed either parallel to the bus, as in Figure 6, or directly from the timing block to every node. Frame synchronization signal is used as an indication of the beginning of the frame. The nodes send requests to transmit in the request line. Note that the transmit direction of the requests is opposite to that of the data bus and other control signals.

The frames transmitted by the nodes consist of L time slots. Every time slot is as many bits wide as there are parallel lines in the data bus. The optimum number of time slots in the frame equals to the number of nodes connected to the bus. The frames are transmitted one time slot at a time controlled by the bus clock. The time period that it takes from a time slot to get from an output of one node to the output of the next node, is called a bus cycle.

The bus timing signals [6] are shown in Figure 7. When a node has data to transmit, it has to wait for the frame synchronization signal (frame pulse), before it can start the transmission. The frame synchronization signal indicates the start of the frame and it circulates around the ring parallel with the

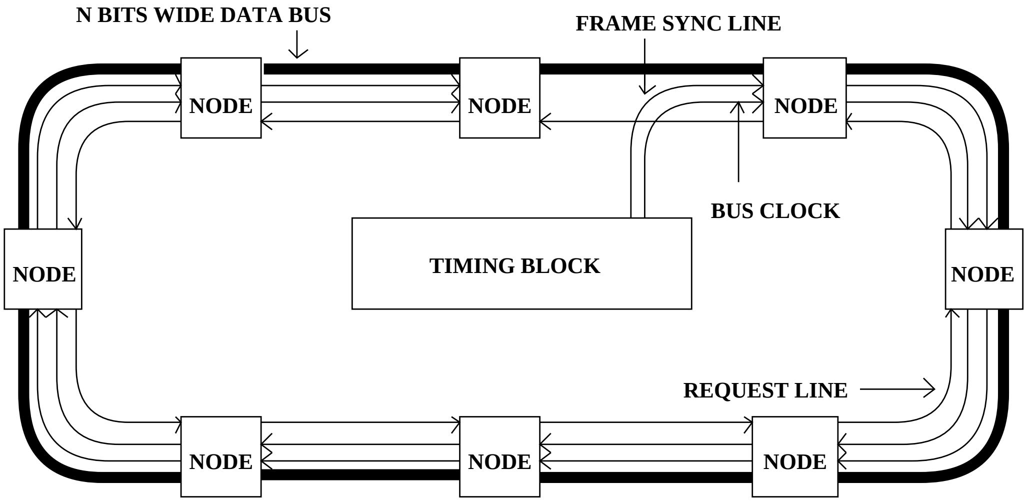

Figure 6: Block diagram of the FSR-bus.

first time slot of the frame. First time slots are darkened in Figure 7. Frames consist of seven time slots.

4.1.1 Frame Structure

The frames of the FSR-bus are constituted of fields. The size of the field in time slots depends on the application. The first field in the frame is reserved for the protocol control information. Other fields include user data. Depending on the application the last field of the frame can include user data or control information. In the noisy environments, the last field of every frame could include for example the frame check sequence. The frame structure of the FSR-bus is seen in Figure 8 [4].

The first field in the frame is called a header and is divided into several

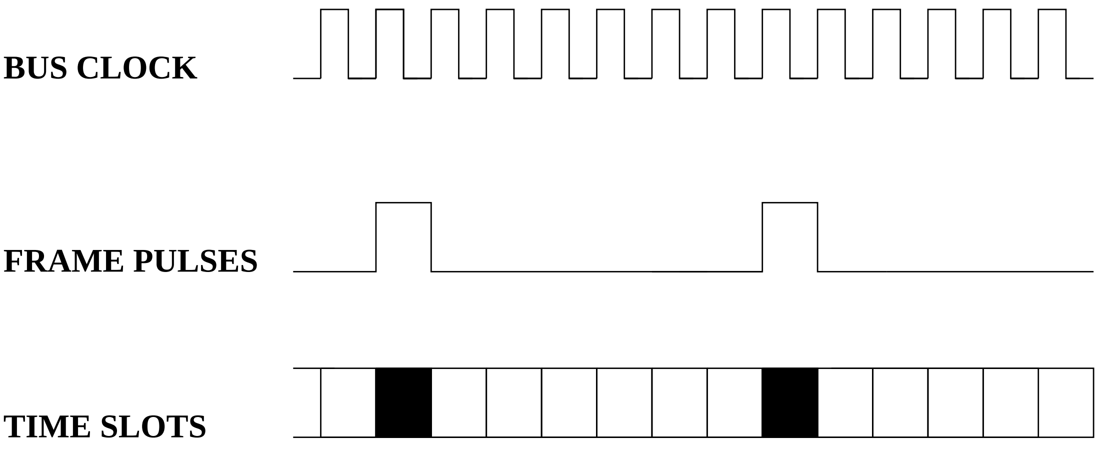

Figure 7: Bus timing signals.

subfields. When a frame passes the node, the node makes the decision, if it can send its own data in that frame by examining the bits in the header. The first bit in the header indicates, if the incoming frame is *occupied* or *empty*. If the same frame is transmitted to more than one node, the *group/single* bit is set. When a number of rings are connected together and a node wants to send data to the node connected to some other ring, *ring address* bits are used. The transmitting node sets the address of the receiver into *destination/group address* bits. The *source address* subfield includes the address of the sender of the frame. When a frame is transmitted from one ring to another, the source address is replaced by the address of the node that connects the rings. When a node receives a frame, the source address of which is its own address, it removes the frame from the ring. In that way frames, the reception of which are failed, are prevented from traveling in the ring forever. With *retransmit* bit the receiver can prevent the sender from removing the frame. If the receiver is temporarily incapable of receiving frames, it can set the retransmit bit and try to receive a frame later.

The rest of the subfields in the header are application dependent. With *start* and *end of packet* bits the transmitting node can indicate the receiving node, which frame is the first and which one is the last frame in the packet constituting of many frames.

4.2 Operation of the Bus

The nodes connected to the bus have the control of the bus. The peripheral attached to the node uses the bus to transmit and receive data. When a peripheral have data to send, it sends the data to the transmit buffer of the node. The node monitors the headers of by-passing frames in order to detect, if the incoming frame is available to the node (empty or addressed to the node), if the frame is to be deleted, or if the frame is just passing the node.

When the node receives data from the bus, it copies the time slots of the frame into the receiving buffer and removes the frame from the bus. The peripheral reads the data from the buffer. The rules, according to which nodes receive data, transmit data, and require service from the bus, constitutes the medium access control algorithm (MAC protocol) of the FSR-bus.

4.2.1 Medium Access Control Algorithm

Every node of the FSR-bus contains the control logic to monitor the by-passing frames. The nodes decide, if they can use the bus by monitoring the control signals and control fields of the header. The flow diagram of the medium access control algorithm is seen in Figure 9 [8].

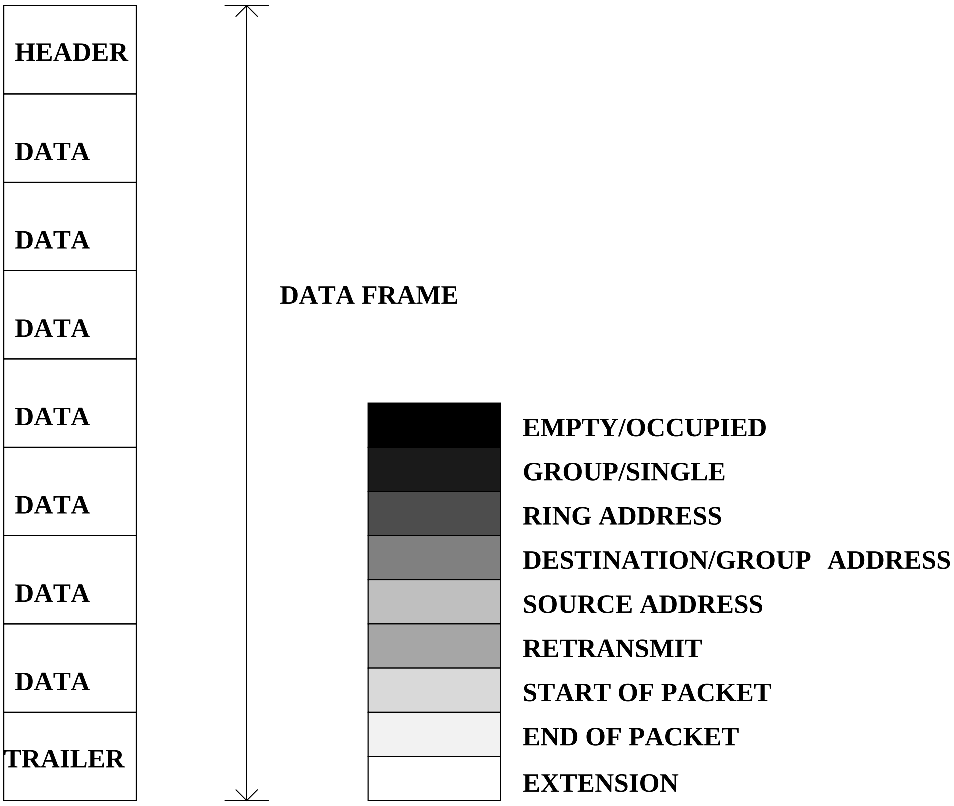

Figure 8: Frame structure of the FSR-bus.

When a node has data to send, it has to wait for an empty frame. The frame is empty, if it is not occupied or if it is emptied by the node. The node empties the frame, if the frame is addressed to it, or if the source address of the frame is the address of the node and retransmit bit is not set. The node can use the incoming empty frame, if it has a turn. Once the node has captured the empty frame and started the transmission, it continues transmitting as long as it receives empty frames or it is not interrupted by requests of other nodes. If the node does not have a turn, it must wait for other free frames or until it gets a turn.

The node gets a turn, if there are no earlier or simultaneous on-going requests, when the node gets ready to transmit. If the node does not have a turn, it must wait until other nodes have removed their requests to get the turn.

If the node is ready to transmit but not sending and it receives occupied frames, it can send its own request, if it has a turn. The request is held on the request line as long as the node does not start the transmission. The request is removed at the instant the node starts transmitting its frame. If the node does not have a turn to send a request, it must wait until other nodes have removed their requests.

When the node receives requests from other nodes, it either passes or suspends them. The received request is passed, if the node receives an occupied frame and the node is not holding its own request on the request line. The request is suspended, if the node is transmitting, receiving an empty frame, or the node is sending its own request. The suspended request is forwarded, when the node stops the action that caused the suspension. When the request is forwarded, it is not suspended until it is removed by the requesting node.

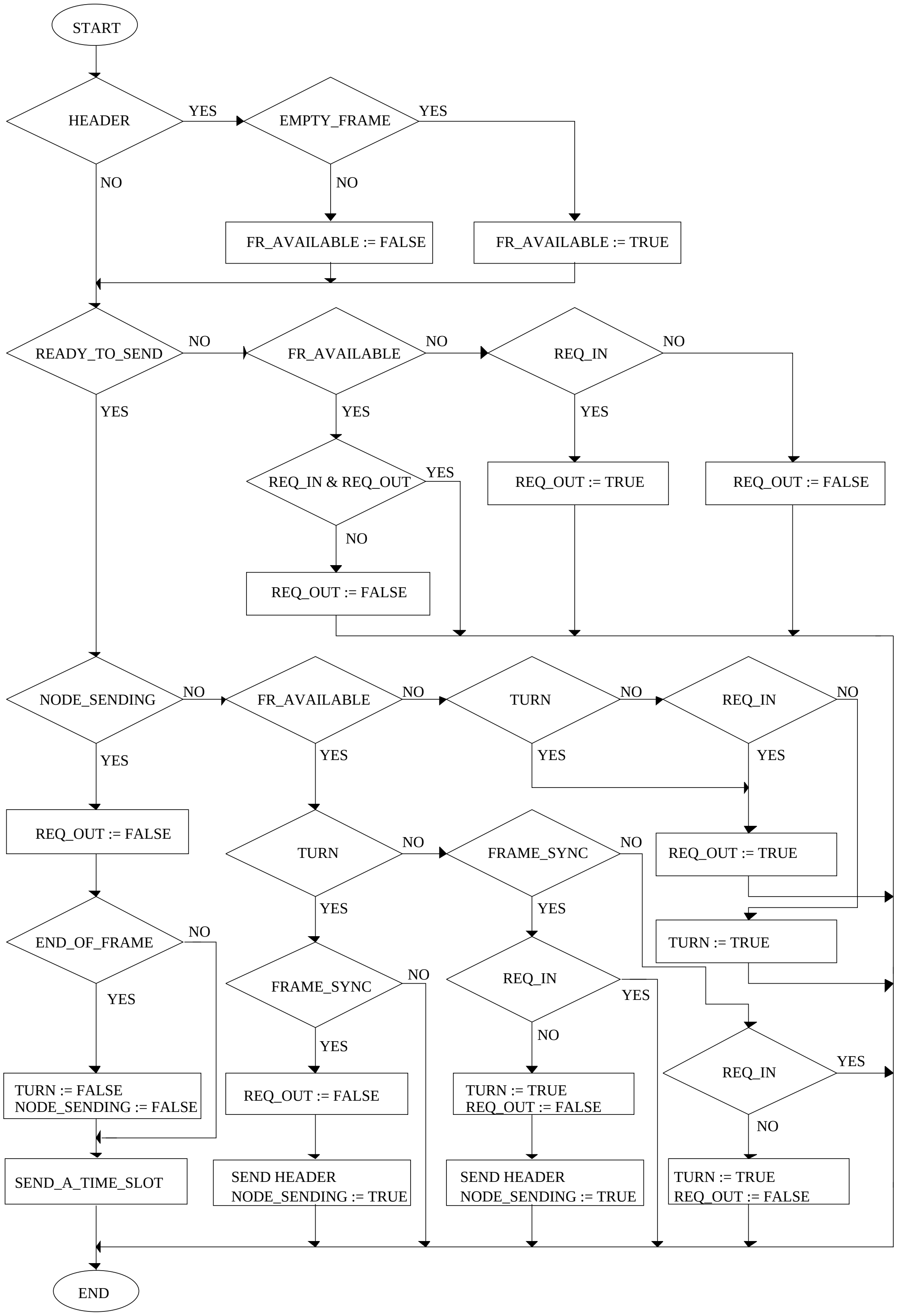

Figure 9: Flow diagram of the medium access control algorithm.

5 Pr/T-Net Model of the Bus

Two different predicate/transition net models of the Frame Synchronized Ring have been developed. The first model includes only the basic features of the ring. It is assumed that every node executes the MAC algorithm correctly. No error situations have been taken into account and the credit size is always 1. If the node is transmitting a frame, it cannot immediately start transmitting another frame, if other nodes send requests to the node, unless the credit size of the node is more than 1. When the credit size is 1, the node can transmit frames only as long as it does not receive requests from the other nodes. In the second model, the receiving node can fail and the credit size is arbitrary. If the credit size is C, every node can send at least C frames, before the transmission can be interrupted by requests of other nodes.

Two Pr/T-net models are tried to be made as simple as possible. The states between `START` and `END` (rectangulars) in the flow diagram of the MAC algorithm have been omitted in the Pr/T-nets. In the beginning, every node is in `START`. The node executes the algorithm in one step and enters the state `END`. The bus clock signal changes the state of every node back to `START`. This decreases the number of reachable states in the reachability analysis. The transitions of the Pr/T-net could be combined into a single transition to further decrease the number of reachable markings. However, it was found very important that the similarity of the Pr/T-net and the MAC algorithm could be easily seen by comparing the graphical representations of the net and the algorithm.

5.1 Basic Predicate/Transition Net

The structure of the FSR-bus is clearly seen in the Pr/T-net model of the bus in Figure 10. The places of the net correspond the components of the bus: nodes, data bus, frame synchronization line, and request line. The place `BUS` includes the frame synchronization line in its tuples, so it is not implemented as an own place. The bus clock line is different from other control lines of the FSR-bus. While other control signals are set into some value and possibly changed after some arbitrarily long time, bus clock cycles regularly. It is not necessary to save the value of the clock signal, because only the actions caused by the signal, when it changes, are interesting. That is why the bus clock signal is implemented as a transition. Place `NODE_ADDRESSES` includes the address of every node in the ring. When a node starts transmitting, it chooses the destination address randomly from that place. Every possible situation caused by alternative destination addresses is gone through in the reachability analysis of the Pr/T-net.

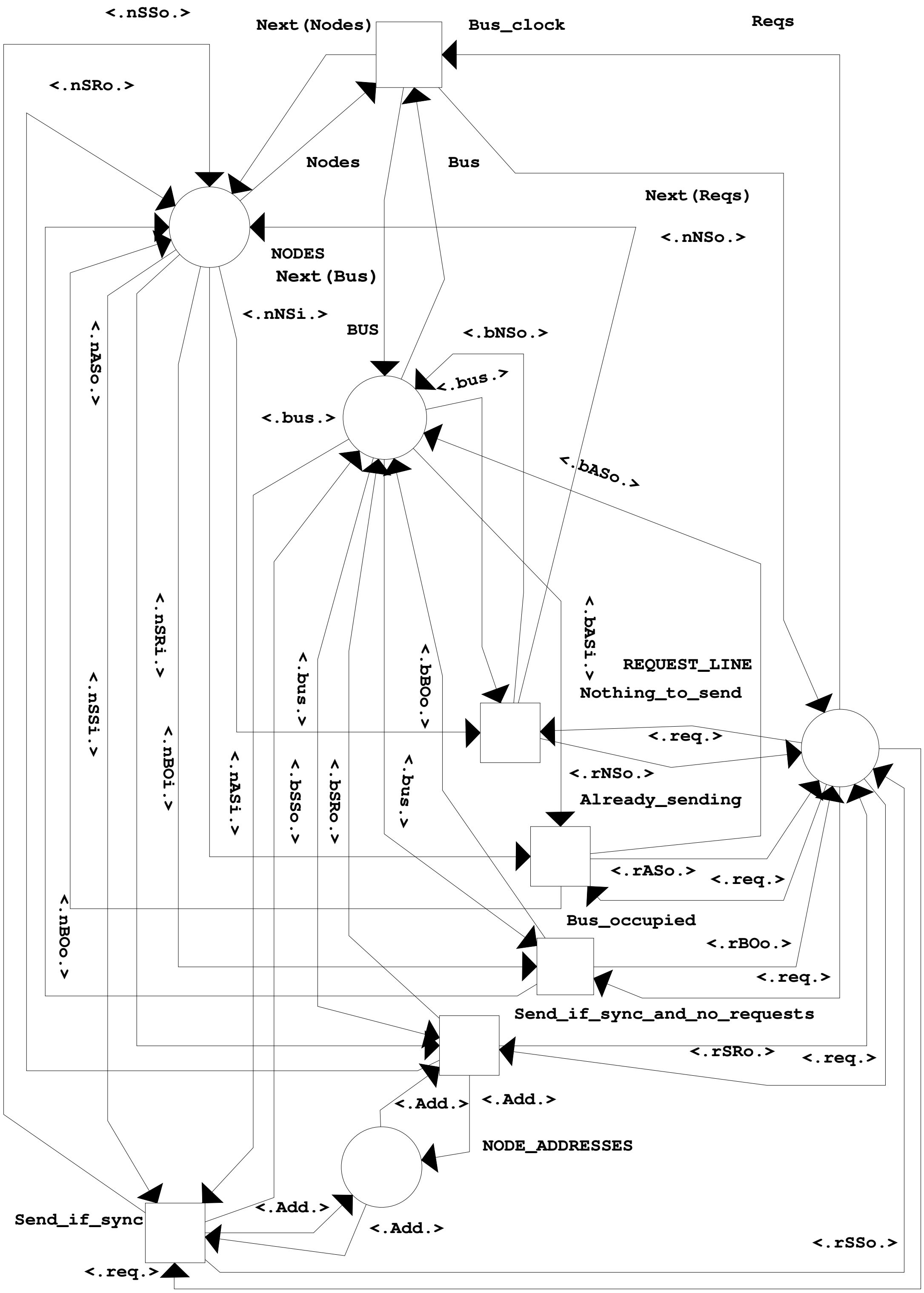

Figure 10: Pr/T-net model of the FSR-bus.

5.1.1 Markings of the Net

Every place contains as many tuples in symbolic sums as there are nodes in the ring. Tuples in place `NODES` are 7-tuples `<.n, BusClock, FrAvailable, ReadyToSend, Turn, NodeSending, TimeSlotsToSend.>`. `n` is the number of the node (`n` $\in [0, N-1]$) and N is the number of nodes in the ring. `BusClock`, `FrAvailable`, `ReadyToSend`, `Turn`, and `NodeSending` are binary variables. They contain the information according to which the node executes the medium access control algorithm. Every node executes the algorithm once in every bus cycle. `BusClock` is initially 0 and is set to 1, when the node has executed the MAC algorithm. When the bit is 1, the node has to wait for the bus clock signal (transition `Bus_clock`) that resets `BusClock`. `FrAvailable` indicates, whether a node can capture and use the incoming frame or not. If the node has data to send, it sets its `ReadyToSend` to 1. If the node has a turn, `Turn` is 1. `NodeSending` is 1 is as long as the node is sending a frame. `TimeSlotsToSend` is a counter that indicates the number of time slots in the frame. When the node starts transmitting, it decreases the counter every time it sends a time slot. When the counter has become 0, the node knows it has sent the whole frame. The initial value of the counter depends on N and the credit size used. With credit size 1, `TimeSlotsToSend` is initially set to the value $N - 2$. The header does not decrease the number of time slots. After sending $N - 2$ time slots + the header, the node knows the next time slot is the last one it sends.

Place `BUS` contains 4-tuples `<.n, Header, EmptyFrame, Address.>`. `n` is the number of the input bus segment of the node number `n` (`n` $\in [0, N-1]$). `Header` is 0, if the incoming time slot is a header. If the incoming time slot is not a header, `EmptyFrame` and `Address` are always 0. So ordinary time slots do not carry any information in the Pr/T-net model. If the incoming frames are not empty, `EmptyFrame` is 1 in the header. The destination address is seen in `Address`, if the time slots is a header (`Address` $\in [0, N-1]$). Because it is assumed that nodes do not fail in the Pr/T-net model of Figure 10, source address is not included in any tuple. In this model all transmitted frames are received and removed from the bus properly.

Place `REQUEST_LINE` includes 3-tuples `<.n,RequestIn,RequestOut.>`, where `n` is the number of the request line segment (`n` $\in [0, N-1]$). `RequestIn` is 1, if the node is receiving requests. If the node passes requests from other nodes or sends its own request, `RequestOut` is set to 1. Note that if there are N nodes in the ring, the Pr/T-net model of the ring contains $2N$ request line segments. Because the order of executions of the nodes is arbitrary, it cannot be known, if some node has changed the value of its output request, before the previous node uses this value. When a node is not ready to send, it needs both its input and output requests to decide, if it can reset the output request. Every node has to have access to both request values so that other nodes cannot change the values, while they are executing the algorithm.

The symbolic sum in place `NODE_ADDRESSES` consists of 1-tuples `<.Add.>`, where `Add` is the address of the node (`Add` $\in [0, N-1]$).

The initial marking of the net is seen below. Initially all the variables of the nodes are zero, node 0 is going to receive an empty frame (the frame synchronization signal is active in the input of the node 0), and no requests are circulating in the ring.

```
NODES: <.0..N-1,0,0,0,0,0,0.>
BUS: <.0,0,0,0.> + <.1..N-1,1,0,0.>
REQUEST_LINE: <.0..N-1,0,0.>
NODE_ADDRESSES: <.0..N-1.>
```

5.1.2 Transitions of the Net

It is assumed in the Pr/T-net of Figure 10 that every node executes the MAC algorithm in a single step. The order of executions of the nodes is arbitrary. This assumption can be done, because the result of the execution of each node depends only on its local variables and the data in the bus segment, to which it is connected. When a node executes the MAC algorithm, it replaces the values of its data bus and request line segments, and its own variables by new values. A node has own tuples, to which these values are saved. No other node can change the values of these tuples.

Because a node executes the MAC algorithm in a single step, the value of the bit `FrAvailable` cannot be used in the transitions directly. If `FrAvailable` were currently 0 and a node received an occupied header not addressed to it, the transition would still see that `FrAvailable` is 0, even though it should be 1. So whenever it is tested, what the value of `FrAvailable` is, it must be separately tested, what are the values of `Header` and `EmptyFrame`. This causes complicated arc expressions, but reduces the size of the state space of the reachability graph of the net. If `Header` is 1, the old value of `FrAvailable` must be used.

The Pr/T-net model of the FSR-bus contains six transitions. As already mentioned the transition `Bus_clock` can fire only, when the bus clock bit of every node is 1. When this transition fires, the bus clock bits are reset, the time slot in the input of the node is transmitted to the input of the next node, and the output request of the node is transmitted to the input request of the previous node. The value of the output request of the node remains unchanged. The transition `Bus_clock` does not have a selector. The action caused by the firing of the transition is seen below. In Figure 10 arc expressions that are not inside "$<$." and ".$>$" represent symbolic sums of tuples. Symbolic sums consisting of one tuple are inside "$<$." and ".$>$".

```
Nodes: <.0,1,fra0,r0,t0,ns0,f0.>+<.1,1,fra1,r1,t1,ns1,f1.>+
<.2,1,fra2,r2,t2,ns2,s2.>
Next(Nodes): <.0,0,fra0,r0,t0,ns0,f0.>+
<.1,0,fra1,r1,t1,ns1,f1.>+<.2,0,fra2,r2,t2,ns2,s2.>
Bus: <.0,h0,o0,a0.>+<.1,h1,o1,a1.>+<.2,h2,o2,a2.>
Next(Bus): <.0,h2,o2,a2.>+<.1,h0,o0,a0.>+<.0,h1,o1,a1.>
Reqs: <.0,reqIn0,reqOut0.>+<.1,reqIn1,reqOut1.>+
<.2,reqIn2,reqOut2.>
Next(Reqs): <.0,reqOut1,reqOut0.>+<.1,reqOut2,reqOut1.>+
<.2,reqOut0,ReqOut2.>
```

The flow diagram in Figure 9 is divided into five parts, each of which corresponds a transition in the model of Figure 10. In every transition, except `Bus_clock`, it is tested, if the incoming time slot is a header. In case the time slot is a header, the frame available bit is set or reset depending on that if the incoming frame is empty or not.

If a node is not ready to send, the transition `Nothing_to_send` is fired. The transition `Already_sending` is fired, if a node is ready to send and is sending a frame. If the node is ready to send but is not sending, the transition `Bus_occupied` is fired, if the incoming frame is not empty. Otherwise if the node does not have a turn, the transition `Send_if_sync_and_no_requests` is fired, and if it has a turn, the transition `Send_if_sync` is fired.

Most transitions do not have a selector. The conditions, when a transition is enabled, are given in arc expressions. The 4-letter arc expressions are named so that the first letter indicates which place the arc connects to the transition (`n NODES`, `b BUS`, and `r REQUEST_LINE`). The next two letters indicate the name of the transition (`NS Nothing_to_send`, `AS Already_sending`, `BO Bus_occupied`, `SR Send_if_sync_and_no_requests`, and `SS Send_if_sync`). The last letter indicates, if the arc is an input arc `i` or an output arc `o` of the transition.

When the transition `Nothing_to_send` fires, `FrAvailable` is set to 1, if the incoming frame is empty (and the incoming time slot is a header). If the incoming frame is addressed to the node, the node empties the frame by resetting the `EmptyFrame` and `Address` bits. The output request is set to 1, if there are incoming requests and the frame is not available. Note that when the transition fires, the new value of `RequestOut` is written into both `RequestIn` and `RequestOut`. It is meaningless, what the value of `RequestIn` is, after the transition has been fired, because before the bit is used again, the transition `Bus_clock` has changed its value. The new value of `RequestIn` is copied from `RequestOut` of the next node. The arc expressions of the transition are seen below. The space bar between "," and ">" indicates that that field of the tuple is the same as the previous field of the tuple.

```
nNSi: <.n,0,FrAvailable,0,0,0,0.>
nNSo: <.n,1,(Header == 0) ?
((!EmptyFrame || n == Address) ? 1 : 0) : FrAvailable,0,0,0,0.>
bus: <.n,Header,EmptyFrame,Address.>
bNSo: <.n,Header,(Header == 0) ? ((EmptyFrame == 1 &&
n == Address) ? 0 : EmptyFrame) : EmptyFrame,(Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 : Address) : Address.>
req: <.n,RequestIn,RequestOut.>
rNSO: <.n,(Header == 0) ? ((!EmptyFrame || n == Address) ?
((!(RequestIn == 1 && RequestOut == 1)) ? 0 : RequestOut) :
RequestIn) : ((FrAvailable == 1) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn), .>
```

When the node has started transmitting, it continues as long as it has time slots to send. When the transition `Already_sending` fires, the number of time slots is decreased by one. After the node has sent all its time slots, the variables `ReadyToSend`, `Turn`, and `NodeSending` are reset. The node sends only time slots (not a header), when the transition is fired. All incoming requests are suspended. The transition does not have a selector. The arc expressions of the transition are seen below. Note that some expressions are the same as in the transition `Nothing_to_send`.

```
nASi: <.n,0,FrAvailable,1,1,1,TimeSlotsToSend.>
nASo: <.n,1,FrAvailable,(TimeSlotsToSend == 0) ? 0 : 1,
(TimeSlotsToSend == 0) ? 0 : 1,(TimeSlotsToSend == 0) ? 0 : 1,
(TimeSlotsToSend > 0) ? (TimeSlotsToSend-1) : TimeSlotsToSend.>
bASi: <.n,1,0,0.>
bASo: <.n,1,0,0.>
rASo: <.n,0,0.>
```

The bus is occupied, if the incoming frame is not empty. A node gets a turn, if there are not incoming requests. `RequestOut` is true, if a node sends its own request or forwards other requests. Transitions `Nothing_to_send`, `Bus_occupied`, `Send_if_sync`, and `Send_if_sync_and_no_requests` are always enabled, when a node is not ready to send. If other transition than `Nothing_to_send` is chosen, `ReadyToSend` is set to 1 and kept in that value, until a node has sent every time slot of its frame. If the node has nothing to send, `ReadyToSend` remains unchanged. The decision, when a node becomes ready to send, could be made in a number of ways. For example the transition `Nothing_to_send` could set `ReadyToSend` to 1 with some probability, when it is fired. If other transitions can change the value of `ReadyToSend`, extra markings are avoided. A node can simultaneously become ready to send and send a header. If only the transition `Nothing_to_send` could change `ReadyToSend`, there would always be an extra marking, in which a node becomes ready to send, but cannot send a header.

The gate of the transition `Bus_occupied` prevents the firing, if the incoming frame is empty. The arc expressions and the gate are seen below.

```
nBOi: <.n,0,FrAvailable,ReadyToSend,Turn,0,0.>
nBOo: <.n,1,(Header == 0) ? ((!EmptyFrame || n == Address) ?
1 : 0) : FrAvailable,(ReadyToSend == 0) ? 1 : ReadyToSend,
(Turn == 0 && RequestIn == 0) ? 1 : Turn,0,0.>
bBOo: <.n,Header,(Header == 0) ? ((EmptyFrame == 1 &&
n == Address) ? 0 : EmptyFrame) : EmptyFrame, (Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 : Address) : Address.>
rBOo: <.n,(Turn == 1 || RequestIn == 1) ? 1 : RequestOut, .>
gate ((Header == 0 && (EmptyFrame == 1 && n != Address)) ||
(Header == 1 && FrAvailable == 0))
```

The transition `Send_if_sync_and_no_requests` is enabled, when the node receives an empty frame, but it does not have a turn (and it is ready to send but not sending). If the node does not receive requests from other nodes (and frame synchronization signal is active), it can start transmitting by sending a header. The destination address of the header is chosen from the place `NODE_ADDRESSES`. The address can be the address of any node in the bus. The node resets its output request, sets all its binary variables to 1, and sets `TimeSlotsToSend` to $N-2$, if it can start transmitting. If there is no frame synchronization or there are incoming requests, the node cannot send a header. In that case, node gets a turn and resets its output request, if there are no incoming requests. Note that while other arc expressions in Figure 10 are abbreviations of tuples, `Add` is the name of the variable `Add` in a tuple of arity 1. The gate of the transition does not allow the transition to fire, if the incoming frame is not empty. The arc expressions and the gate of the transition are seen below.

```
nSRi: <.n,0,FrAvailable,ReadyToSend,0,0,0.>
nSRo: <.n,1,(Header == 0) ? ((!EmptyFrame || n == Address) ?
1 : 0) : FrAvailable,(ReadyToSend == 0) ? 1 : ReadyToSend,
((Header == 0 && RequestIn == 0) || (Header == 1 &&
RequestIn == 0)) ? 1 : 0, (Header == 0 && RequestIn == 0) ?
1 : 0, (Header == 0 && RequestIn == 0) ? (N-2) : 0.>
bSRo: <.n,(Header == 0 && RequestIn == 0) ? 0 : Header,
(Header == 0 && RequestIn == 0) ? 1 : ((Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 : EmptyFrame) :
EmptyFrame), (Header == 0 && RequestIn == 0) ? Add :
((Header == 0) ? ((EmptyFrame == 1 && n == Address) ? 0 :
Address) : Address).>
rSRo: <.n,((Header == 0 && RequestIn == 0) ||
(Header == 1 && RequestIn == 0)) ? 0 : RequestOut, .>
gate ((Header == 0 && (EmptyFrame == 0 || n == Address)) ||
```

```
(Header == 1 && FrAvailable == 1))
```

The transition `Send_if_sync` is enabled, when the node has data to send but it is not sending, it receives an empty frame, and it has a turn. The node sends a header, the number of time slots is set to the maximum value $(N-2)$, and requests from other nodes are suspended. The arc expressions of the transition are seen below.

```
nSSi: <.n,0,FrAvailable,ReadyToSend,1,0,0.>
nSSo: <.n,1,(Header == 0) ? ((!EmptyFrame || n == Address) ?
1 : 0) : FrAvailable,(ReadyToSend == 0) ? 1 : ReadyToSend,1,
(Header == 0) ? 1 : 0, (Header == 0) ? (N-2) : 0.>
bSSo: <.n,(Header == 0) ? 0 : Header, (Header == 0) ? 1 :
EmptyFrame, (Header == 0) ? Add : Address.>
rSSo: <.n,(Header == 0) ? 0 : RequestOut, .>
```

The complete `PROD` net description of the Pr/T-net in Figure 10 is given in Appendix B.

5.2 Extended Pr/T-Net with Credit Size and Retransmit Bit

The basic model in Figure 10 is now extended to include simple error situations and credit size greater than 1. The credit size can be changed by increasing or decreasing the variable `TimeSlotsToSend`. It is assumed that the number of time slots in the frame is equal to the number of the nodes in the ring. The first time slot a node sends is always a header, which does not decrease the time slot counter. A node sends the last time slot, when the time slot counter is zero. So when the credit size C is greater than one, the number of time slots in the maximum credit is $C*(N-1)$, where N is the number of the nodes in the ring $(N>1)$. When N equals to one, credit size is unnecessary, because the only node in the bus can begin transmitting, whenever it has data to send. Note that when $C=1$, the maximal number of time slots of the basic model differs from that of the extended model. In the transition `Already_sending` of the extended model, the variables of a node are reset one time slot earlier than in the basic model.

Two error situations are added into the basic model. In the first one, the destination node is permanently uncapable of receiving frames and the source node has to remove the frame it has sent. In the second case, a node is temporarily uncapable of receiving frames. When for example the receiving buffer of the destination node overflows, it can not receive the incoming frame addressed to it. It sets the retransmit bit and the source node retransmits the frame immediately, if it has credit left. In case the source node does not have credit left, it can be assumed that the next frame the node sends, is a retransmission. In fact it does not matter, what the node sends, because in the model the frames do not carry data.

Error situations can occur, when a node is receiving the header of the frame. If the node has succeeded in receiving the header, it is assumed that every time slot that belongs to that frame is received without errors.

The Pr/T-net model of the FSR-bus in which the credit size is greater than one and errors occur is seen in Figure 11. Five transitions corresponding different parts of the flow diagram in Figure 9 belong to the figure, but are omitted for clarity. Errors can not occur, when the transitions `Already_sending` or `Bus_occupied` are fired, because it is assumed that errors occurs only, when a node is receiving header of the frame.

5.2.1 Markings of the Extended Model

Places `NODES`, `REQUEST_LINE`, and `NODE_ADDRESSES` include the same kind of tuples as in the basic model. The place `BUS` includes 6-tuples `<.n, Header, EmptyFrame, DestinationAddress, SourceAddress, Retransmit.>`. The number of the node that has sent the frame can be seen in `SourceAddress`. If the destination node fails to receive the header of the frame, `Retransmit` is set to 1. The initial marking of the net is seen below. C is the credit size and N is the number of the nodes. In the beginning, every node has a maximum credit size, no requests have been sent and the bus is empty. The credit size is set to the maximum value, because the transition `Send_if_sync_and_no_requests` must know, if there is credit left, before it can fire. The frame synchronization signal is in the input of the node number 0.

```
NODES: <.0..(N-1),0,0,0,0,0,C*(N-1).>
BUS: <.0,0,0,0,0,0,0.> + <.1..(N-1),1,0,0,0,0.>
REQUEST_LINE: <.0..(N-1),0,0.>
NODE_ADDRESSES: <.0..(N-1).>
```

5.2.2 Transitions of the Extended Model

The transition `Bus_clock` is similar to that in the basic model except the arc expression to and from the place `BUS` must now consist of 6-tuples instead of 4-tuples.

When a node is not ready to send and it is not uncapable of receiving frames, the transition `Nothing_to_send` is enabled. The selector of the transition prevents the firing, if the node has sent a frame, the reception of which has been failed, because in that case a node has data to send (the failed frame). Other effects caused by the transition are the same as in the basic model.

When a node does not have data to send and the reception of a frame addressed to it fails, the transition `Nothing_to_send_RE` is enabled. `RE` is an abbreviation of `receive error`. Again the node can not be the source address, because in that case it has data to send. When the transition fires, `Retransmit` is set to 1 in the arc expression `BEo` (bus error output) and `FrAvailable` is set to 0.

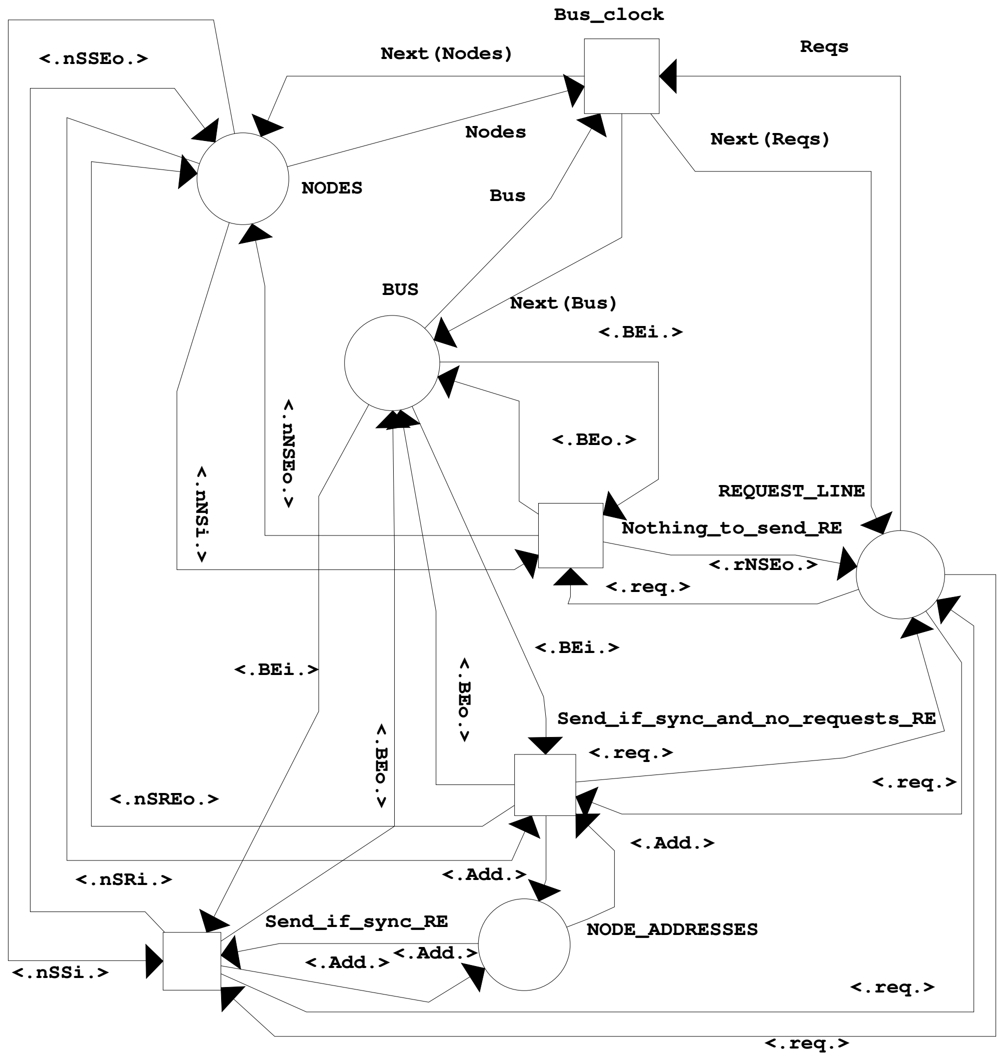

Figure 11: Pr/T-net model of the FSR-bus with credit size and retransmission bit.

When a node has data to send, but it is not sending and the incoming frame is not available, the transition `Bus_occupied` is enabled. Note from the selector of the transition that the node can not be the receiver of a frame, because in that case the frame is available. If the node is the source of the frame, the transition fires only, if the retransmit bit is 0.

The transition `Send_if_sync_and_no_requests` must be changed so that requests from other nodes can not suspend the transmission of the node, if it has credit left (time slot variable is greater than zero). The transition is enabled, if the node has data to send, but it is not sending yet and the incoming frame is empty or the source address of the frame is the address of the node and retransmit bit is 1. If the node has credit left, the node can send a header, even though there are incoming requests. When the credit runs out and the node receives requests from other nodes, it can not capture the incoming frame. Then it sets its credit to the maximum value, but does not send a header.

When a node has data to send and simultaneously is receiving a frame addressed to it, a receive error can occur. Then the transition `Send_if_sync-and_no_requests_RE` is enabled. When this transition fires, `Retransmit` is set to 1 and frame is not available to the node any more.

The transition `Send_if_sync` is enabled, when a node has data to send, turn, and frame is available, but the node is not sending yet. Even though the incoming frame is not empty, the transition can fire, if the node is the source node of the frame and the retransmit bit is 1.

If a node receives a header addressed to it, when the transition `Send_if_sync` is enabled, an receive error can occur. The credit size is then increased to the maximum value and retransmit bit is set to 1 by `Send_if_sync_RE`.

Once a node has sent a header, the transition `Already_sending` is fired as many times as there are time slots in a frame (not including a header). Between every frame `Already_sending` must not be enabled, even though `TimeSlotsToSend` is greater than 0. If a node has credit, it can start transmitting a new frame and `Already_sending` is enabled again.

The complete `PROD` net description of the Pr/T-net in Figure 11 is given in Appendix C.

6 Analysis of the Algorithm

All the analysis results presented in this Section are based on the reachability graph of the Pr/T-net model of the Frame Synchronized Ring. The reachability graph is automatically generated by `PROD`. With query language `probe` it is asked, if certain properties hold in the graph. The reachability graph of the basic Pr/T-net model of the FSR-bus includes 187 vertices and 403 edges, when the number of nodes in the ring (N) is 2. (From now on, a vertex or a state means a marking in a reachability graph and a node means a node in the FSR-bus.) The size of the reachability graph grows very fast in respect to N. When N is three, the graph includes 5 898 vertices and 18 852 edges, and when N is four, the graph consists of 139 471 vertices and 624 881 edges. The number of reachable states grows very fast because of different destination addresses, to which nodes can send frames. In the graphs mentioned above, a complete simulation of the behaviour of the FSR-bus has been carried out. If it is assumed that a node can send frames to itself only, which is the longest possible path a frame can travel in the ring, the graph consists of 454 436 vertices, when N is five. The shortest path a frame can travel, is a path form a node to its immediate successor. When every node can send frames only to its immediate successor, there are only 383 301 vertices in the reachability graph with six nodes in the ring. When N is three, the graph, where every node sends frames to itself, consists of 2 460 vertices and if every node send frames to its immediate successor only, there are 525 vertices in the reachability graph.

When the number of nodes in the ring was larger than four, the complete reachability graph could not be successfully generated. The number of reachable states was more than two million, and the disk space ran out. It took about four days to generate two million states, which consumed about 270 Mbytes disk space. Three methods were used for reducing the size of the reachability graph: stubborn set method, symmetry method, and bit packing. These methods will be discussed further in Section 6.4.

The state space explosion becomes even more serious problem, when the reachability graph of the Pr/T-net in Figure 11 is generated. The retransmit bit does not cause the explosion, but the greater the credit size the larger the reachability graph. With five nodes in the ring and credit size 2 the reachability graph contains almost five times as many vertices as the graph, in which the credit size is 1.

The size of the reachability graph generated by `PROD` can be examined with the command `statistics`. Below it is seen the sizes of the graphs of the basic and extended models, when $N = 2$, $N = 3$, and $N = 4$. Note that the number of strongly connected components in the basic model equals to that in the extended model, when N is the same. The reason, why there are more that one strongly connected component, is that once the ring has been set running, some initial situations can never occur again. For example in the beginning the frame available bit of every node is 0, but after the first bus clock signal, at least the frame available bit of one node is 1. If this was not the case, no node could ever send a frame.

```
0#statistics
Number of nodes: 187
Number of (real) arrows: 403
Number of terminal nodes: 0
Number of nodes that have been completely processed: 187
Number of strongly connected components: 4
Number of nontrivial terminal strongly connected components: 1

Number of nodes: 5898
Number of (real) arrows: 18852
Number of terminal nodes: 0
Number of nodes that have been completely processed: 5898
Number of strongly connected components: 10
Number of nontrivial terminal strongly connected components: 1

Number of nodes: 139471
Number of (real) arrows: 624881
Number of terminal nodes: 0
Number of nodes that have been completely processed: 139471
Number of strongly connected components: 28
Number of nontrivial terminal strongly connected components: 1

Number of nodes: 399
Number of (real) arrows: 927
Number of terminal nodes: 0
Number of nodes that have been completely processed: 399
Number of strongly connected components: 4
Number of nontrivial terminal strongly connected components: 1

Number of nodes: 22467
Number of (real) arrows: 76890
Number of terminal nodes: 0
Number of nodes that have been completely processed: 22467
Number of strongly connected components: 10
Number of nontrivial terminal strongly connected components: 1

Number of nodes: 685843
Number of (real) arrows: 3340925
```

```
Number of terminal nodes: 0
Number of nodes that have been completely processed: 685843
Number of strongly connected components: 28
Number of nontrivial terminal strongly connected components: 1
```

6.1 Deadlocks

The main goal of this work was to find every possible deadlock and event sequences that lead to deadlocks that might occur in the medium access control algorithm of the FSR-bus. There are two different cases, which have to be taken into account. Firstly, it must be proved that the MAC algorithm can not deadlock because of some kind of error in the algorithm. So every node can always execute the algorithm after the bus clock pulse. Secondly, it must be proved that the bus does not deadlock. The situation, in which the bus operates in such a way that no node can send data, even though the MAC algorithm of every node operates correctly, can be considered a deadlock (nothing happens in the bus). This second feature is tightly related to the fairness of the algorithm and it is examined further in Section 6.2.

From the `probe` queries in the previous section, it is easily seen that there are no terminal nodes in the reachability graphs of Pr/T-net models of the FSR-bus, when $2 \leq N < 5$. This means that in every possible situation that is modelled a node can execute the MAC algorithm and reach a reasonable state. Because the Pr/T-net models the correct operation of the MAC algorithm, error situations, due to for example noisy environments, cannot cause deadlocks in the reachability analysis. So every new state that is generated is reasonable. For example a situation, where a node is not ready to send, but it has a turn, is not a reasonable state, even though it may occur in the real bus. So whenever the nodes operate correctly, the bus cannot deadlock in the models.

Intuitively it is reasonable to assume that even though the number of nodes in the ring is increased, deadlocks do not occur. Adding a node into the ring just generates a new tuple into the places of the Pr/T-net. New tuples obey the same MAC algorithm as other nodes, only the number of possible destination addresses increases. The order, in which the frames and requests travel from node to node, changes, but principally the whole ring operates in the same way as earlier that is before one node was added.

Proposition 6.1 *There are no terminal nodes in the reachability graph of the Pr/T-net model of the Frame Synchronized Ring with arbitrary number of nodes.*

Proof: The proposition is proved by contradiction. Let the number of nodes in the Pr/T-net in Figure 10 be N and let us assume that there is one or more terminal nodes in the reachability graph of the net.

The deadlock in the model means that, after the moment the deadlock occurs, the transition `Bus_clock` can never be enabled. The transition is enabled, if and only if the bus clock bit of every node is 1. So if the transition is not enabled, at least one of the bus clock bits must be 0. If the transition is fired, the bus clock bit of every node is set to 0.

A node can execute the medium access control algorithm (fire one of the five transitions of the net in Figure 10), if and only if its bus clock bit is 0. After the execution the bus clock bit is 1, so the node can not execute the algorithm again, before the transition `Bus_clock` is fired.

At least one of the five transitions, corresponding the different parts in the MAC algorithm, is always enabled, when the bus clock bit of a node is 0. In the flow chart of the medium access control algorithm, there is always a path from `START` to `END`. It does not matter, which branch is chosen in decision diamonds, because every diamond has `YES` and `NO` branches. So it is impossible that no transition is enabled, when the bus clock bit is 0. When every node has once executed the algorithm, the `Bus_clock` transition becomes enabled, which leads to contradiction with our assumption that there are terminal nodes in the reachability graph. □

According to proposition 6.1 there are not deadlocks in the reachability graph of the Pr/T-net model of the FSR-bus, when the MAC algorithm is executed correctly by the nodes. This implies that the FSR-bus cannot deadlock. This is very important result, because now it is known that every deadlock that occurs in the bus is caused by the environment. The environment can for example change the request line in such a way that every node is continuously receiving requests. In such a situation no node can never get access to the bus.

6.2 Fairness

Fairness is closely related to deadlocks. The situation, where a node never gets access to the bus, can be considered a deadlock. In fact, this situation is a livelock, where a node waits for the access to the bus forever. If there are no deadlocks in the reachability graph of the FSR-bus and the bus can be proved to be fair, then the MAC algorithm is correct.

According to [3] the FSR-bus is fair, if every node having data to send gets access to the bus after a finite waiting time. According to the same paper the fairness can be defined by properties 6.2 and 6.3 given below.

Property 6.2 *A node that has issued a service request will be granted to get access to the bus.*

Property 6.3 *Every node, suspended from transmission, will be granted to issue a service request.*

In the reachability graph property 6.2 means that on every path starting from a vertex, in which a node has issued a service request, a node will eventually get access to the bus. Property 6.2 is easy to prove by contradiction. If at least one path is found, where a node has issued a request, but can never get the access to the bus, property 6.2 does not hold. The reachability graph of the FSR-bus consists of a finite number of vertices. So a path, in which a node has to wait forever, must include a cycle. A cycle is a path, the first marking of which equals to the last marking. When property 6.2 is negated and implication is changed using the tautologies ($A \Rightarrow B \Leftrightarrow \neg A \vee B$), ($A \wedge B \Leftrightarrow \neg(\neg A \vee \neg B)$), and ($\forall \Leftrightarrow \neg\exists\neg$), as in first-order predicate logic, the negated property can be read as: There is a cycle in a reachability graph starting from a vertex, in which a node has issued a service request, and in this cycle a node will never get access to the bus. This property is easily proved with `probe`.

The sketch of proof of property 6.2 with `probe` goes as follows.

- First such vertices in the reachability graph are searched for, where a node in the bus has issued a service request. A node has issued a service request, if both its request bits are 1. Note that when a node is forwarding requests from other nodes, its both request bits are 1 too. So it is also proved that no node sends requests infinitely long time. Note that only those vertices are searched, in which the bus clock bit of every node is 1. This decreases the number of vertices in the query set 80%, and `probe` computes the answer faster. This reduction can be done, because after a node has issued a request, its bus clock bit is 1 and both of its request bits are 1. So it does not matter, what the values of bus clock bits of other nodes are (Other nodes cannot affect the requesting node). The easiest situation occurs, when every bus clock bit is 1.
- From these vertices of the reachability graph a path, which contains a cycle and in which a node can not get access to the bus, is searched for. A node do not get access to the bus, if it is ready to send, but the node sending bit is 0.
- If such a path is found, it is possible that the cycle is looped forever. The node does not get access to the bus, the negation of property 6.2 holds, and the property 6.2 does not hold.

Property 6.2 is proved with the `probe` query seen below. The query returns every cycle, in which a node has issued a service request, but does not get access to the bus. If `probe` returns an empty set, there are no such cycles in the reachability graph and property 6.2 holds.

```
#define Property1(node_number) query volatile \
```

```
node((((REQUEST_LINE) : (field[0] == (node_number) && \
field[1] == 1 && field[2] == 1)) != empty) and \
fire(Bus_clock(1)) true) and \
dspan((((NODES) : (field[0] == 0 && \
field[3] == 1 && field[5] == 0)) != empty), true) false
```

The sketch of proof of property 6.3 is quite similar to that of property 6.2. Property 6.3 is negated and it is proved that the negation of the property does not hold. The negation of property 6.3 can be read as: There is a cycle in the reachability graph starting from a vertex, in which a node is suspended, and in this cycle a node can never issue a service request. This cannot be directly proved with `probe`, because there are three different reasons, why a node does not issue a request, even though it is suspended. 1) A node does not issue a service request, if it gets access to the bus, before it has issued a request. 2) If there are incoming requests, the node can not issue its own request. 3) When there are not incoming requests, it may be possible that a new request is received, before a node has issued its own request. The sketch of proof of property 6.3 with `probe` goes as follows.

- First such vertices of the reachability graph are searched for, where a node in the bus has been suspended. A node is suspended, if it is ready to send, but it does not send anything.
- A path starting from these vertices is searched for, which contains a cycle and in which 1) a node does not send a frame and 2) a node continuously receives requests of other nodes or 3) a node does not receive requests, but does not issue its own request.
- If such a path is found, it is possible that a cycle is looped forever. A node can not issue a service request, the negation of property 6.3 holds, and property 6.3 does not hold.

Property 6.3 can be proved with the `probe` query seen below. `probe` answers to the query by returning every cycle, in which a node is suspended, a node continuously receives requests from other nodes, a node never sends its own request, and a node never gets access to the bus. If `probe` returns an empty set, a suspended node either sends its own request or gets access to the bus. In that case property 6.3 holds.

```
#define Property2(node_number) query volatile \
node(((NODES) : (field[0] == (node_number) && field[3] == 1 \
&& field[5] == 0)) != empty) and dspan(((((REQUEST_LINE) : \
(field[0] == (node_number) && (field[1] == 1 || (field[1] \
== 0 && field[2] == 0)))) != empty) && (((NODES) : (field[0] \
== (node_number) && field[5] == 0)) != empty)), true) false
```

Now it is easy to prove properties 6.2 and 6.3. The properties are proved for every node in the model consisting of four nodes. So if every node, which is ready to send, will get access to the bus in a finite time, the FSR-bus is fair.

Proposition 6.4 *The Pr/T-net model of the FSR-bus in Figure 10 satisfies the properties 6.2 and 6.3.*

Proof: Let us assume that properties 6.2 and 6.3 does not hold in the Pr/T-net in Figure 10. Then `probe` queries find all such cycles, where the properties do not hold. Because such paths are not found, properties 6.2 and 6.3 do hold in the Pr/T-net.

```
0#Property1(0)
0 paths
-------------------------------------------------
0#Property2(0)
0 paths
-------------------------------------------------
0#Property1(1)
0 paths
-------------------------------------------------
0#Property2(1)
0 paths
-------------------------------------------------
0#Property1(2)
0 paths
-------------------------------------------------
0#Property2(2)
0 paths
-------------------------------------------------
0#Property1(3)
0 paths
-------------------------------------------------
0#Property2(3)
0 paths
-------------------------------------------------
```

□

The fairness of the Pr/T-net of the FSR-bus implies that the FSR-bus is fair. So no node has to wait forever to get access to the bus. Now it has been proved that the bus operates correctly.

6.3 Waiting Times of the Nodes

Even though the fairness of the FSR-bus has now been proved, it does not give us much information about the performance of the bus. A node that is ready to send, will start transmitting its data in a finite time, but how long this waiting time can actually be. In real time applications long waiting times are undesirable, because delayed frames are usually lost. If the `probe` queries of the previous Section are changed so that they do not detect infinite paths, but do detect all the paths, where a node is waiting, it should be possible to find for example the maximal waiting time of a node.

When a node becomes ready to send, it has to wait for a turn. When it has got a turn, it has to wait to 1) be allowed to issue a service request and 2) get access to the bus. These two waiting times are examined separately. First it is tried to find a maximal waiting time a node has to wait from the moment it gets ready to send, to the moment it can send a service request. Then it is tried to find the maximal waiting time of the node from the moment it gets ready to send to the moment it sends the header. All waiting times are measured in number of frames passing the node, before it can capture a frame.

6.3.1 Request Waiting Time

When a node becomes ready to send, it can not begin transmitting or issue its service request at once, if there is heavy traffic on the bus. The maximal time a node has to wait is now searched with `probe`. The heuristic to find the maximal waiting time goes as follows.

- From the reachability graph of the Pr/T-net such vertices are searched for, where a node in the bus is ready to send, but it does not have a turn and it is not sending. If a node has a turn, it can issue a request at once.
- Paths starting from these vertices are examined. The paths, on every vertex of which except the last one a node does not issue a request, and on the last vertex of which a node issues a request, are saved into a file.
- The longest path is chosen from the set of waiting paths saved.

Below two macros are seen, with which every waiting path, where a node is waiting to get a turn to issue its service request, is found. The macro `Wait_nodes` returns every vertex, in which a node is ready to send, is not sending, and does not have a turn. The macro `Wait_paths` is used in every vertex returned by `Wait_nodes`. `Wait_paths` returns every path, on every vertex of which except the last one, the request bits are not 0 and 1. When input request becomes 0, a node gets a turn to issue its request. The longest path is then found from the set of paths generated by `PROD` with the simple C program given in Appendix A. The program counts the number of transitions on the path. The longest path is a path, which contains the largest number of transitions. The program can be easily extended to find all equally long paths or for example ten longest paths.

```
#define Wait_nodes(n) query node(((NODES) : (field[0] == (n) \
&& field[1] == 0 && field[3] == 1 && field[4] == 0 && \
field[5] == 0)) != empty)

#define Wait_paths(n) query dpath(((((NODES) : (field[0] == \
(n) && field[3] == 1 && field[5] == 0)) != empty) && \
(((REQUEST_LINE) : (field[0] == (n) && (!(field[1] == 0 && \
field[2] == 1)))) != empty)), true) \
(((REQUEST_LINE) : (field[0] == (n) && field[1] == 0 && \
field[2] == 1)) != empty)
```

In the reachability graph of the Pr/T-net model of the FSR-bus, there are 7 184 vertices, where a node is waiting for a turn to issue a service request. From these vertices a maximal waiting time measured in by-passing frames is four, which is the same as the number of the nodes in the ring. One situation, in which the longest waiting time found occurs, goes as follows. A waiting node does not necessarily have to be node 0. Its number can be anything, as long as the order of the nodes does not change. The possible numberings of the nodes are seen in Figure 12. The rectangulars represent the nodes. For example if node 2 is chosen as a waiting node, then the output node of the waiting node is node 3 and the input node is node 1.

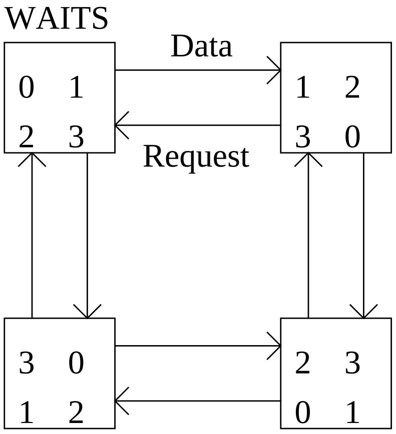

Figure 12: Possible numberings of the nodes.

- Node 0 becomes ready to send, when its both request out and request in bits are 1. It does not have a turn. Node 1 is next to node 0 and is sending requests, so it has a turn. The requests have reached the previous node of the node 0, which is node 3. The next node to the node 1 is node 2, which receives an empty frame and the frame synchronization signal. So it captures the frame. Node 3 has become ready to send, before it has received the request from node 2. So it has a turn.

- Node 2 captures a frame and sends it to itself.
- Node 3 has a turn, so it sends a frame to node 2.
- Node 2 sends a frame to itself again.
- Node 1 has been sending requests, so it gets a turn to send next.
- When node 1 removes its request, node 0 can issue its request. So it had to wait four frames to get a turn to issue a request.

6.3.2 Bus Access Waiting Time

To find the maximal waiting time from the moment a node becomes ready to send to the moment it finally gets access to the bus, all possible waiting situations have to be taken into account. From these paths the maximal path is found by the C program shown in Appendix A. There are totally 29 832 vertices in the reachability graph, where a node 0 is waiting to get access to the bus. For each of these vertices there are 5000 – 10 000 paths, at the last vertex of which a node begins transmitting. It was found too time and space consuming to go through all these alternatives, so two heuristics were used to find the maximal waiting time.

In the first heuristic, it is assumed that a maximal waiting time occurs in the same situation, when a node has to wait the maximal time, before it can issue its service request. The second heuristic is not so strong. It is assumed that a maximal waiting time occurs, when a node first has to wait a turn to issue a service request (perhaps only a short time period).

In the first heuristic maximal waiting paths were searched from every vertex, where a node has to wait a maximal time to get a turn to issue a request. The paths were found with the `Waiting_time` query seen below. The longest waiting path found was nine frames.

In the second heuristic, such vertices were searched for, where a node could not issue a request (it receives requests from other nodes) and the bus was not empty. This was done with the query `Waiting_vertices` seen below. `Waiting_vertices` returns 761 vertices. Paths starting from these vertices were examined with the query `Waiting_time` and longest path was found with the C program. The longest path found was ten frames and one such path is the following (waiting path for node 0). Node numbers can be anything, as long as the order does not change, as in Figure 12.

- Node 2 sends a frame to itself, so frame available bit of every other node is 0. Other nodes are not ready to send.
- Node 2 sends a frame to itself again, but now nodes 1 and 3 become ready to send.
- Node 0 becomes ready to send, when it receives a frame of node 2.
- Node 3 sends a frame to itself.
- Node 1 gets the turn. So an empty frame passes node 0.
- Node 1 sends a frame to itself. Node 0 gets a turn to issue a service request. Node 0 had to wait three frames, before it could issue a request.
- Node 2 sends a frame to node 1.
- Node 1 sends a frame to itself.
- Node 3 sends a frame to node 1.
- Node 1 sends a frame to itself.
- Node 2 sends a frame to node 1.
- Node 1 sends a frame to itself.
- Node 0 gets access to the bus after ten by-passing frames.

Note that in the maximal waiting path that was found, a node did not have to wait a maximal time to get a turn to issue a service request. This explains, why maximum waiting path with the first heuristic was only nine frames.

When the number of nodes in the Pr/T-net model was reduced to three, the maximal waiting path found was five frames. First a node had to wait two frames to get a turn to issue a request, after which it has to wait three additional frames to get access to the bus. The paths were searched with `Waiting_time` macro seen below, where the number of time slots was 1. One waiting path that was found with `PROD` and is five frames long goes as follows. Node 0 is waiting to get access to the bus.

- First node 0 sends a frame to node 2. Node 1 is not ready to send. When node 2 receives a frame, it sends a frame to itself. Nodes 0 and 1 are both ready to send.
- Node 2 sends a frame to itself.
- An empty frame passes node 0.

- Node 1 sends a frame to itself. Node 0 gets a turn and issues a service request.
- Node 2 sends a frame to node 1.
- Node 1 sends a frame to itself.
- Node 0 gets access to the bus.

The macro `Waiting_vertices` returns every vertex of the reachability graph, in which a node is ready to send, does not have a turn, is not sending, the input request line of the node is 1, and the node receives an occupied header that is not addressed to it. The macro `Waiting_time` is asked in the vertices found by `Waiting_vertices`. `Waiting_time` returns every path, in the last vertex of which a node begins transmitting and it has not sent any time slots.

```
#define Waiting_vertices(node_number) query volatile \
node((((NODES) : (field[0] == (node_number) && \
field[1] == 1 && field[2] == 0 && field[3] == 1 && \
field[4] == 0 && field[5] == 0)) != empty) && \
(((REQUEST_LINE) : (field[0] == (node_number) && \
field[1] == 1)) != empty) && (((BUS) : (field[0] == \
(node_number) && field[1] == 0 && field[2] == 1 && \
field[3] != (node_number))) != empty))

#define Waiting_time(node_number) query dpath((((NODES) : \
(field[0] == (node_number) && field[3] == 1 && field[5] == \
0)) != empty), false) (((NODES) : (field[0] == \
(node_number) && field[5] == 1 && field[6] == 2)) != empty)
```

The longest waiting time was found to be ten by-passing frames in the model of four nodes. When a model consists of three nodes, a longest waiting time found was only five frames. However, Raatikainen [9] has developed an equation, according to which the longest waiting time is 11 frames, when there are four nodes in the ring and each frame consists four time slots. In equation (3), the first term tells the number of frames a node has to wait to get access to the bus, when it has a turn. The third term tells the number of frames a node has to wait to get a turn to issue a service request. The sum expression decreases the waiting time, when the number of nodes in the ring is larger than five. T_{FR} is the number of time slots in a frame and can be omitted, if only by-passing frames are counted.

$$T_{Q_{MAX}}(N) = [(2^{N-1} - 1) - \sum_{i=\lfloor \frac{N+1}{2} \rfloor + 2}^{N-1} (2^{N-i} - 1) + N]T_{FR}, \quad N > 3. \qquad (3)$$

It was found that the longest time a node had to wait to get a turn to issue a request was four frames, as in equation (3). The longest waiting time a node had to wait to get access to the bus, when it already had a turn, was seven frames, as in equation (3). However, the longest waiting times never occurred in the same waiting situation. The longest waiting time occurred, when a node had to wait a turn to issue a request only three frames. One reason, why as long waiting paths as in equation (3) were not found, is that in the first heuristic, every situation, in which a node has to wait four frames to get a turn to issue a request, were not detected. This is caused by the fact that the exact time, when a node issues a request, can not be seen in the reachability graph. When a node issues a request, both request bits are set to 1. The bits are 1 too, if a node continuously receives requests from other nodes and cannot issue its own request. So the first or the last frame that passes the node is easily missed.

The maximal waiting times of the nodes are studied further. The Pr/T-net model and `probe` queries are developed so that equation (3) could be proved with `PROD`. However, it is very interesting to see that there can be quite long waiting times in the bus. In the real bus, the waiting times depend on the traffic on the bus. The situation that the longest maximal waiting time occurs, is rare and its probability decreases, when the number of the nodes in the ring increases.

6.4 Problems in Analysis

Though all the goals that were placed in Introduction have been achieved, a number of problems arose during this work. The main problem in the reachability analysis was the state space explosion. The largest reachability graph, which could be successfully generated, consisted of four nodes. A couple of methods were used to reduce the size of the state space. However, the use of these methods was found frustrating, because of the exponential growth of the computation time needed to generate the reachability graph.

When the number of reachable states increases, the computation time needed to find an answer to a `probe` query increases too. To go through every reachable state in a graph consisting of about one hundred thousand vertices takes several minutes with `probe`. It easily takes a couple of days to search for waiting paths in reachability graphs. Even though larger reachability graphs could be generated, the computation time to answer the queries could become unacceptably long.

6.4.1 Attacks against State Space Explosion

In PROD the reachability graph generator program unfolds a Pr/T-net into a P/T-net (place/transition net), before it applies the stubborn set or symmetry method. There was not enough memory to unfold the Pr/T-net of the FSR-bus, because the transition Bus_clock had so many low level instances. Transition Bus_clock has 20 binary variables, when there are only two nodes in the ring. So there are 2^{20} feasible substitutions, which can be applied to the variables of the arc expressions of the transition. The number of feasible substitutions increases, when nodes are added to the ring.

The transition Bus_clock was replaced by two transitions. Bus_clock_a gave the clock signal to the data bus and Bus_clock_b to the request line, as seen below. The number of feasible substitutions was reduced to 2^8. But an extra transition increased the number of reachable states. So when the stubborn set method was applied to the Pr/T-net of four nodes, the number of vertices in the graph was 46 768 (139 471) and the number of edges was 86 790 (624 881). The number in the parenthesis is the corresponding number of the reachability graph of the basic model. The computation of the reachability graph with stubborn set method took about one day. The generation of all the 139 471 vertices took a couple of hours. If the method reduced the size of the graph about to one third and the graph with five nodes included about four million vertices, it would take about a month to generate the reduced reachability graph of five nodes. However, it could be seen that the stubborn set method greatly reduced the number of edges in the reachability graph, even though the number of reachable states did not reduce as much as it was hoped.

```
#trans Bus_clock
in{
    REQUEST_LINE: <.0,reqIn0,reqOut0.>+<.1,reqIn1,reqOut1.>;
    BUS: <.0,h0,o0,a0.>+<.1,h1,o1,a1.>;
    NODES: <.0,1,fra0,r0,t0,ns0,f0.>+<.1,1,fra1,r1,t1,ns1,f1.>;
}
out{
    REQUEST_LINE: <.0,reqOut1,reqOut0.>+<.1,reqOut2,reqOut1.>;
    BUS: <.0,h2,o2,a2.>+<.1,h0,o0,a0.>;
    NODES: <.0,0,fra0,r0,t0,ns0,f0.>+<.1,0,fra1,r1,t1,ns1,f1.>;
}
#endtr

#trans Bus_clock_a
in{
    BUS: <.n0,0,o,a,1.>+<.n1,1,0,0,1.>;
```

```
}
out{
    BUS: <.0,(n0 == (N-1)) ? 0 : 1,(n0 == (N-1)) ? o : 0,
(n0 == (N-1)) ? a : 0,0.>+<.1,(n0 == 0) ? 0 : 1,(n0 == 0)
? o : 0,(n0 == 0) ? a : 0,0.>;
}
gate ((n0 == 0 && n1 == 1) || (n0 == 1 && n1 == 0));
#endtr

#trans Bus_clock_b
in{
    REQUEST_LINE: <.0,reqIn0,reqOut0,1.>+
<.1,reqIn1,reqOut1,1.>;
}
out{
    REQUEST_LINE: <.0,reqOut1,reqOut0,0.>+
<.1,reqOut2,reqOut1,0.>;
}
#endtr
```

When symmetry method was applied, in addition to the unfolding problem, the computation of symmetries was found to be very slow. It took almost five days to compute all the symmetries in the model of three nodes. The reachability graph included 3 556 (5 898) vertices. The computation of symmetries would take unacceptably long time, when a number of nodes is larger than three.

The generation of the reachability graph does not take a very long time, when none of the methods proposed previously is used. It is possible to generate more than million states in one day. With the third method that was applied, the disk space, needed by the graph, was tried to be reduced. A graph consisting of four million vertices can be generated in a week, if there is enough disk space, into which a graph can be saved.

The Pr/T-net model of the FSR-bus contains many binary variables. If one 32-bit memory word is used to express the markings of the net, it should be possible to generate more reachable states in the same space. All the variables in the places `NODES`, `REQUEST_LINE`, and `BUS` were joined into one pair, `NODEBUS`, where the first field was the number of the node and the second field included all its variables. The values of the variables are examined with macros that do bit operations. Below the macros, places, and one transition of the Pr/T-net, in which all variables are packed into one memory word, are seen. The reduced reachability graph took about 68% of the disk space consumed by the whole reachability graph. So only two million reachable states could be generated, before the disk space ran out. The complete reachability graph could not be generated.

```
#define TTTS0(n) (((n) & 512) | ((n) & 1024) | ((n) & 2048))
#define TTTS1(n) ((n) & 512)
#define TTTS2(n) ((n) & 1024)
#define TTTS3(n) ((n) & 2048)

#define ADD0(n) (((n) & 8192) | ((n) & 4096) | ((n) & 16384) |
((n) & 32768))
#define ADD1(n) ((n) & 4096)
#define ADD2(n) ((n) & 8192)
#define ADD3(n) ((n) & 16384)
#define ADD4(n) ((n) & 32768)

#define NISDEST(n,nb) (((n) == 0 && ADD0(nb) == 0) ||
((n) == 1 && ADD1(nb) != 0) || ((n) == 2 && ADD2(nb) != 0) ||
((n) == 3 && ADD3(nb) != 0) || ((n) == 4 && ADD4(nb) != 0))

#define BClock(n) ((n) & 256)
#define FrAvailable(n) ((n) & 128)
#define ReadyToSend(n) ((n) & 64)
#define Turn(n) ((n) & 32)
#define NodeSending(n) ((n) & 16)
#define RequestIn(n) ((n) & 8)
#define RequestOut(n) ((n) & 4)
#define Header(n) ((n) & 2)
#define EmptyFrame(n) ((n) & 1)

int OutNodeBits;

#place NODEBUS lo(<.0,0.>) hi(<.(N-1),65535.>)
mk(<.0,0.>+<.1..(N-1),2.>)

#place NODE_ADDRESSES lo(<.0.>) hi(<.N-1.>) mk(<.0..(N-1).>)

#trans Already_sending
in{
    NODEBUS: <.n,NodeBits.>;
}
out{
    NODEBUS: <.n,OutNodeBits.>;
}
gate (Header(NodeBits) != 0 && EmptyFrame(NodeBits) == 0 &&
BClock(NodeBits) == 0 && FrAvailable(NodeBits) != 0 &&
```

```
ReadyToSend(NodeBits) != 0 && Turn(NodeBits) != 0 &&
NodeSending(NodeBits) != 0 && (ADD0(NodeBits) == 0));
comp (OutNodeBits) {
  OutNodeBits = 386;
  if (TTTS2(NodeBits) != 0)
    OutNodeBits += 512;
  else if (TTTS3(NodeBits) != 0)
    OutNodeBits += 1024;
  if (!(TTTS0(NodeBits) == 0))
    OutNodeBits += 112;
  Accept(OutNodeBits);
}
#endtr
```

None of three methods that were tried to reduce the state space, was not efficient enough that the generation of the reachability graph of the net consisting of five nodes would have been succeeded. However, it is possible to join all the transitions of the net into one transition. In this way only every Nth marking of the net would be generated. This method is examined further.

6.4.2 Query Running Times

The computation time of a query can be greatly reduced, if the answer to the query is simply `YES` or `NO`. It took about eight hours for `probe` to answer the query of property 1 and about 30 hours to answer the query of property 2. Both queries returned an empty set. Both queries had to be asked four times, separately for every node number $(0, 1, 2,$ and $3)$. In the queries of the properties, no methods to speed up to computation or to reduce the space needed by the computation are used. If the same queries were used to prove the fairness of the Pr/T-net consisting of five nodes, the computation time would be very long.

The computation time increased, when `probe` searched for waiting paths (`probe` has to write every path into a file). Furthermore, all the waiting paths from a vertex consumed about six million bytes of disk space. That is why it was found impossible to go through all 29 832 waiting situations in the reachability graph of four nodes. To generate the waiting paths for one vertex took about twenty minutes. So to generate all the waiting paths would take more than 400 days. With two heuristics presented in Section 6.3 the longest waiting path was found from a set containing only 187 waiting situations.

The queries can be further developed to find answers more quickly. However, query running times were not found unacceptably long. Even though the paths could be found more faster, the disk space needed to save the paths would run out. One way to reduce the number of paths in the answer set, is to add more conditions into the queries. But when more conditions are added, it must be made sure that the desired path, for example a maximum waiting path, is still found in the reachability graph.

7 Conclusions

In this work predicate/transition nets were used in the modelling and analysis of the medium access control algorithm of the Frame Synchronized Ring. Pr/T-nets turned out to be a suitable choice, because the modelling of the bus was fast and easy with them, compared to other formalisms, for example process algebras and temporal logic. In addition, the analysis is very difficult and time consuming with temporal logic, but can be done automatically with Pr/T-nets.

The analysis of the bus is based on the reachability analysis. The reachability graphs of the nets were generated by a Pr/T-net reachability analysis tool, `PROD`. With `PROD` it was proved that the bus cannot deadlock by proving that there are no terminal nodes in the reachability graph of the net that models the bus. Two models of the bus were developed. One modelled the basic operation of the bus and the algorithm. The other included error situations, in which a reception of a frame failed. In addition, the nodes of the other model had credit, i.e., they could send more than one frame, before other nodes could interrupt them by sending requests. It was proved that neither of these models includes any terminal nodes. This was the main goal of this work.

The developers of the bus and in the future the users of the bus want to know, whether the bus can deadlock. However, the property has not been proved, until now. Even though the probability of a deadlock were negligible, one deadlock could cause unnecessary and even dangerous problems in real time applications. If the FSR-bus were used in packet switching, all alarm messages would be possibly lost in an emergency situation, if the bus deadlocked. That is why it is very important that it has been proved that the bus does not deadlock.

The state space of the reachability graph grew so fast that only a reachability graph of the net consisting of four nodes or less could be generated. The state space explosion of the reachability analysis was tried to be avoided with three methods: stubborn set method, symmetry method, and bit packing. Even though the computation time of the reachability graph increases, when these methods are applied, they turned out to be quite efficient. With stubborn set method the number of reachable states could be reduced 66% and the number of edges as much as 86% in the Pr/T-net of four nodes. In the bit packing all the variables of the net were packed into one 32-bit memory word. This reduced the disk space consumed by the reachability graph about 68%. The symmetry method was not as efficient as two other methods. In addition, the computation of symmetries was slow. Unfortunately, the state space grew so quickly that the reachability graph of the net consisting of five nodes could not be generated with any of these methods. However, important results about the efficiency of the methods were obtained.

The fairness of the FSR-bus was proved with `probe` queries. The FSR-bus is fair, if every node, which is ready to send, can start transmitting after a finite waiting time. It was proved that the reachability graph of the basic Pr/T-net of the FSR-bus with four nodes satisfies these properties. Fairness is an important property of a communication protocol. An unfair protocol cannot be used in any applications.

The property that the FSR-bus cannot deadlock and the fairness constitute the main results of this work. Both properties have been tried to be proved with simulations and analytical models. Due to randomness it is possible that some event sequence never occurs in simulations. Analytical models are very inaccurate. That is why the properties have not been able to be proved, until now. The results are very significant, because now one can be sure that whenever a node has a frame to send, it cannot deadlock and it does not have to wait forever to get access to the bus.

In real time applications it is not sufficient that the communication protocol is fair. In addition, it is important that nodes do not have to wait long to get access to the bus. When waiting times become long enough, frames are delayed, and they are received too late. A receiver has made a decision (possibly a wrong one) without them. This is why it was interesting to see, how long the nodes of the FSR-bus actually have to wait. Two waiting situations were examined. In the first one, it was studied, how long a node have to wait, before it can issue a service request. In the second one, the total waiting time, from the moment a node becomes ready to send to the moment a node starts transmitting, was examined. The longest waiting time that was found in the first case was four by-passing frames in the net of four nodes. The longest total waiting time that was found was ten frames. It is not proved that there are not longer waiting paths in the bus, because all 29 832 waiting situations were not examined. Anyway it is shown that in suitable situations the nodes have to wait long to get access to the bus. However, it was found that the longest queuing situation occurs very seldom.

Acknowledgements

This report is the result of my work in Digital Systems Laboratory at Helsinki University of Technology in 1993. I would like to thank professor Leo Ojala for giving me the opportunity to do this work. I would also like to thank Pertti Raatikainen of Telecommunications Laboratory of Technical Research Centre of Finland for valuable advice and comments. Raatikainen has spent his valuable time on checking the Pr/T-net of the FSR-bus with me. Juha Zidbeck has given me many valuable comments during discussions with him and Pertti Raatikainen. Johan Lilius, Mikko Tiusanen, and Kimmo Varpaaniemi have given me a lot of ideas and comments. I would like to thank them all. This work was funded by Telecom Finland.

References

[1] Genrich, H. J.: *Predicate/Transition Nets*. Braner, W., Reisig, W., and Rozenberg, G. (ed.). Petri Nets: Central Models and Their Properties, Advances In Petri Nets 1986, Part I. Lecture Notes in Computer Science 254. Berlin 1987, Springer-Verlag. pp 207–247.

[2] Grönberg, P., Tiusanen, M., and Varpaaniemi, K.: *PROD—A Pr/T-Net Reachability Analysis Tool*. Otaniemi 1993, Digital Systems Laboratory, Helsinki University of Technology, Series B: Technical Reports, No. 11. 44 p.

[3] Harju, J. and Raatikainen, P.: *Performance of the FSR-bus*. Proceedings of the Second Summer School on Telecommunications. Lappeenranta 1993, Lappeenranta University of Technology. pp 66–73.

[4] Harju, J., Raatikainen, P., Reinikka, T., and Zidbeck, J.: *Analysis of a Link Layer Protocol for High Speed Data Bus*. IEEE 18th Conference on Local Computer Network, Minneapolis, USA, September 19–22, 1993. New York 1993, IEEE Computer Society Press. pp 432–439.

[5] Jensen, K.: *Coloured Petri Nets*. Braner, W., Reisig, W., and Rozenberg, G. (ed.). Petri Nets: Central Models and Their Properties, Advances In Petri Nets 1986, Part I. Lecture Notes in Computer Science 254. Berlin 1987, Springer-Verlag. pp 248–299.

[6] Raatikainen, P. and Zidbeck, J.: *Implementation of the FSR-bus: a new high speed data bus solution*. Proceedings of the Second Summer School on Telecommunications. Lappeenranta 1993, Lappeenranta University of Technology. pp. 56–65.

[7] Raatikainen, P., Zidbeck, J., and Flinck, H.: *Frame Synchronized Ring: a Bus Concept for High Speed Processing Applications*. Integrated Broadband Communication Networks and Services (IBCN&S) Conference Proceedings, Copenhagen, Denmark, April 19–23, 1993. Copenhagen 1993, Elsevier Science Publicers, pp. 12.1.1–12.1.12.

[8] Raatikainen, P. and Zidbeck, J.: *Flow diagram of the FSR-bus*. To be published on VTT Publications Series later this year.

[9] Raatikainen, P.: *Upper bound of the queuing time of the nodes of the FSR-bus*. To be published on VTT Publications Series later this year.

[10] Rauhamaa, M.: *A Comparative Study of Methods for Efficient Reachability Analysis*. Otaniemi 1990, Digital Systems Laboratory, Helsinki University of Technology, Series A: Research Reports, No. 14. 61 p.

[11] Reisig, W.: *Petri Nets.* Berlin 1982, Springer–Verlag. 161 p.

[12] Valmari, A.: *Error Detection by Reduced Reachability Graph Generation.* Proceedings of the Ninth European Workshop on Application and Theory of Petri Nets. Venice 1988. pp. 95–112.

[13] Valmari, A.: *Heuristics for Lazy State Generation Speeds up Analysis of Concurrent Systems.* Mäkelä, M., Linnainmaa, S., and Ukkonen, E. (ed.), Proceedings of the Finnish Artificial Intelligence Symposium, Vol 2. Helsinki 1988. pp. 640–650.

[14] Varpaaniemi, K. and Rauhamaa, M.: *The Stubborn Set Method in Practice.* Jensen, K. (ed.). Application and Theory of Petri Nets 1992. 13th International Conference Sheffield, UK, June 22–26, 1992. Proceedings. Lecture Notes in Computer Science 616. Berlin 1992, Springer-Verlag. pp. 389–393.

Appendix A: Source Listing of the C Program for finding the Longest Path

```
/* A program for finding the longest path from the set of paths
   generated by PROD.

   The longest path is a path that contains the largest number
   of transitions.

   The program takes a file generated by PROD as an input. The
   input file consists of paths separated by commas. The path
   that contains the largest number of transitions is written
   into the output file.

*/

#include <stdio.h>
#include <string.h>

#define MAXSIZE 64000             /* Maximum path length */

void main(int argc, char **argv)
{
  FILE *in,*out;
  char buffer[MAXSIZE];
  char longest[MAXSIZE];
  int i,c,t,maxt;

  if ((in = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "%s: can't open %s\n", argv[1]);
    exit(1);
  }

  i = 0; t = 0; maxt = 0;
  while ((c = getc(in)) != EOF) {
    if (c != ',') {         /* Paths are separated by commas */
      buffer[i] = (char) c;
      i++;
      if (c == '>')              /* > indicates a transition */
        t++;
    }
    else {
```

```
      buffer[i] = '\0';
      if (t > maxt) {
        strcpy(longest,buffer);
        maxt = t;
      }
      t = 0;
    }
  }
  if ((out = fopen("longest.log", "w")) == NULL) {
    fprintf(stderr, "%s: can't open longest.log\n");
  }
  else {
    i = 0;
    while (longest[i] != '\0') {
      putc(longest[i], out);
      i++;
    }
  }
  fclose(in);
  fclose(out);
  printf("%d\n",maxt);
}
```

Appendix B: Basic Net Description of the FSR-Bus

```
#define N 4

#place NODES lo(<.0,0,0,0,0,0,0.>) hi(<.N-1,1,1,1,1,1,N-2.>)
mk(<.0..N-1,0,0,0,0,0,0.>)

#place REQUEST_LINE lo(<.0,0,0.>) hi(<.N-1,1,1.>)
mk(<.0..N-1,0,0.>)

#place BUS lo(<.0,0,0,0.>) hi(<.N-1,1,1,N-1.>)
mk(<.1..N-1,1,0,0.>+<.0,0,0,0.>)

#place NODE_ADDRESSES lo(<.0.>) hi(<.N-1.>) mk(<.0..N-1.>)

#trans Send_if_sync
in{
    NODE_ADDRESSES: <.Add.>;
    NODES: <.n,0,FrAvailable,ReadyToSend,1,0,0.>;
    BUS: <.n,Header,EmptyFrame,Address.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    NODE_ADDRESSES: <.Add.>;
    BUS: <.n,(Header == 0) ? 0 : Header,(Header == 0) ? 1 :
EmptyFrame,(Header == 0) ? Add : Address.>;
    REQUEST_LINE: <.n,(Header == 0) ? 0 : RequestOut,
(Header == 0) ? 0 : RequestOut.>;
    NODES: <.n,1,(Header == 0) ? ((!EmptyFrame ||
n == Address) ? 1 : 0) : FrAvailable,(ReadyToSend == 0) ?
1 : ReadyToSend,1,(Header == 0) ? 1 : 0,(Header == 0) ?
(N-2) : 0.>;
}
gate ((Header == 0 && (EmptyFrame == 0 || n == Address)) ||
(Header == 1 && FrAvailable == 1));
#endtr

#trans Send_if_sync_and_no_requests
in{
    NODE_ADDRESSES: <.Add.>;
    NODES: <.n,0,FrAvailable,ReadyToSend,0,0,0.>;
    BUS: <.n,Header,EmptyFrame,Address.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
```

```
}
out{
    NODE_ADDRESSES: <.Add.>;
    BUS: <.n,(Header == 0 && RequestIn == 0) ? 0 : Header,
(Header == 0 && RequestIn == 0) ? 1 : ((Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 : EmptyFrame) :
EmptyFrame),(Header == 0 && RequestIn == 0) ? Add : ((Header ==
0) ? ((EmptyFrame == 1 && n == Address) ? 0 : Address) :
Address).>;
    REQUEST_LINE: <.n,((Header == 0 && RequestIn == 0) ||
(Header == 1 && RequestIn == 0)) ? 0 : RequestOut,
((Header == 0 && RequestIn == 0) || (Header == 1 &&
RequestIn == 0)) ? 0 :RequestOut.>;
    NODES: <.n,1,(Header == 0) ? ((!EmptyFrame ||
n == Address) ? 1 : 0) : FrAvailable,(ReadyToSend == 0) ? 1 :
ReadyToSend,((Header == 0 && RequestIn == 0) || (Header == 1 &&
RequestIn == 0)) ? 1 : 0,(Header == 0 && RequestIn == 0) ? 1 :
0,(Header == 0 && RequestIn == 0) ? (N-2) : 0.>;
}
gate ((Header == 0 && (EmptyFrame == 0 || n == Address)) ||
(Header == 1 && FrAvailable == 1));
#endtr

#trans Bus_occupied
in{
    NODES: <.n,0,FrAvailable,ReadyToSend,Turn,0,0.>;
    BUS: <.n,Header,EmptyFrame,Address.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,Header,(Header == 0) ? ((EmptyFrame == 1 && n ==
Address) ? 0 : EmptyFrame) : EmptyFrame,(Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 : Address) : Address.>;
    REQUEST_LINE: <.n,(Turn == 1 || RequestIn == 1) ? 1 :
RequestOut,(Turn == 1 || RequestIn == 1) ? 1 : RequestOut.>;
    NODES: <.n,1,(Header == 0) ? ((!EmptyFrame ||
n == Address) ? 1 : 0) : FrAvailable,(ReadyToSend == 0) ? 1 :
ReadyToSend,(Turn == 0 && RequestIn == 0) ? 1 : Turn,0,0.>;
}
gate ((Header == 0 && (EmptyFrame == 1 && n != Address)) ||
(Header == 1 && FrAvailable == 0));
#endtr
```

```
#trans Already_sending
in{
    NODES: <.n,0,FrAvailable,1,1,1,TimeSlotsToSend.>;
    BUS: <.n,1,0,0.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,1,0,0.>;
    REQUEST_LINE: <.n,0,0.>;
    NODES: <.n,1,FrAvailable,(TimeSlotsToSend == 0) ? 0 : 1,
(TimeSlotsToSend == 0) ? 0 : 1,(TimeSlotsToSend == 0) ? 0 : 1,
(TimeSlotsToSend > 0) ? (TimeSlotsToSend-1) :
TimeSlotsToSend.>;
}
#endtr

#trans Nothing_to_send
in{
    NODES: <.n,0,FrAvailable,0,0,0,0.>;
    BUS: <.n,Header,EmptyFrame,Address.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,Header,(Header == 0) ? ((EmptyFrame == 1 &&
n == Address) ? 0 : EmptyFrame) : EmptyFrame,(Header == 0) ?
((EmptyFrame == 1 && n == Address) ? 0 :Address) : Address.>;
    NODES: <.n,1,(Header == 0) ? ((!EmptyFrame ||
n == Address) ? 1 : 0) : FrAvailable,0,0,0,0.>;
    REQUEST_LINE: <.n,(Header == 0) ? ((!EmptyFrame || n ==
Address) ? ((!(RequestIn == 1 && RequestOut == 1)) ? 0 :
RequestOut) : RequestIn) : ((FrAvailable == 1) ?
((!(RequestIn == 1 && RequestOut == 1)) ? 0 : RequestOut) :
RequestIn),(Header == 0) ? ((!EmptyFrame || n == Address) ?
((!(RequestIn == 1 && RequestOut == 1)) ? 0 : RequestOut) :
RequestIn) : ((FrAvailable == 1) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn).>;
}
#endtr

#trans Bus_clock
in{
    REQUEST_LINE: <.0,reqIn0,reqOut0.>+<.1,reqIn1,reqOut1.>+
<.2,reqIn2,reqOut2.>+<.3,reqIn3,reqOut3.>;
```

```
    BUS: <.0,h0,o0,a0.>+<.1,h1,o1,a1.>+
<.2,h2,o2,a2.>+<.3,h3,o3,a3.>;
    NODES: <.0,1,fra0,r0,t0,ns0,f0.>+<.1,1,fra1,r1,t1,ns1,f1.>+
<.2,1,fra2,r2,t2,ns2,f2.>+<.3,1,fra3,r3,t3,ns3,f3.>;
}
out{
    REQUEST_LINE: <.0,reqOut1,reqOut0.>+<.1,reqOut2,reqOut1.>+
<.2,reqOut3,reqOut2.>+<.3,reqOut0,reqOut3.>;
    BUS: <.0,h3,o3,a3.>+<.1,h0,o0,a0.>+
<.2,h1,o1,a1.>+<.3,h2,o2,a2.>;
    NODES: <.0,0,fra0,r0,t0,ns0,f0.>+<.1,0,fra1,r1,t1,ns1,f1.>+
<.2,0,fra2,r2,t2,ns2,f2.>+<.3,0,fra3,r3,t3,ns3,f3.>;
}
#endtr
```

Appendix C: Extended Net Description of the FSR-Bus

```
#define N 4
#define C 2

#place NODES lo(<.0,0,0,0,0,0,0.>)
hi(<.(N-1),1,1,1,1,1,C*(N-1).>)
mk(<.0..(N-1),0,0,0,0,0,C*(N-1).>)

#place REQUEST_LINE lo(<.0,0,0.>) hi(<.(N-1),1,1.>)
mk(<.0..(N-1),0,0.>)

#place BUS lo(<.0,0,0,0,0,0.>) hi(<.(N-1),1,1,(N-1),(N-1),1.>)
mk(<.1..(N-1),1,0,0,0,0.>+<.0,0,0,0,0,0.>)

#place NODE_ADDRESSES lo(<.0.>) hi(<.N-1.>) mk(<.0..(N-1).>)

#trans Send_if_sync
in{
    NODE_ADDRESSES: <.Add.>;
    NODES: <.n,0,FrAvailable,ReadyToSend,1,0,TimeSlotsToSend.>;
    BUS: <.n,Header,EmptyFrame,DestinationAddress,
SourceAddress,Retransmit.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    NODE_ADDRESSES: <.Add.>;
    BUS: <.n,Header,(Header == 0) ? 1 : EmptyFrame,
(Header == 0) ? ((Retransmit == 0) ? Add :
DestinationAddress) :0,(Header == 0) ? n :
SourceAddress,0.>;
    REQUEST_LINE: <.n,(Header == 0) ? 0 : RequestOut,
(Header == 0) ? 0 : RequestOut.>;
    NODES: <.n,1,(Header == 0) ? 1 : FrAvailable,
(ReadyToSend == 0) ? 1 : ReadyToSend,1,(Header == 0) ? 1 : 0,
C*(N-1).>;
}
gate ((Header == 0 && (EmptyFrame == 0 ||
n == DestinationAddress || (Retransmit == 1 &&
n == SourceAddress))) || (Header == 1 && FrAvailable == 1));
#endtr
```

```
#trans Send_if_sync_and_no_requests
in{
    NODE_ADDRESSES: <.Add.>;
    NODES: <.n,0,FrAvailable,ReadyToSend,0,0,TimeSlotsToSend.>;
    BUS: <.n,Header,EmptyFrame,DestinationAddress,
SourceAddress,Retransmit.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    NODE_ADDRESSES: <.Add.>;
    BUS: <.n,Header,(Header == 0 && ((RequestIn == 0 &&
TimeSlotsToSend == 0) || TimeSlotsToSend > 0)) ? 1 : 0,
(Header == 0 && ((RequestIn == 0 && TimeSlotsToSend == 0) ||
TimeSlotsToSend > 0)) ? ((Retransmit == 0) ? Add :
DestinationAddress) : 0,(Header == 0 && ((RequestIn == 0 &&
TimeSlotsToSend == 0) || TimeSlotsToSend > 0)) ? n : 0,0.>;
    REQUEST_LINE: <.n,((Header == 0 && (RequestIn == 0 ||
TimeSlotsToSend > 0)) || (Header == 1 && RequestIn == 0)) ?
0 : RequestOut,((Header == 0 && (RequestIn == 0 ||
TimeSlotsToSend > 0)) || (Header == 1 && RequestIn == 0)) ?
0 : RequestOut.>;
    NODES: <.n,1,(Header == 0) ? ((EmptyFrame  == 0 ||
n == DestinationAddress || (Retransmit == 1 &&
n == SourceAddress)) ? 1 : FrAvailable) : FrAvailable,
(ReadyToSend == 0) ? 1 : ReadyToSend,
((Header == 0 && (RequestIn == 0 || TimeSlotsToSend > 0)) ||
(Header == 1 && RequestIn == 0)) ? 1 : 0,(Header == 0 &&
(RequestIn == 0 || TimeSlotsToSend > 0)) ? 1 : 0,
(Header == 0) ? ((RequestIn == 0) ? C*(N-1) :
((TimeSlotsToSend == 0) ? C*(N-1) : TimeSlotsToSend)) :
C*(N-1).>;
}
gate ((Header == 0 && (EmptyFrame == 0 ||
n == DestinationAddress || (n == SourceAddress &&
Retransmit == 1))) || (Header == 1 && FrAvailable == 1));
#endtr

#trans Bus_occupied
in{
    NODES: <.n,0,FrAvailable,ReadyToSend,Turn,0,
TimeSlotsToSend.>;
    BUS: <.n,Header,EmptyFrame,DestinationAddress,
SourceAddress,Retransmit.>;
```

```
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,Header,(n == SourceAddress && Retransmit == 0) ?
0 : EmptyFrame,(n == SourceAddress && Retransmit == 0) ? 0 :
DestinationAddress,(n== SourceAddress && Retransmit == 0) ?
0 : SourceAddress,Retransmit.>;
    REQUEST_LINE: <.n,(Turn == 1 || RequestIn == 1) ? 1 :
RequestOut,(Turn == 1 || RequestIn == 1) ? 1 : RequestOut.>;
    NODES: <.n,1,(Header == 0 && n == SourceAddress &&
Retransmit == 0) ? 1 : FrAvailable,(ReadyToSend == 0) ? 1 :
ReadyToSend,(Turn == 0 && RequestIn == 0) ? 1 : Turn,0,
C*(N-1).>;
}
gate ((Header == 0 && EmptyFrame == 1 &&
n != DestinationAddress && ((n == SourceAddress &&
Retransmit == 0) || n != SourceAddress)) ||
(Header == 1 && FrAvailable == 0));
#endtr

#trans Already_sending
in{
    NODES: <.n,0,FrAvailable,1,1,1,TimeSlotsToSend.>;
    BUS: <.n,1,0,0,0,0.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,1,0,0,0,0.>;
    REQUEST_LINE: <.n,0,0.>;
    NODES: <.n,1,FrAvailable,(TimeSlotsToSend == N ||
TimeSlotsToSend == (N-3)) ? 0 : 1,(TimeSlotsToSend == N ||
TimeSlotsToSend == (N-3)) ? 0 : 1,(TimeSlotsToSend == N ||
TimeSlotsToSend == (N-3)) ? 0 : 1,(TimeSlotsToSend > 0) ?
(TimeSlotsToSend-1) : TimeSlotsToSend.>;
}
#endtr

#trans Nothing_to_send
in{
    NODES: <.n,0,FrAvailable,0,0,0,TimeSlotsToSend.>;
    BUS: <.n,Header,EmptyFrame,DestinationAddress,
SourceAddress,Retransmit.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
```

```
}
out{
    BUS: <.n,Header,(Header == 0 && EmptyFrame == 1 &&
(n == DestinationAddress || (n == SourceAddress &&
Retransmit == 0))) ? 0 : EmptyFrame,(Header == 0 &&
EmptyFrame == 1 && (n == DestinationAddress ||
(n == SourceAddress && Retransmit == 0))) ? 0 :
DestinationAddress,(Header == 0 && EmptyFrame == 1 &&
(n == DestinationAddress || (n == SourceAddress &&
Retransmit == 0))) ? 0 : SourceAddress,(Header == 0 &&
EmptyFrame == 1 && n == DestinationAddress) ? 0 :
Retransmit.>;
    NODES: <.n,1,(Header == 0) ? ((EmptyFrame == 0 ||
n == DestinationAddress || (n == SourceAddress &&
Retransmit == 0)) ? 1 : 0) : FrAvailable,0,0,0,
C*(N-1).>;
    REQUEST_LINE: <.n,(Header == 0) ? ((!EmptyFrame ||
n == DestinationAddress || (n == SourceAddress &&
Retransmit == 0)) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn) :
((FrAvailable == 1) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn),
(Header == 0) ? ((!EmptyFrame ||
n == DestinationAddress || (n == SourceAddress &&
Retransmit == 0)) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn) :
((FrAvailable == 1) ? ((!(RequestIn == 1 &&
RequestOut == 1)) ? 0 : RequestOut) : RequestIn).>;
}
gate (Retransmit == 0 || n != SourceAddress);
#endtr

#trans Bus_clock
in{
    REQUEST_LINE: <.0,reqIn0,reqOut0.>+<.1,reqIn1,reqOut1.>+
<.2,reqIn2,reqOut2.>+<.3,reqIn3,reqOut3.>;
    BUS: <.0,h0,o0,da0,sa0,rt0.>+<.1,h1,o1,da1,sa1,rt1.>+
<.2,h2,o2,da2,sa2,rt2.>+<.3,h3,o3,da3,sa3,rt3.>;
    NODES: <.0,1,fra0,r0,t0,ns0,f0.>+
<.1,1,fra1,r1,t1,ns1,f1.>+<.2,1,fra2,r2,t2,ns2,f2.>+
<.3,1,fra3,r3,t3,ns3,f3.>;
}
out{
```

```
    REQUEST_LINE: <.0,reqOut1,reqOut0.>+<.1,reqOut2,reqOut1.>+
<.2,reqOut3,reqOut2.>+<.3,reqOut0,reqOut3.>;
    BUS: <.0,h3,o3,da3,sa3,rt3.>+<.1,h0,o0,da0,sa0,rt0.>+
<.2,h1,o1,da1,sa1,rt1.>+<.3,h2,o2,da2,sa2,rt2.>;
    NODES: <.0,0,fra0,r0,t0,ns0,f0.>+<.1,0,fra1,r1,t1,ns1,f1.>+
<.2,0,fra2,r2,t2,ns2,f2.>+<.3,0,fra3,r3,t3,ns3,f3.>;
}
#endtr

#trans Nothing_to_send_RE
in{
    NODES: <.n,0,FrAvailable,0,0,0,TimeSlotsToSend.>;
    BUS: <.n,0,1,n,SourceAddress,0.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    NODES: <.n,1,0,0,0,0,TimeSlotsToSend.>;
    BUS: <.n,0,1,n,SourceAddress,1.>;
    REQUEST_LINE: <.n,RequestIn,RequestIn.>;
}
gate (n != SourceAddress);
#endtr

#trans Send_if_sync_and_no_requests_RE
in{
    BUS: <.n,0,1,n,SourceAddress,0.>;
    NODES: <.n,0,FrAvailable,ReadyToSend,0,0,TimeSlotsToSend.>;
    NODE_ADDRESSES: <.Add.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,0,1,n,SourceAddress,1.>;
    NODES: <.n,1,0,(ReadyToSend == 0) ? 1 : ReadyToSend,0,0,
TimeSlotsToSend.>;
    NODE_ADDRESSES: <.Add.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
gate (n != SourceAddress);
#endtr

#trans Send_if_sync_RE
in{
    BUS: <.n,0,1,n,SourceAddress,0.>;
```

```
    NODES: <.n,0,FrAvailable,ReadyToSend,1,0,TimeSlotsToSend.>;
    NODE_ADDRESSES: <.Add.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
out{
    BUS: <.n,0,1,n,SourceAddress,1.>;
    NODES: <.n,1,FrAvailable,(ReadyToSend == 0) ? 1 :
ReadyToSend,1,0,C*(N-1).>;
    NODE_ADDRESSES: <.Add.>;
    REQUEST_LINE: <.n,RequestIn,RequestOut.>;
}
gate (n != SourceAddress);
#endtr
```