Model Checking of Mobile Systems and Diagnosability of Weakly Fair Systems

Vasileios Germanos

School of Computing Science

Newcastle University

A thesis submitted for the degree of

Doctor of Philosophy

November 2015

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor Victor Khomenko, for all the time and effort he spent on helping me and reading through my thesis and for all the guidance he gave me. His advice and encouragement throughout all stages of the project were invaluable and will always be of benefit to me.

Next, I would like to thank my master thesis supervisor, Maciej Koutny, for giving me the opportunity to work with the Advanced Model-Based Engineering and Reasoning (AMBER) group and for all the advice he has given me.

I would like to sincerely thank Stefan Haar, Stefan Schwoon and my supervisor, for giving me the opportunity to work with them.

I would like also to thank my examiners, Franck Pommereau and Jason Steggles, who provided encouraging and constructive feedback. I am grateful for their thoughtful and detailed comments.

I also would like to thank my supervisor in my bachelor degree (ASPETE, Athens), Gerasimos Pagiatakis, and one of my teachers there, Vagelis Vassilikos, who both inspired me to pursue my studies.

I would like to thank my friends both here in Newcastle and in Greece, for all their support and encouragement.

Last but not least, I would like to thank my mother Tula and my sisters Ageliki and Marina, for their love, support, encouragement and patience.

Abstract

This thesis consists of two independent contributions. The first deals with *model checking* of *reference passing systems*, and the second considers *diagnosability* under the *weak fairness* assumption.

Reference passing systems, like mobile and reconfigurable systems are everywhere nowadays. The common feature of such systems is the possibility to form dynamic logical connections between the individual modules. However, such systems are very difficult to verify, as their logical structure is *dynamic*. Traditionally, decidable fragments of π-calculus, e.g. the well-known Finite Control Processes (FCP), are used for formal modelling of reference passing systems. Unfortunately, FCPs allow only 'global' concurrency between processes, and thus cannot naturally express scenarios involving 'local' concurrency inside a process. This thesis proposes Extended Finite Control Processes (EFCP), which are more convenient for practical modelling. Moreover, an almost linear translation of EFCPs to FCPs is developed, which enables efficient model checking of EFCPs.

In partially observed systems, *diagnosis* is the task of detecting whether or not the given sequence of observed labels indicates that some unobservable fault has occurred. Diagnosability is an associated property, stating that in any possible execution an occurrence of a fault can eventually be diagnosed. In this thesis, diagnosability is considered under the weak fairness (WF) assumption, which intuitively states that no transition from a given set can stay *enabled* forever - it must eventually either fire or be disabled. A major flaw in a previous approach to WF-diagnosability in the literature is identified and corrected, and an efficient method for verifying WF-diagnosability based on a reduction to LTL-X model checking is presented.

Contents

List of Figures

Chapter 1

Introduction

Many modern computing systems are able to perform simultaneously several computations and also can interact with each other. This kind of ability is called *concurrency* and the systems share this property are named *concurrent systems*. However, concurrency increases their complexity making the detection of possible faults harder. Thus, it is very common most people experience software and hardware devices that do not perform as they should. To be overcome this effect, it is necessary to be developed formal verification techniques, e.g. model checking [17, 18], for verifying the correctness of such systems. Model checking checks whether a specification property of the system under consideration is satisfied. To be achieved this, initially a model must be designed, which describes the behaviour of the system, using a formal language, like Petri nets [97] and process algebras [20]. Then, all the possible states of the model are checked whether satisfy the given specification property of the system.

Moreover, it is often the case, during the design process of systems, faults are not possible to be ruled out. That happens because they may be part of the systems or the environment where the systems evolve. Diagnosis is the procedure that determines whether or not the system contains some fault. To that end, diagnosability [87] is a key specification property for formal verification of large and complex systems, because makes possible the detection of faults in a finite time after their occurrence.

Modern concurrent systems enjoy a set of features like reconfigurability, logical mobility and dynamic allocation of resources. The systems enjoy these features

are called *reference passing systems*. These features make these systems very complex and concurrency adds another layer of complexity too. As a result, formal verification becomes a very challenging task because existing formal languages are difficult to express the behaviour of these systems. Thus, new efficient formal languages should be developed that can specify reference passing systems and make their formal verification feasible.

This thesis introduces a formal specification language that is suitable for modelling reference passing systems. Also, it is provided its translation to an existing formalism for which an efficient verification technique has been developed in [52]. As it was mentioned earlier, diagnosability is a key property for formal verification. Haar et al. in [44] proposed the *weak* diagnosis. This diagnosis, although it uses the term 'weak', it is more powerful than the usual diagnosis as in [7]. Diagnosis can detect a fault that has occurred in the past [7]. However, weak diagnosis can reveal faults that are concurrent or in the future of the observation, under the weak fairness. Based on weak diagnosis, a first definition of diagnosability under weak fairness was proposed in [1]. However, that definition is incompatible with the notion of diagnosis in [44] and contains a major flaw. It is often the case that due to the presence of some independent concurrent action in a system, it is not possible this system to be diagnosed in a finite time. To this end, in this thesis a notion of weakly fair diagnosability [35, 36] which corrects and supersedes the one in [1], is presented and an efficient method for formally verifying weakly fair diagnosability is developed.

1.1 Formal Verification of Reference Passing Systems

Many contemporary systems enjoy a number of features that significantly increase their power, usability and flexibility [63]:

- *Dynamic reconfigurability*: The overall structure of many existing systems is flexible. Nodes in ad-hoc networks can dynamically appear or disappear; individual cores in Networks-on-Chip can be temporarily shut down to save power; resilient systems have to continue to deliver (reduced) functionality

even if some of their modules develop faults.

- *Logical mobility*: Mobile systems permeate our lives and are becoming ever more important. Ad-hoc networks, where devices like mobile phones and laptops form dynamic connections are common nowadays, and the vision of pervasive (ubiquitous) computing [54], where several devices are simultaneously engaged in interaction with the user and each other, forming dynamic links, is quickly becoming a reality.

- *Dynamic allocation of resources*: It is often the case that a system has several instances of the same resource (e.g., network servers or processor cores in a microchip) that have to be dynamically allocated to tasks depending on the current workload, power mode, priorities of the clients, etc.

The common feature of such systems is the possibility to form dynamic logical connections between the individual modules. It is implemented using *reference passing* [20, 88]. A module can become aware of another module by receiving a *reference* (e.g., in the form of a network address) to it, which enables subsequent communication between these modules. This can be thought of as a new (logical) *channel* dynamically created between these modules. We will refer to such systems as *Reference Passing Systems* (RPS).

As people are increasingly depended on the correct functionality of RPSs, the cost incurred by design errors in such systems can be extremely high. However, even the conventional concurrent systems are notoriously difficult to design correctly because of the complexity of their behaviour, and reference passing adds another layer of complexity due to the logical structure of the system becoming dynamical. Hence, computer-aided formal verification has to be employed in the design process to ensure the correct behaviour of RPSs. However, validation of such systems is almost always limited to simulation/testing, as their formal verification is very difficult due to either the inability of the traditional verification techniques to express reference passing[1] (at least in a natural way) or by poor scalability of the existing verification techniques for RPSs.

[1] Some existing tools like SPIN allow to send channels via channels; however, they do not allow dynamic creation of new channels, which is often essential in RPSs.

This is very unfortunate: As many safety-critical systems must be resilient (and hence reconfigurable), they are often RPSs and thus have very complicated behaviour. Hence, for such systems the design errors are both very likely and very costly, and formal verification must be an essential design step. This thesis addresses this problem by developing an efficient formalism that can specify RPSs and make their formal verification feasible.

There is a number of formalisms that are suitable for specification of RPSs. The main considerations and tradeoffs in choosing an appropriate formalism are its expressiveness and the tractability of the associated verification techniques. Expressive formalisms (like π-calculus [67] and Ambient Calculus [15]) are Turing powerful and so not decidable in general. Fortunately, the ability to pass references *per se* does not lead to undecidability, and it is possible to put in place some restrictions (e.g., finiteness of the control) that would guarantee decidability, while still maintaining a reasonable modelling power.

Finite Control Processes (FCP) [20] are a fragment of π-calculus, where the system is constructed as a parallel composition of sequential entities. Each sequential entity has a finite control, and the number of such entities is bounded in advance. The entities communicate synchronously via channels, and have the possibility to create new channels dynamically and to send channels via channels.

As π-calculus is the most well-known formalism suitable for RPS specification, we fix FCPs (as a natural decidable and reasonably expressive fragment of π-calculus) as the primary RPS specification formalism, from which a new extension will be derived.

The development of this new subclass is essential because the processes constitute RPSs often have 'local' concurrency that cannot be expressed with FCP. To be more precise, an entity of an RPS can perform several instances of the same action simultaneously. This can be thought of as server in a server-client system that can serve several clients in parallel. Even if this behaviour is common, it cannot be modelled using FCP. In the literature, Meyer et al. in [65] was introduced a translation of FCP to Petri Nets in order to formally verify mobile systems. The proposed method was evaluated using various cases studies. In particular, in one of the case studies a client-server system was used. Firstly, it was modelled using FCP and then the FCP specification was converted to Petri

Nets [97]. Also, it was pointed out that the proposed technique can be applied to a wider subclass of π-calculus. For this reason, it was modelled a concurrent server and two clients. In this system when a client contacts with the server, the server spawns a new session and is ready to serve another client. Thus, several clients can be served in parallel. In this special case, even if the specification of this system is not an FCP, it rsesults to a Petri net representation.

Moreover, in distributed systems, data can be transmitted from one source to more than one destinations in a single transmission (multicasting). This kind of behaviour is a special case of 'local' concurrency and can be met, for instance, in of Internet of Things (IoT), cloud computing systems, and routing protocols in multi-core processor systems. Using FCP to model multicasting is not optimal because they are too restrictive. That happens because an FCP is a parallel composition of a finite number of sequential processes. However, with multicasting a process can contain parallel actions.

To this end, a new subclass of π-calculus is developed and introduced in this thesis to make feasible the practical modelling and formal verification of RPS, the *Extended Finite Control Processes* (EFCP). It is inspired on the observations were pointed out in [65] and on the fact that processes that constitute mobile and reconfigurable systems often have 'local' concurrency. This work is new and originally published in [50].

1.2 Diagnosability under Weak Fairness

It is often the case, during the design process of systems, faults are not possible to be ruled out. That happens because they may be part of the system or the environment where the systems evolve. Diagnosability [87] is a key specification property for formal verification of large and complex systems, because makes possible the detection of faults in a finite time after their occurrence.

The procedure of describing some abnormal behaviour of a system is called *diagnosis*. In formal verification, diagnosability is a specification property that ascertains whether it is possible the detection of fault by giving a set of observations [5]. When we are able to deduce the occurrence of a fault after observing the system's behaviour for fairly long time, we say that the system is diagnosable

[57]. On the contrary, it is often the case that we cannot conclude about the occurrence of a fault. In this case, the system is not diagnosable. Thus, more sensors should be added in the system to be possible the detection of a fault.

Recent work [44] presented a diagnosis method that encompasses *weak fairness*. There, concurrent systems are modelled by partially observable safe Petri nets, and diagnosis is carried out under the assumption that all executions of the Petri net are weakly fair, that is, the only infinite executions admitted are those in which any transition *enabled* at some stage will be *disabled* at some later stage, i.e. either it will actually fire later in that execution, or else some conflicting transition will fire. Under this assumption, a given finite observation diagnoses a fault if no finite execution yielding this observation can be extended to a weakly fair fault-free execution. The work in [44] gave a procedure for deciding this diagnosis problem. It remained open for which systems this procedure reliably diagnoses faults, i.e. how to determine whether a system is *diagnosable* under the weak fairness assumption. In Chapter 4 [36], this problem is addressed.

In this thesis, a major flaw in a previous approach to WF-diagnosability in the literature [44] is identified and corrected. In particular, based on the definition of WF-diagnosability provided in [44] is possible a system can still be not diagnosable in finite time due to the occurrence of some unrelated concurrent action as explained in detail in Chapter 4. Moreover, an efficient method for verifying WF-diagnosability based on a reduction to LTL-X model checking is presented.

1.3 Related Works

In this section, we discuss related work on formal verification of RPS systems and diagnosability.

1.3.1 Related work about formal verification of RPS

To formally verify mobile and reconfigurable systems a suitable specification formalism is a subclass of π-calculus, the FCP. There are two main approaches to verification of FCP. The first one is to directly generate the reachable state space of the model. This approach is relatively straightforward, but it has a number

of disadvantages. In particular its scalability is poor due to the complexity of the semantics restricting the use of heuristics for pruning the state space and the need to perform expensive operations (like computing the canonical form of the term) every time a new state is generated.

The alternative approach is to translate a π-calculus term into a simple formalism, e.g., Petri nets [10, 97]. The translation to PNs bridges the gap between the expressiveness and verifiability. While FCPs are suitable for modelling RPSs but difficult to verify due to the complicated semantics, PNs are a low-level formalism equipped with efficient verification techniques. This approach has a number of advantages, in particular it does not depend on a concrete verification technique, and can adapt any such technique for PNs. Furthermore, RPSs often are highly concurrent, and so translating them into a *true concurrency* formalism like PNs has a number of advantages, in particular one can efficiently utilise partial-order reductions for verification, alleviating thus the problem of combinatorial *state space explosion* [91]. That is, a small specification often has a huge number of reachable states, which is beyond the capability of existing computers.

Examples of these approaches exist in the literature. The Mobility Workbench (MWB) [93] builds the state space of a π-calculus term on the fly. The verification kit HAL [29] translates a model into a *History Dependent Automaton* [74], which is then translated into a finite automaton that can then be verified using standard methods. The technique in [51] translates the recursion-free fragment of π-calculus into high-level PNs and verifies the latter using an unfolding based technique for high-level PNs. The approach can express only finite runs, and so its practical applicability is limited. In the approach developed in [71], π-graphs (a graphical variant of π-calculus) are translated into high-level PNs. The technique works on a fragment that is equivalent to FCPs. Viet Van Pham in his thesis [73] encoded π-graphs into Petri nets. Based on this encoding it was implemented a prototype tool that simulates π-graphs models and converts them to Petri nets. The resulting Petri net models can be manipulated using the SNAKES framework [81]. This framework provides all the necessary to define and simulate most of Petri nets models.

The approach in [64] translates FCPs into safe low-level PNs, which are then verified using an unfolding based technique. The experiments [64] indicate that

this technique is much more scalable than the ones above, and it has the advantage of generating low-level rather high-level PNs. However, in the worst case the resulting PN is exponential in the size of the original FCP.

Complexity-theoretic considerations[2] suggest that a polynomial-size translation of FCPs into low-level safe PNs must exist. Such a translation and the associated software tools have been developed in [63].

However, RPSs have many kind of behaviours that cannot be expressed using FCP. To that end, a new formalism that extends FCP has been developed, the EFCP [50]. This new subclass of π-calculus is more powerful and removes expressiveness limitations of FCP. Particularly, EFCP syntax introduces replaces the prefixing operator '.' with a more powerful sequential composition ';'. To that end, Petri Box Calculus [8] also uses ';' as a sequence operator. A box is a labelled Petri net. In Petri Box calculus there are five operator boxes; choice, sequence, iteration, parallel composition and synchronisation. These operators are related to Petri nets. For instance, in the case of sequence operator a Petri net model is just an alternation of places and transitions [8]. The only similarity with our sequential composition operator stays in the symbolic representation and not behaviour. Our sequential operator is related to π-calculus and allows us to syntactically define behaviours (i.e., local concurrency) that with prefixing cannot be expressed. Moreover, we define a restriction to the proposed sequential composition (see Section 3.2). Without this restriction, during the execution of an EFCP process may be generated continuously new threads. Thus, our restriction is essential in order to avoid the well-known state space explosion problem [91]. In Petri Box calculus the sequence operator is more general and defines the sequence of places and transitions in causal way without conflicts and loops.

An EFCP process can model the whole behaviour of a RPS, like a sequence of possible actions, concurrency, multicasting etc., due to its expressiveness and powerful syntax. In an EFCP process the order of actions is important for two reasons. Firstly, the order of how the actions inside a process evolve must be preserved. This is essential if one wants to model the system's behaviour correctly. Secondly, as it is described in Section 3.2 our sequential composition has

[2] FCPs and low-level safe PNs can simulate and be simulated by a Turing machine with polynomially bounded tape.

a fundamental restriction to avoid the blow up of new generated processes that can lead to state space explosion.

One may relate the translation from EFCP to FCP with Petri net branching processes. A Petri net can model concurrent systems. The partial order semantics of a Petri net, is the set of it processes [9, 40]. A process models a possible run of a concurrent system. To that end, we can have branching processes that provide a complete relationship between different runs of the system. In this case the system has exactly on run. This branching run as introduced in [70] in the case of a Petri net is called the unfolding of the net. In a branching run, if the system has a conflict [40], the system divided into several, independent, parallel copies of itself, one for each resolution of the conflict. Thus, each copy provides a different behaviour of the system. In EFCP, now, in the translation to FCP (see Section 3.4), new processes can be generated that models parts of the EFCP process that violate the definition of FCP. Consequently, these new processes are linked with the initial process in order to preserve the order of actions as provided in the EFCP process. Thus, a similarity of EFCP with branching processes is the causal relation between each part of the system.

Moreover, EFCP makes feasible the formal verification of RPSs. This is achieved by translating EFCP to FCP. As a result, the resulting FCP specification can be converted to PN [52], for which efficient verification techniques exist.

1.3.2 Related work about diagnosability under weak fairness

In [87] is presented a formal language framework that is suitable for diagnosis and analysis of diagnosability properties of discrete event systems represented by finite automata. Diagnosis can detect a fault that has occurred in the past [7]. Diagnosis can also be applied in computer systems in order to evaluate their security policy. *Non-interference* is a security policy model that is introduced in [38, 39]. To that end, a computer can be thought of as a machine with inputs and outputs. These inputs and outputs can be categorized as either low sensitivity or high sensitive. A computer has the non-interference property iff any sequence of

low inputs will produce the same low outputs, despite of what the high level inputs are. Thus, if a low level user is working on the machine, it will respond in exactly the same manner (i.e., on the low outputs) even if a high level user is working with sensitive data. The low level user will not be able to obtain any information about the activities (if any) of the user [39]. There is a case where, at the time the computer system starts operating, if the computer has a high sensitive information within it, or low users can create some high sensitive information at a time subsequent to computer's starting time ("write-up"), then the computer can leak all that high sensitive information to the low users without violating the security policy as presented in [95]. In this case it is possible to perform diagnosis techniques in order to detect any violation of the non-interference diagnosis.

To that end, diagnosis and especially weak diagnosis can be applied in order to evaluate another security policy model, the *Bell-LaPadula* [6]. This security model does not suffer form the problem described above because it explicitly forbid "read-up". In this case, WF-diagnosability can verify whether a malicious insider modified the Bell-LaPadula security policy, or if this will happen in a future state, it can be detected.

Moreover, in [87] is introduced a technique for diagnosability verification that uses a *diagnoser*. The diagnoser is an automaton that contains only observable transitions and possible states of the system can be estimated by observing its traces. The system under consideration can be modelled as a PN, where each transition is labelled with the performed action. The actions are partitioned into *observable* and *unobservable*. Moreover, some of the unobservable actions are designated as *faults*. In [47, 90] improvements have been introduced, which were based on the *twin plant* method, where the basic idea was to build a *verifier* by constructing the synchronous product of the system with itself on observable transitions. If the original system is given as a labelled Petri net, then the verifier can be constructed directly, by synchronising the original net with its replica at the Petri net level. The verifier is used for diagnosis of the system under consideration. That is achieved by comparing every pair of executions in the system that have the same projection on the observable transitions.

Nevertheless, the main drawback of the proposed state-based twin plant method [47, 90] is the combinatorial *state space explosion* problem [91]. Meaning that,

a large number of possible states can be created from a relatively small system specification. To alleviate this issue PN unfolding techniques were applied in [43, 58]. By applying PN *unfolding* we can obtain a finite and complete prefix that represents all reachable markings of this PN. This can lead to memory savings because executions are not considered as sequences of transitions but as partially ordered sets. Initially, the PN unfolding was introduced in [61]. In [25] it was improved and in [45] parallelised. In [27] the unfolding technique was applied to distributed diagnosis and in [24] to LTL-X model checking. Moreover, in [42] diagnosability based on observable partial orders is proposed. Also, in [59] the twin plant method is applied for verifying diagnosability using PN unfoldings.

In [58] diagnosability verrfication is achieved using an existing parallel LTL-X model checker that is based on PN unfoldings [23, 89]. Diagnosability is expressed as an LTL-X property of the verifier. To formally verify it, the verifier and a Büchi automata are synchronised creating a net that accepts the negation of the diagnosability property. In [89] the LTL-X model checking proposed in [24] was parallelised and extended to high level PN. Also, the work in [89] was implemented in the PUNF tool [49]. Then, verification is performed using the unfolding-based LTL-X model checking [24]. PN unfolding is a quite efficient technique as experimental results showed in [58]. In this thesis, we consider diagnosability under the weak fairness assumption. Weak diagnosis can reveal faults that are concurrent or in the future of the observation, under the weak fairness. Based on weak diagnosis, a first definition of diagnosability under weak fairness was proposed in [1]. In this thesis, a major flaw in a previous approach to WF-diagnosability in the literature [44] is identified and corrected. Moreover, we present an efficient method for verifying WF-diagnosability based on a reduction to LTL-X model checking [35, 36].

1.4 Thesis Overview

The remainder of the thesis is organised as follows. First, the basic notions are presented in Chapter 2. This chapter explains the principles of system formal verification. In particular, it explains what model checking is. Also, the Linear Temporal Logic and Linear Temporal properties are presented. Moreover, for-

malisms like π-calculus and Petri nets are presented. Chapters 3 and 4 are based on these formalisms.

In Chapter 3, we develop an extension of FCP, the EFCP [50], which is efficient for modelling and verifying RPSs. FCP are a parallel composition of sequential threads. As, a result threads in FCP cannot have 'local' concurrency. However, this kind of behaviour is often in RPSs. EFCP removes this limitation. Thus, systems that belong to IoT, cloud computing, routing protocols of multi-core processors and generally systems that are highly concurrent and reconfigurable can be modelled and verified. Further, we provide a translation of EFCP to FCP. Then, the latter model can be translated to Petri Nets [52, 63] making the system under consideration verifiable. To evaluate this new subclass of π-calculus the SpiNNaker architecture has been used as a case study.

In Chapter 4, we consider diagnosability under the weak fairness assumption. A major flaw in a previous approach to WF-diagnosability in the literature [44] is identified and corrected. The approach in [44] contains a major flaw. It is often the case that due to the presence of some independent concurrent action in a system, it is not possible this system to be diagnosed in a finite time. Moreover, we present an efficient method for verifying WF-diagnosability based on a reduction to LTL-X model checking [35, 36].

Finally, in Chapter 5 we summarize the issues this thesis has addressed related to model checking of RPSs and generally concurrent systems, its contributions and their limitations.

1.5 List of Publications

Below is a list of publications contributing to the Thesis and the Chapters where they are relevant:

Journals:

1. V. Germanos, S. Haar, V. Khomenko and S. Schwoon: *Diagnosability under Weak Fairness. Special Issue on Best Papers from ACSD'2014, ACM Press,*

ACM Transactions on Embedded Computing Systems 14(4), Article No. 69. (Chapter 4)

2. V. Khomenko and V. Germanos: *Modelling and Analysis Mobile Systems Using π-Calculus (EFCP). In: LNCS Transactions on Petri Nets and Other Models of Concurrency X, Vol. 9410, 2015, Springer Verlag.* (Chapter 3)

Conference Papers:

1. V. Germanos, S. Haar, V. Khomenko and S. Schwoon: *Diagnosability under Weak Fairness. Proc. of ACSD'2014, Mokhov, A. and Bernardinello, L. (Eds.). IEEE Computing Society Press (2014) 132-141.* Selected as one of best papers, nominated for the best paper award. (Chapter 4)

Chapter 2

Basic Notions

In order to create tools for simulation or verification of systems, it is essential to define formalisms to describe the behaviour of a system. Ideally, such a formalism should be expressive, easy to use in practice, and amenable to automatic verification. In this chapter, principles of formal verification are explained, and the well-known standard models of concurrency, Petri nets and π-calculus, are presented.

2.1 System Verification Principles

A system under consideration can be modelled by a transition system in order to be verified according to some formally specified properties. These properties can be checked in an automated manner by model checking algorithms.

2.1.1 System Modelling

A system can be formally modelled using an appropriate formal language, like Petri nets [86, 97] or a kind of process algebras.

A *transition system* (TS) [18] is a directed graph in which nodes represent *states*, and edges depict *transitions*. States provide information about the system at a certain point of its execution and transitions show the change from one state to another.

2.1.2 Model Checking

Model checking is one of the techniques that are used for verification purposes. Algorithms that explore the state space of a given transition system are used.

However, a relatively small model can generate a very large state space, making the verification infeasible. That happens because the number of states depends on how many variables the system has and the size of their domain. This means that the growth of the state space is exponential in the number of variables. For instance, if a is the domain of y variables, then the growth rate of states is $|a|^y$. That means, if just one Boolean variable is added to the model then the size of the state space can be doubled. This is also called the *state-space explosion problem*.

This has led to the development of techniques for alleviating this problem of formal verification. A prominent technique has been introduced in [11] that applied ordered binary decision diagrams (OBDDs) [2] in order to represent implicitly transition relations of a given system. In [13, 62] McMillan verified systems that have more that 10^{20} states using OBDDs to represent transition relations. Coudert et al [19] and Pixley [75, 76, 77] have exploited OBDDs to check equivalence of deterministic finite-state machines, however, their work is not related to McMillan's. This technique has been improved in [12] where systems that have more than 10^{250} states were verified. Moreover, based on this approach the symbolic model checker [62] has been developed.

In symbolic model checking, the transition relation of the system is represented as a Boolean function and is not explicitly constructed. In the same way, Boolean functions represent sets of states. As a result in many cases, the space needed for Boolean functions is exponentially smaller compare with the one needed for explicit representation.

We recall the definition of OBDDs provided in [16]:

Definition 1. Ordered Binary Decision Diagrams. Let A be a set of propositional variables, and $\prec$ a linear order on A. An *ordered binary decision diagram* $\mathcal{O}$ over A is an acyclic graph (V, E) whose non-terminal vertices (*nodes*) are labelled by variables from A, and whose edges and terminal nodes are labelled by 0, 1. Each non-terminal node u has out-degree 2, such that one of its outgoing

edges is labelled 0 (the *low edge* or *else-edge*), and the other is labelled 1 (the *high edge* or *then-edge*). If u has label a_i and the successors of u are labelled a_j, a_k, then $a_i \prec a_j$ and $a_i \prec a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $\prec$. Each OBDD node u represents a Boolean function $\mathcal{O}_u$. The terminal nodes of $\mathcal{O}$ represent the constant functions given by their labels. A non-terminal node u with the label a_i whose successors at the high and low edges are u and w respectively, defines the function $\mathcal{O}_u := (a_i \wedge \mathcal{O}_u) \vee (\neg a_i \wedge \mathcal{O}_w)$.

OBDDs are related to decision diagrams or trees [68]. We use the example presented in [16] (see. Figure 2.1). In (a) we see the decision tree of the Boolean function $x \wedge (y \vee z)$. In (b) the same function is represented in a more concise way. Here, we should note that we obtain the OBDD by merging isomorphic subtrees and deleting redundant edges from the decision tree. In this example the variable ordering is $x \prec y \prec z$.

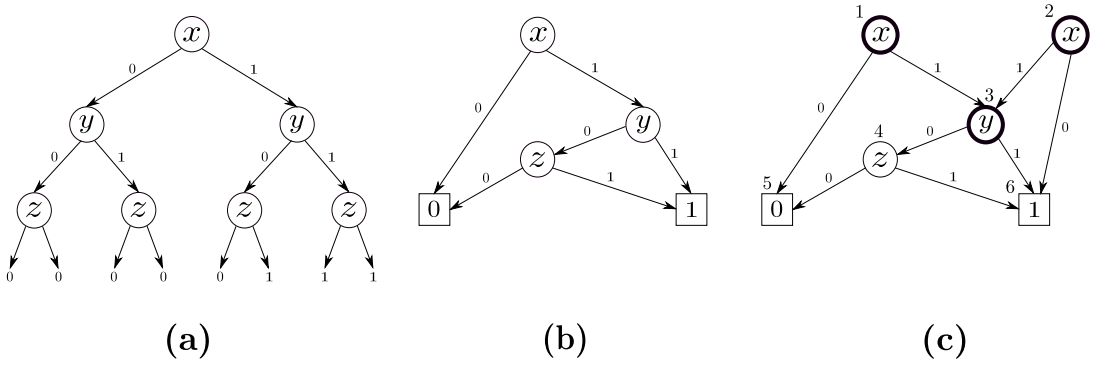

Figure 2.1: (a) Decision tree for $x \wedge (y \vee z)$, (b) OBDD for function $x \wedge (y \vee z)$, and (c) a shared OBDD.

The number of nodes of the OBDD specifies its size and depends on the variable order $\prec$. We can use *shared* OBDDs to represent several Boolean functions at once. In (c) the nodes 1, 2, 3 represent the Boolean functions $x \wedge (y \vee z)$, $\neg x \wedge (y \vee z)$, and $y \vee z$ respectively.

Other prominent techniques to alleviate the state space explosion problem are listed below [46]:

- *Induction* [46]. Often a large number of identical components exists in a

model, so verification can be conducted by induction on this number.

- *Abstraction* [17]. A system can be modelled by abstracting from unimportant features.

- *Partial order reduction* [37]. Many interleavings of component traces in asynchronous systems can be equivalent as far as satisfaction of the formula to be checked is concerned. This can significantly reduce the size of the model checking problem.

- *Composition.* This can be named as the 'divide and conquer' technique, where the verification problem is split into several simpler verification problems.

2.1.3 Linear-Time Properties

A temporal property of a system can be specified using either the state-based or the action-based model checking view. The difference between these two approaches is that in the action-based approach only action labels are used. Here, an infinite sequence of actions $a_0a_1a_2\ldots$ performed by the transition system denotes a computation. The atomic proposition a is true for a computation $a_0a_1a_2\ldots$ iff $a_0 = a$. On the other hand, in the state-based view only state labels (predicates) are taken into account and an infinite sequence of states $s_0s_1s_2\ldots$, where for each i, s_{i+1} is reachable from s_i in a single step, constitutes a possible computation. There are cases that is not possible the state-based approach to be translated into an action-based and vice versa. The reason is that we cannot always deduce from the state labels the possible enabled actions. In addition, based on the action labels it is not always possible to deduce their target state. Actions are atomic, non-persistent occurrences that happen in a particular point in time. On the other hand, states are related to things that and exist and they have a measurable value at any moment. Thus, in case we have an action-based representation then in some case states are difficult to be deduced [4].

In Chapter 4 the state-based properties are verified. Therefore, they were taken into consideration only the states' atomic propositions for formulating the

properties of the case studies. That is, because only their atomic propositions are 'visible' and a sequence of them is referred as a *trace* of the transition system.

When one has to check whether a linear-time property holds or not, it is assumed that the model under consideration does not have terminal states. Thus, before the verification it is necessary to perform deadlock checking, and eliminate any deadlocks found in the model.

In case a linear time property is violated, model checking tools provide a trace demonstrating this violation. These traces are called *counterexamples*, and are very helpful for debugging the models [46].

The first step to check whether a system satisfies some requirements is to model it using some formal language. Then, the resulting model must be verified w.r.t. the predefined requirements. These requirements are often formulated using *Linear Temporal Logic* (LTL) properties, which denote the accepted various behaviours (i.e., traces) a transition system is allowed to show.

An LTL property is defined using the atomic propositions (AP) of the model, and it characterises a subset of the infinite words over the alphabet of AP. A transition system TS satisfies an LTL property ϕ, TS $\vDash \phi$, iff each of its infinite traces satisfies ϕ. When a state, s, satisfies a LTL property ϕ, is written as, $s \vDash \phi$. That is, $s \vDash \phi$ iff all traces which start in this state respect ϕ. In the next section, LTL properties and their logic will be presented in more detail.

2.2 Linear Temporal Properties and Logic

Modelling a concurrent system is a complicated and error prone process due to synchronization errors. This can be alleviated by applying rigorous formal reasoning. LTL can describe properties of concurrent systems at any level of abstraction [30, 55]. That is, they define what behaviour a system must have and analyse what it will do [55]. In other words, a temporal logic formula expresses permissible behaviour of the system, and it is satisfied when it holds for every execution of the system. In this section, initially fairness is covered, and then the syntax of LTL formalism is presented.

2.2.1 Fairness

In concurrent systems, the behaviour when is a process is always pre-empted by another process and so can never progress is called unfair. If one wants to have *fair* behaviour, all processes eventually must be served [21] [22].

Moreover, it is very common during the modelling of concurrent systems to exclude some specific behaviours, which may be unnecessary or unrealistic. This can be achieved by applying appropriate *fairness* assumptions to the system. Moreover, these assumptions are often useful when we want to verify *liveness*, which states that eventually some progress will happen. (The use of fairness assumptions does not make sense for safety properties, as any reachable state can be reached by a finite trace, and finite traces are considered to be fair [4].) Thus, fairness plays an important role in model checking.

2.2.2 Fairness Constraints

As mentioned previously, fairness constraints are of vital importance when a realistic concurrent model needs to be designed by ruling out unrealistic executions. The most prominent notions of fairness constraints are *unconditional*, *strong*, and *weak* fairness [56] [84] [53]; the following list shows their meaning:

1. *Unconditional fairness*: Every process gets its turn infinitely many times.

   If a trace is unconditionally fair, this property holds infinitely many times.

2. *Strong fairness*: Every process that is enabled infinitely many times gets its turn infinitely many times.

   It means that if a transition is enabled infinitely many times, but not necessarily always (i.e., there may be some finite periods during which the transitions is not enabled), then it will be executed eventually.

3. *Weak fairness*: Every process that is continuously enabled from a certain time instant on gets its turn eventually.

   If a transition is continuously enabled, without periods in which the transition is not enabled, then it has to be eventually executed.

To verify whether an execution is unconditionally fair, it is enough to consider the actions that occur along the execution. Nevertheless, when we want to check whether an execution is strongly or weakly fair, the above consideration is not enough. Instead, we need to take into account the enabled transitions in all visited states as well. Note that fairness constraints are imposed on infinite traces, because any finite trace is considered to be fair [4]. Also, different fairness assumption rules can be made about different transitions.

2.2.3 Linear Temporal Logic

For checking the correctness of a system, one has to reason about the system's executions, and in the context of this thesis, about fairness issues. Formalisms which treat these aspects are temporal logics. These formalisms extend propositional or predicate logic with modalities that allow one to refer to time [4]. They can be used for describing the relative order of events. Linear Temporal Logic (LTL) was proposed by Pnueli in 1977 for specifying the properties of infinite sequences of states (or actions) that are necessary for formally verifying computer systems [80]. In LTL the occurrence of an event corresponds to the advance of a single time-unit. Therefore, the time is discrete. The present moment refers to the current state and the next moment corresponds to the immediate successor state. Thus, we assume that the system behaviour is observable at the time units 0, 1, 2, ... [4]. The following definition (Def. 1) shows the syntax of LTL. Fig. 2.2 illustrates the semantics of temporal modalities of LTL.

Definition 2. The syntax of linear temporal logic is as follows:

$$\phi ::= \top \mid \bot \mid \alpha \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (X\phi)$$

$$\mid (\Diamond\, \phi) \mid (\Box\, \phi) \mid (\phi\, \mathcal{U}\, \phi)$$

where α is an atomic proposition.

Modalities can be combined together producing new temporal modalities and describe LTL formulas as the following examples show:

$$\Box\, (p \vee q) \tag{2.1}$$

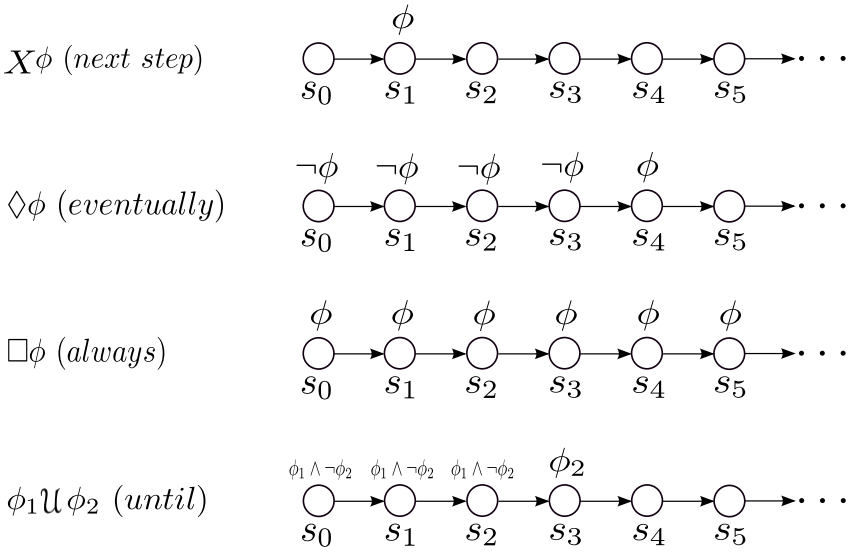

Figure 2.2: Illustration of LTL temporal connectives.

$$\Diamond \Box \phi \tag{2.2}$$

The examples 2.1 and 2.2 state that always either p or q must hold, and eventually always property ϕ must hold respectively.

An important subclass of LTL is LTL-X, which is the X-free fragment of LTL. LTL model checking is based on the automata-based approach (see subsection 2.2.4) [92]. In [55], Lamport argued that every 'sensible' LTL property must be expressible without the X operator. The 'next' is completely orthogonal to the other modalities. Meaning that, its presence does not contribute to define any of the other modalities in terms of each other. In addition, X cannot be derived from any combination of the other modalities [46].

2.2.4 Büchi Automata-based LTL Model Checking

In this section we recall notions related to LTL model checking based on Büchi automata (see [4, 34, 92] for more details), which will be used in Chapter 4.

A *Büchi automaton* is an extension of a non-deterministic finite state automaton to infinite inputs. It accepts an infinite input sequence iff some corresponding run visits any of the designated final states infinitely many times. The language of

a Büchi automaton is defined as the set of all infinite inputs that can be accepted.

In [92], it is presented an automata-based approach that formally verifies whether a system S satisfies an LTL formula ϕ. Deciding whether all computations of S satisfy ϕ is equivalent to deciding whether some computation of S satisfies $\neg\phi$. For the latter case, $\neg\phi$ has to be converted into a Büchi automaton $A_{\neg\phi}$ that accepts the computations satisfying $\neg\phi$ [34]. Afterwards, S and $A_{\neg\phi}$ are synchronised so that the language of the resulting Büchi automaton $S \times A_{\neg\phi}$ is the intersection of the language of $A_{\neg\phi}$ and the set of all possible computations of S. Thus, one can reduce the original verification problem to checking whether the language accepted by the Büchi automaton $S \times A_{\neg\phi}$ is empty, which is the case iff there is not final state that is both reachable from the initial state and lies on a cycle.

If an execution satisfying $\neg\phi$ is found, it is returned as an error trace, whereas if there is no such execution then one can conclude that $S \vDash \phi$ [4].

2.3 π-Calculus and FCP

The π-calculus [66, 88] is a formalism that can be used for modelling reconfigurable systems and reasoning about their behaviour [88]. The key idea of the formalism is that messages and the channels they are sent on have the same type: they are just *names* from some set $\Phi \stackrel{\text{df}}{=} \{a, b, x, y, i, f, r, \ldots\}$, which are the simplest entities of the π-calculus. This means a name that has been received as a message in one communication may serve as channel in a later interaction. To communicate, processes consume *prefixes* π.

Definition 3. (π-calculus) The grammar of processes and summations of π-calculus

$$P ::= M \mid P \mid P' \mid \nu r : P \mid !P$$
$$M ::= \mathbf{0} \mid \pi.P \mid M + M'$$

where

$$\pi ::= \overline{a}\langle b\rangle \;\mid\; a(x) \;\mid\; \tau. \tag{2.3}$$

The *output prefix* $\overline{a}\langle b\rangle$ sends name b along channel a. The *input prefix* $a(x)$ receives a name that replaces x on channel a. The input and output actions are called *visible actions* and prefix τ stands for a *silent action.* The following definition gives the grammar of π-calculus processes and a subclass of them, the summations. The summation form represents a process that can evolve in one of several options.

In the composition $P \mid P'$, processes P and P' can evolve independently and can interact via shared names. The replication $!P$ behaves like an infinite composition $P \mid P \mid \ldots$. The rest cases are common with the syntax of FCP, and they are explained below.

Finite Control Processes (*FCP*) [20], as described in [52], are a fragment of π-calculus, where the system consists of a parallel composition of a fixed number of sequential entities (threads). The control of each thread can be represented by a finite automaton, and the number of threads is bounded in advance. The threads communicate synchronously via channels, and have the possibility to create new channels dynamically and to send channels via channels. These capabilities are often sufficient for modelling mobile applications and instances of parameterised systems, and the appeal of FCPs is due to combining this modelling power with decidability of verification problems [20, 65].

Threads, also called *sequential processes*, are constructed as follows. A *choice process* $\sum_{i\in I} \pi_i.S_i$ over a finite set of indices I executes a prefix π_i and then behaves like S_i. The special case of choices over an empty index set $I = \emptyset$ is denoted by $\mathbf{0}$ — such a process has no behaviour. Moreover, when $|I| = 1$ we drop Σ. We use $\bigodot$ to refer to iterated prefixing, e.g. $\overline{a_1}\langle b_1\rangle.\overline{a_2}\langle b_2\rangle.\overline{a_3}\langle b_3\rangle.\overline{a_4}\langle b_4\rangle.\mathbf{0}$ can be written as $\left(\bigodot_{i=1}^{4} \overline{a_i}\langle b_i\rangle\right).\mathbf{0}$. A *restriction* $\nu r : S$ generates a name r that is different from all other names in the system. We denote a (perhaps empty) sequence of restrictions $\nu r_1 \ldots \nu r_k$ by $\nu\tilde{r}$ with $\tilde{r} = r_1 \ldots r_k$. To implement parameterised recursion, we use *calls to process identifiers* $K\lfloor\tilde{a}\rfloor$. We defer the

explanation of this construct for a moment. To sum up, threads take the form

$$S ::= K\lfloor\tilde{a}\rfloor \mid \sum_{i\in I} \pi_i.S_i \mid \nu r : S.$$

An FCP F is a parallel composition of a fixed number of threads:

$$F ::= \nu\tilde{a}.(S_{init,1} \mid \ldots \mid S_{init,n}).$$

Note that in FCPs the *parallel composition* operator $\mid$ is allowed at the top level, but not inside the threads, whereas in general π-calculus there is no such restriction. We use Π to denote iterated parallel composition, e.g. the above definition of an FCP can be re-written as $F ::= \nu\tilde{a} : \prod_{i=1}^{n} S_i$.

Our presentation of parameterised recursion using calls $K\lfloor\tilde{a}\rfloor$ follows [88]. Process identifiers K are taken from some set $\Psi \stackrel{\text{df}}{=} \{H, K, L, \ldots\}$ and have a *defining equation* $K(\tilde{f}) := S$. Here S can be understood as the implementation of identifier K. The process has a list of *formal parameters* $\tilde{f} = f_1, \ldots, f_k$ that are replaced by *factual parameters* $\tilde{a} = a_1, \ldots, a_k$ when a call $K\lfloor\tilde{a}\rfloor$ is executed. Note that both lists $\tilde{a}$ and $\tilde{f}$ have the same length. When we talk about an *FCP specification* F, we mean process F with all its defining equations.

To implement the replacement of $\tilde{f}$ by $\tilde{a}$ in calls to process identifiers, we use *substitutions*. A substitution is a function $\sigma : \Phi \to \Phi$ that maps names to names. If we make domain and codomain explicit, $\sigma : A \to B$ with $A, B \subseteq \Phi$, we require $\sigma(a) \in B$ for all $a \in A$ and $\sigma(x) = x$ for all $x \in \Phi \setminus A$. We use $\{\tilde{a}/\tilde{f}\}$ to denote the substitution $\sigma : \tilde{f} \to \tilde{a}$ with $\sigma(f_i) \stackrel{\text{df}}{=} a_i$ for $i \in \{1, \ldots, k\}$. The *application of substitution σ to S* is denoted by $S\sigma$ and defined in the standard way [88].

Input prefix $a(i)$ and restriction νr *bind* the names i and r, respectively. The *set of bound names* in a process $P = S$ or $P = F$ is $bn\,(P)$. A name which is not bound is *free*, and the *set of free names* in P is $fn\,(P)$. We permit α-conversion of bound names. The following statements provide the definition of α-conversion [88]:

- If the name w does not occur in the process P, then $P\{w/z\}$ is the process obtained by replacing each free occurrence of z in P by w.
- A change of bound names in a process P is the replacement of a subterm

$x(z).Q$ of P by $x(w).Q\{w/z\}$, or the replacement of a subterm $\nu z\ Q$ of P by $\nu : w\ Q\{w/z\}$, where in each case w does not occur in Q.

- Processes P and Q are $\alpha-$convertible, $P = Q$, if Q can be obtained from P by a finite number of changes of bound names.

The following expression is an example of α-conversion:

$$y(z).\bar{z}x.\mathbf{0} = y(w).\bar{w}x.\mathbf{0}$$

Therefore, w.l.o.g., we make the following assumptions common in π-calculus theory and collectively referred to as *no clash* **(NOCLASH)**, as introduced in [52], henceforth. For every π-calculus specification, we require that:

- a name is bound at most once;
- a name is used at most once in formal parameter lists;
- the sets of bound names, free names and formal parameters are pairwise disjoint;
- if a substitution $\sigma = \{\tilde{a}/\tilde{x}\}$ is applied to P then $bn\,(P)$ and $\tilde{a} \cup \tilde{x}$ are disjoint.

Assuming **(NOCLASH)**, the names occurring in a π-calculus specification F can be partitioned into the following sets:

$\mathcal{P}$ public names that are free in F;

$\mathcal{R}$ names bound by restriction operators;

$\mathcal{I}$ names bound by input prefixes;

$\mathcal{F}$ names used as formal parameters in defining equations.

The *size* of a π-calculus specification is defined as the size of its initial term plus the sizes of the defining equations. The corresponding function $\|\cdot\|$ measures

the number of channel names, process identifiers, the lengths of parameter lists, and the number of operators in use:

$$\begin{aligned}
\|\mathbf{0}\| &\stackrel{\text{df}}{=} 1 \\
\|K\lfloor\tilde{a}\rfloor\| &\stackrel{\text{df}}{=} 1 + |\tilde{a}| \\
\|\nu r : P\| &\stackrel{\text{df}}{=} 1 + \|P\| \\
\|K(\tilde{f}) := S\| &\stackrel{\text{df}}{=} 1 + |\tilde{f}| + \|S\| \\
\|\textstyle\sum_{i\in I} \pi_i.S_i\| &\stackrel{\text{df}}{=} 3|I| - 1 + \sum_{i\in I} \|S_i\| \\
\|\prod_{i=1}^{n} S_i\| &\stackrel{\text{df}}{=} n - 1 + \textstyle\sum_{i=1}^{n} \|S_i\|
\end{aligned}$$

It is not so simple to define reduction on terms of π-calculus, because two subterms of a process-term may interact despite the fact that they may not be adjacent. To define the behaviour of a process and the reduction on process terms, we rely on a relation called *structural congruence* $\equiv$. It is the smallest congruence where α-conversion of bound names is allowed, $+$ and $\mid$ are commutative and associative with $\mathbf{0}$ as the neutral element, and the following laws for restriction hold:

$\nu x.\mathbf{0} \equiv \mathbf{0}$

$\nu x.\nu y.P \equiv \nu y.\nu x.P$

$\nu x.(P \mid Q) \equiv P \mid (\nu x.Q)$ if $x \notin fn\,(P)$

The behaviour of π-calculus processes is determined by the *reaction relation* $\rightarrow$. The reaction relations are defined by inference rules [66, 88]:

$$\text{(Tau)}\ \tau.S + M \rightarrow S \qquad \text{(React)}\ (x(y).S + M) \mid (\overline{x}\langle z\rangle.S' + N) \rightarrow S\{z/y\} \mid S'$$

$$\text{(Res)}\ \frac{P \rightarrow P'}{\nu a.P \rightarrow \nu a.P'} \qquad \text{(Struct)}\ \frac{P \rightarrow P'}{Q \rightarrow Q'}\ \text{if } P \equiv Q \text{ and } P' \equiv Q'$$

$$\text{(Par)}\ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \text{(Const)}\ K\lfloor \tilde{a} \rfloor \to S\{\tilde{a}/\tilde{f}\} \quad \text{if } K(\tilde{f}) := S$$

The rule (Tau) is an axiom for silent steps. (React) describes the communication of two parallel threads, consuming their send and receive actions respectively and continuing as a process, where the name y is substituted by z in the receiving thread S. (Const) describes identifier calls, likewise using a substitution. The remaining rules define $\to$ to be closed under structural congruence, parallel composition and restriction. By $\mathcal{R}(F)$ we denote the set of all processes reachable from F. The *transition system* of FCP F factorises the reachable processes along structural congruence.

2.3.1 Normal form assumptions

We require that the sets of identifiers called (both directly from F and indirectly from defining equations) by different threads are disjoint. This restriction corresponds to the notion of a *safe* FCP [65] and can be achieved by replicating some defining equations. The resulting specification F' is bisimilar with F and has size $O(n\|F\|) = O(\|F\|^2)$. We illustrate the construction on the following example of an FCP specification F (left) together with its replicated version F' (right):

$$\begin{array}{ll}
K(f_1, f_2) := \tau.L(f_1, f_2) & K^1(f_1^1, f_2^1) := \tau.L^1(f_1^1, f_2^1) \\
L(f_3, f_4) := \tau.K(f_3, f_4) & L^1(f_3^1, f_4^1) := \tau.K^1(f_3^1, f_4^1) \\
 & K^2(f_1^2, f_2^2) := \tau.L^2(f_1^2, f_2^2) \\
 & L^2(f_3^2, f_4^2) := \tau.K^2(f_3^2, f_4^2) \\
 & K^3(f_1^3, f_2^3) := \tau.L^3(f_1^3, f_2^3) \\
 & L^3(f_3^3, f_4^3) := \tau.K^3(f_3^3, f_4^3) \\
K\lfloor a, b \rfloor \mid K\lfloor b, c \rfloor \mid L\lfloor a, c \rfloor & K^1\lfloor a, b \rfloor \mid K^2\lfloor b, c \rfloor \mid L^3\lfloor a, c \rfloor
\end{array}$$

Intuitively, in the resulting FCP specification each thread has its own set of defining equations. This normal form is applicable also to EFCP in Section 3.5.

2.3.2 Match and mismatch operators

The match and mismatch operators are a common extension of π-calculus [52]. Intuitively, the process $[x = y].P$ behaves as P if x and y refer to the same channel, and as $\mathbf{0}$ otherwise, and the process $[x \neq y].P$ behaves as P if x and y refer to different channels, and as $\mathbf{0}$ otherwise.

2.3.3 Expressing polyadicity

Polyadic communication can be used to make modelling more convenient. Using polyadic communication *tuples* of names can be exchanged in a single reaction. More precisely, a sending prefix $\overline{a}\langle x_1 \dots x_m \rangle$ (with $m \geq 0$) and a receiving prefix $a(y_1 \dots y_n)$ (with $n \geq 0$ and all y_i being different names) can synchronise iff $m = n$, and after synchronisation each y_i is replaced by x_i, $\{y_i/x_i\}$. Formally,

$$\text{(React)}\ (a(\tilde{y})\,;P_1 + Q_1)\,|(\overline{a}\langle\tilde{x}\rangle\,;P_2 + Q_2) \rightarrow P_1\{\tilde{x}/\tilde{y}\}\,|\,P_2\ if\ |\,\tilde{y}\,|=|\,\tilde{x}\,|$$

2.4 Petri nets

In this section, the basic definitions that describe *Petri nets* [72] are revised. *Petri nets* are a mathematical modelling language which was developed originally by Carl Adam Petri. Diagnosability under weak fairness, which is presented in Chapter 4, is based on this formal method. The account presented here will follow that in [97], [10], [96].

They have a simple graphical representation consisting of four symbols (*elements*) at the basic level. In the example shown in Figure 2.3, the elements used are circles and squares to represent *places* and *transitions*, respectively. Dots (*tokens*) are used to denote the holding of a condition at a specific point of time, places contain tokens is to said to be *marked*. Arcs indicate the flow relation between the respective circles and boxes.

Another representation method typical for nets, is the *condition/event*-nets. Systems consisting of *conditions* and *events* form the most detailed description level of marked nets. Here, conditions can, in contrast to places, carry only one token.

Definition 4. (**Petri net**) A *Petri net* is a quadruple $N = (P,T,F,M_0)$ where

- P and T are disjoint sets of places and transitions,
- F is a multiset of $F \subseteq (P \times T) \cup (T \times P)$, called the *flow relation*,
- $W : F \to (\mathbb{N} \setminus \{0\})$ is the *arc weight* mapping,
- M_0 is a non-null multiset of places, called the *initial marking*, which satisfies the restrictions:

  i. $\forall t \in T\ \exists p \in P.\ F_{p,t} > 0$ and $\forall t \in T\ \exists p \in P.\ F_{t,p} > 0$ and

  ii. $\forall p \in P.\ [M_{0b} \neq 0$ *or* $(\exists t \in T.\ F_{t,p} \neq 0)$ *or* $(\exists t \in T.\ F_{p,t} \neq 0)]$

  where a *marking* is a multiset of places, i.e. a function $M : P \to \mathbb{N} = \{0,1,2,\dots\}$ assigning a number of *tokens* to each place.

The *pre-* and *postset* of $z \in P \cup T$ are denoted by ${}^\bullet z \stackrel{\text{df}}{=} \{y \mid (y,z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z,y) \in F\}$. If $\langle \text{p,t} \rangle \in F$ for a place $p \in P$ and a transition $t \in T$, then p is an *input place* of t. Reversely, if $\langle \text{t,p} \rangle \in F$, then p is an *output place* of t.

A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if for every $p \in {}^\bullet t$, $M(p) \geq 1$. Such a transition can be *fired*, leading to a marking $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$, where '–' and '+' stand for the multiset difference and sum, respectively. We denote this by $M[t\rangle M'$. For a finite sequence of transitions $\sigma = t_1 \dots t_k$ $(k \geq 0)$ we write $M[\sigma\rangle M'$ if there are markings $M_1,\dots,M_{k+1}$ such that $M_1 = M$, $M_{k+1} = M'$ and $M_i[t_i\rangle M_{i+1}$, for $i = 1,\dots,k$. If $M = M_0$, we say σ is an *execution* of N. Analogously, infinite executions can be defined.

The set of *reachable* markings of N is the smallest (w.r.t. $\subset$) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[t\rangle M'$ for some $t \in T$ then $M' \in [M_0\rangle$. A marking M is said to be *coverable* from M_0, if there exists a marking $M' \in [M_0\rangle$ such that $M(p) \leq M'(p)$ for all $p \in P$ in N.

Figure 2.3 shows a classic Petri net model. Here, the initial marking is $p_1 \mapsto 1$, $p_2 \mapsto 1$, $p_3 \mapsto 1$, $p_4 \mapsto 0$, $p_5 \mapsto 0$. In the initial marking transition t_1 can fire. When it fires, it consumes the tokens from places p_1, p_2, and p_3 and produces tokens to places p_4 and p_5.

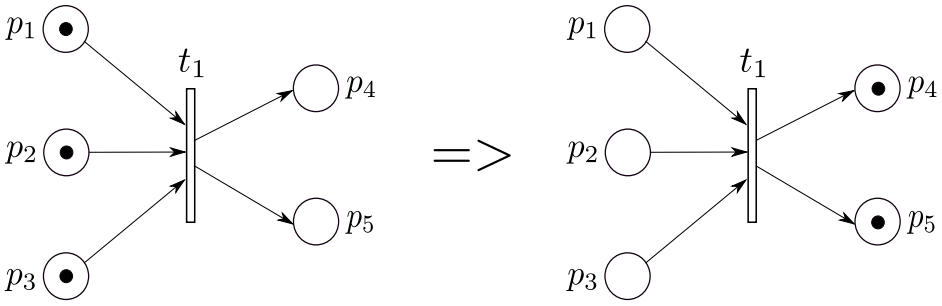

Figure 2.3: A Petri net example. Places are represented graphically as circles, transitions as squares, tokens as black dots. The transition when it fires, it consumes the tokens from places p_1, p_2, and p_3 and produces tokens to places p_4 and p_5.

A PN N is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. A marking of N is called a *deadlock* if it enables no transitions. N is *deadlock-free* if none of its reachable markings is a deadlock.

Chapter 3

Extended Finite Control Processes

3.1 Introduction

This chapter introduces *Extended Finite Control Processes* (*EFCP*), which add new features to FCPs [20], in particular limited local concurrency within a thread, while still allowing one to formally verify such systems. Thus, practical modelling of reconfigurable systems becomes more convenient. A formal description of the new formalism is presented, and an almost linear translation from EFCP to FCP, which forms the basis for formal verification of reference passing systems, is introduced in this chapter.

The threads in an FCP can communicate synchronously via channels, and are able to create new channels dynamically and send channels via channels. However, FCP threads are *sequential* processes, without any 'local' concurrency inside them. This makes FCPs too restrictive when one wants to model scenarios involving local concurrency within a thread, for instance, in case of routing protocols in multi-core processor systems. An essential feature of such protocols is the ability of a core to send a datum to several destinations concurrently.

EFCPs are sufficient for modelling many practical reconfigurable systems. Moreover, since an efficient translation from FCPs to safe Petri nets exists [52, 63], it can be reused for EFCPs (via an intermediate translation to FCPs that is

developed). Hence efficient formal verification algorithms for Petri nets can be used to verify EFCPs.

For example, the following process is an FCP, which is a parallel composition of three sequential processes (threads).

$$\begin{aligned} K_1 &:= a(x) \,.\, \overline{x}\langle b\rangle \,.\, \mathbf{0} \\ K_2 &:= \nu u : \overline{a}\langle u\rangle \,.\, \overline{w}\langle u\rangle \,.\, \mathbf{0} \\ K_3 &:= w(t) \,.\, t(v) \,.\, \mathbf{0} \\ & K_1 \,|\, K_2 \,|\, K_3 \end{aligned}$$

Note that in FCPs the parallel composition operator '|' can be used only in the initial term, i.e. the threads are fully sequential and their number is bounded in advance.

To be able to model a wide range of RPSs, a higher degree of freedom in the syntax is required to specify their various behavioural scenarios. To that end, an extension of FCPs, the EFCP, is introduced, which allows local concurrency and replaces the prefixing operator '.' with a more powerful *sequential composition* operator ';'. Also, Box Alegbra [8] uses ';' as a sequence operator, however, the only similarity with our sequential composition operator stays in the symbolic representation and not behaviour. The full EFCP syntax is defined in Section 3.2, and an example is given below.

$$\begin{aligned} K_{\mathit{1}} &:= \nu r : \big((\overline{a}\langle r\rangle \,|\, \overline{b}\langle r\rangle) \,;\, (r(x) \,|\, r(y))\big) \\ K_{\mathit{2}} &:= a(z) \,;\, \overline{z}\langle c\rangle \\ K_{\mathit{3}} &:= b(w) \,;\, \overline{w}\langle d\rangle \\ & K_{\mathit{1}} \,|\, K_{\mathit{2}} \,|\, K_{\mathit{3}} \end{aligned} \tag{3.1}$$

3.2 The syntax of Extended Finite Control Processes

To define the EFCP syntax the notion of *finite processes* is required. Such processes have special syntax ensuring that the number of actions they can execute

is bounded in advance.

The arguments of the parallel composition operator, when used inside a thread, are limited to finite processes only. Similarly, the left hand side of sequential composition must be a finite process, but the right hand side is not required to be such.

This new subcalculus is defined by a context-free grammar consisting of two sub-grammars, one for the *finite executed processes* and one for generic processes.

Definition 5 (Grammar for finite processes)**.**

$$F ::= \mathbf{0} \mid \pi \mid F + F \mid F \,|\, F \mid \nu\tilde{r} : F \mid F \,;\, F$$

where π is a prefix as described in 2.3.

The syntax of the generic processes includes that of finite processes, but also allows for extra features like recursive definitions.

Definition 6 (EFCP grammar)**.** Let F be a finite process defined above. The syntax of an EFCP thread is then

$$P ::= K\lfloor\tilde{x}\rfloor \mid F \mid P + P \mid \nu\tilde{r} : P \mid F \,;\, P$$

An EFCP specification is comprised of a set of defining equations of the form $K(\tilde{f}) := P$ and an initial term of the form $\nu\tilde{r} : \prod_{i=1}^{n} P_i$, where P and all P_i are EFCP threads.

Note that an EFCP thread cannot contain the construction $P \,|\, P$ (only the initial term can have it), but it can contain $F \,|\, F$.

3.3 Structural congruence and operational semantics

The structural congruence relation is used in the definition of the behaviour of a process term. The choice '+' and the parallel composition '|' are commutative and associative with **0** as the neutral element. Sequential composition ';' is associative with **0** as the neutral element, but not commutative.

Definition 7. Structural congruence: The structural congruence $\equiv$ is the smallest congruence that satisfies the following axioms:

Alpha Conversion:

$\nu r : P \equiv \nu r' : P\{r'/r\}$ if $r' \notin fn(P)$.

$a(x) ; P \equiv a(x') ; P\{x'/x\}$ if $x' \notin fn(P)$.

Laws for sequential composition:

$\mathbf{0} ; P \equiv P$

$F ; \mathbf{0} \equiv F$

$(F_1 ; F_2) ; P \equiv F_1 ; (F_2 ; P)$

Laws for restriction:

$\nu r : \mathbf{0} \equiv \mathbf{0}$

$(\nu\alpha)(\nu\beta) : P \equiv (\nu\beta)(\nu\alpha) : P$

Laws for parallel composition:

$F_1 \,|(F_2 \,|\, F_3) \equiv (F_1 \,|\, F_2) \,|\, F_3$

$F_1 \,|\, F_2 \equiv F_2 \,|\, F_1$

$F \,|\, \mathbf{0} \equiv F$

Laws for summation:

$P_1 + (P_2 + P_3) \equiv (P_1 + P_2) + P_3$

$P_1 + P_2 \equiv P_2 + P_1$

$P + \mathbf{0} \equiv P$

Definition 8. Structural operational semantics: The transition system of

EFCP is defined by the following rules:

$$\text{(Seq)}\ \frac{F \to F'}{F\,;Q \to F'\,;Q} \qquad \text{(Tau)}\ \tau\,;P+Q \to P$$

$$\text{(Res)}\ \frac{P \to P'}{\nu r : P \to \nu r : P'} \qquad \text{(Struct)}\ \frac{P \to P'}{Q \to Q'}\ \text{if } P \equiv Q \text{ and } P' \equiv Q'$$

$$\text{(Par)}\ \frac{P \to P'}{P \,|\, Q \to P' \,|\, Q} \qquad \text{(Const)}\ K\lfloor \tilde{a} \rfloor \to P\{\tilde{a}/\tilde{f}\}\ \text{ if } K(\tilde{f}) := P$$

$$\text{(React)}\ (x(y)\,;P_1+Q_1)\,|(\overline{x}\langle z\rangle\,;P_2+Q_2) \to P_1\{z/y\}\,|\,P_2$$

Note that these rules are similar to the π-calculus rules in Section 2.3 with the exception of the (Seq) rule expressing the semantics of our more powerful sequential composition operator ' ; '. Now, we will illustrate reduction using the example 3.1 above.

$$K_1 \,|\, K_2 \,|\, K_3$$

$$\nu r : \big((\overline{a}\langle r\rangle \,|\, \overline{b}\langle r\rangle)\,;(r(x)\,|\,r(y))\big) \,|\, a(z)\,;\overline{z}\langle c\rangle \,|\, b(w)\,;\overline{w}\langle d\rangle$$

By Res rule we have:

$$\nu r : \big((\overline{a}\langle r\rangle \,|\, \overline{b}\langle r\rangle)\,;(r(x)\,|\,r(y)) \,|\, a(z)\,;\overline{z}\langle c\rangle \,|\, b(w)\,;\overline{w}\langle d\rangle\big)$$

Then, by React and Par rules we have:

$$\nu r : ((r(x)\,|\,r(y)) \,|\, \overline{r}\langle c\rangle \,|\, \overline{r}\langle d\rangle)$$

3.4 Translation of EFCPs to FCPs

In this section, a formal description of the new formalism is presented, and an almost linear translation from safe EFCPs to safe FCPs is introduced. The purpose of translating EFCP to FCP is for the latter to be translated to safe low-level Petri nets [65], for which efficient verification techniques can be applied.

3.4.1 Description

The translation has to eliminate the parallel composition operator inside threads and the use of sequential composition. Since an FCP consists of sequential processes (*threads*), any thread of an EFCP that is not sequential must be converted to a sequential one. This can be done by shifting all the concurrency to the initial term. Moreover, sequential composition has to be replaced by prefixing. To avoid blow up in size, new declarations are introduced during this process.

To ensure that the order of actions is preserved and that the context (binding of channel names) is correct, extra communication between threads may be required. New process definitions are introduced in two cases. The first is when local concurrency exists within a thread, e.g.:

$$\begin{aligned} K[x] &:= \nu r : ((\overline{a}\langle x\rangle \mid \overline{b}\langle r\rangle)\,;\tau) \\ &K\lfloor u\rfloor \end{aligned}$$

The translation result is (see also SeqPar rule below):

$$\begin{aligned} K[x] &:= \nu r : (\overline{begin_1}\langle x\rangle \,.\, \overline{begin_2}\langle r\rangle \,.\, end_1() \,.\, end_2() \,.\, \tau \,.\, \mathbf{0}) \\ K_1 &:= begin_1(x) \,.\, \overline{a}\langle x\rangle \,.\, \overline{end_1}\langle\rangle \,.\, K_1 \\ K_2 &:= begin_2(r) \,.\, \overline{b}\langle r\rangle \,.\, \overline{end_2}\langle\rangle \,.\, K_2 \\ &K\lfloor u\rfloor \mid K_1 \mid K_2 \end{aligned}$$

Here K_1 and K_2 are fresh PIDs and $begin_1$, $begin_2$, end_1, end_2 are fresh public names. Note that the necessary context is passed to the auxiliary FCP threads K_1 and K_2 using communication on $begin_1$ and $begin_2$.

The second case is when there is a sequential composition with a non-trivial left-hand side, e.g.:

$$\begin{aligned} K[x] &:= \nu r : \big(\underbrace{(\overline{a}\langle x\rangle + \overline{b}\langle r\rangle)}_{\text{l.h.s.}}\,;\,\underbrace{(\overline{c}\langle x\rangle + \overline{d}\langle r\rangle)}_{\text{r.h.s.}}\big) \\ &K\lfloor u\rfloor \end{aligned}$$

This process is translated as follows (see also SeqChoice rule below):

$$\begin{aligned} K[x] &:= \nu r : (\overline{a}\langle x\rangle \,.\, K_1\lfloor x, r\rfloor + \overline{b}\langle r\rangle \,.\, K_1\lfloor x, r\rfloor) \\ K_1[x, r] &:= \overline{c}\langle x\rangle \,.\, \mathbf{0} \;+\; \overline{d}\langle r\rangle \,.\, \mathbf{0} \\ K\lfloor u\rfloor & \end{aligned}$$

Here K_1 is a fresh PID. Note that the initial term did not change and that the context is passed via parameters of a call.

3.5 Formal definition of EFCP to FCP translation

In this section, the translation is defined in a formal way. They are defined the translation rules that are used for translating an EFCP specification to an FCP one and the size of the translations is provided. Moreover, a example of translation to EFCP to FCP is described to facilitate the process of how these rules are applied. To evaluate our proposed theory we model the SpiNNaker architecture using EFCP, and further we verify it using a developed tool that translates the EFCP to FCP.

3.5.1 Translation

EFCP has the form

$$\begin{aligned} K_1[\tilde{x_1}] &:= P_1 \\ &\vdots \\ K_n[\tilde{x_n}] &:= P_n \\ \nu\tilde{r} : (Q_1 \,|\, \ldots \,|\, Q_k) & \end{aligned}$$

where the syntax of each P_i is given by Definitions 5 and 6, and we assume that no Q_i in the initial term uses '|' or ';'. Meaning that Q_i is now an FCP thread. To that end, the total number of the Q threads can be higher than the number of EFCP processes. That happens because new threads can be created during the

translation process as we will see later on. Note that the EFCP is assumed to be safe and to satisfy **(NOCLASH)**. Safe EFCPs are defined similarly to safe FCPs, see Sect. 2.3.1.

Definition 9. (**Translation**) The translation $[\![\cdot]\!]_B$ from EFCP to FCP is defined inductively on the syntactical structure of the EFCP. Here B is the parameter of the translation. It defines the *context*, the set of names that were bound prior to the occurrence of the term to be translated. The translation is applied to each process declaration separately:

$$[\![K[\tilde{x}] := P]\!]_{\emptyset} \stackrel{\text{df}}{=} K[\tilde{x}] := [\![P]\!]_{\tilde{x} \cap fn(P)} \tag{Decl}$$

Base cases:

$$[\![\mathbf{0}]\!]_B \stackrel{\text{df}}{=} \mathbf{0} \tag{Stop}$$

$$[\![\pi]\!]_B \stackrel{\text{df}}{=} \pi \,.\, \mathbf{0} \tag{Pref}$$

$$[\![K\lfloor\tilde{x}\rfloor]\!]_B \stackrel{\text{df}}{=} K\lfloor\tilde{x}\rfloor \tag{Call}$$

Parallel composition:

$$\left[\!\!\left[\prod_{i=1}^{k} P_i\right]\!\!\right]_B \stackrel{\text{df}}{=} \left(\bigodot_{i=1}^{k} \overline{begin_i}\langle B \cap fn\,(P_i)\rangle\right) . \bigodot_{i=1}^{k} end_i() \tag{Par}$$

where $begin_i$ and end_i are fresh public names and K_i are fresh PIDs.
$K_i := begin_i(B \cap fn\,(P_i)).[\![P_i\,;\overline{end_i}\langle\rangle\,;K_i]\!]_{B\cap fn(P_i)}, \quad i = 1 \ldots k$
$\prod_{i=1}^{k} K_i$ *is added concurrently to the initial term.*

Restriction:

$$[\![\nu\tilde{r} : P]\!]_B \stackrel{\text{df}}{=} \nu\tilde{r} : [\![P]\!]_{(B\cup\tilde{r})\cap fn(P)} \tag{Restr}$$

Choice composition:

$$\left[\!\left[\sum_{i=1}^{k} P_i\right]\!\right]_B \stackrel{\text{df}}{=} \sum_{i=1}^{k} [\![P_i]\!]_{B \cap fn(P_i)} \qquad \text{(Choice)}$$

Match and mismatch:

$$[\![[a = x] \,.\, P]\!]_B \stackrel{\text{df}}{=} [a = x] \,.\, [\![P]\!]_{B \cap fn(P)} \qquad \text{(Match)}$$

$$[\![[a \neq x] \,.\, P]\!]_B \stackrel{\text{df}}{=} [a \neq x] \,.\, [\![P]\!]_{B \cap fn(P)} \qquad \text{(Mismatch)}$$

Sequential composition base cases:

$$[\![\mathbf{0} \,;\, P]\!]_B \stackrel{\text{df}}{=} [\![P]\!]_B \qquad \text{(SeqStop)}$$

$$[\![\tau \,;\, P]\!]_B \stackrel{\text{df}}{=} \tau \,.\, [\![P]\!]_B \qquad \text{(SeqTau)}$$

$$\left[\!\left[\overline{a}\langle\tilde{b}\rangle \,;\, P\right]\!\right]_B \stackrel{\text{df}}{=} \overline{a}\langle\tilde{b}\rangle.[\![P]\!]_{B \cap fn(P)} \qquad \text{(SeqSend)}$$

$$\left[\!\left[a(\tilde{b}) \,;\, P\right]\!\right]_B \stackrel{\text{df}}{=} a(\tilde{b}).[\![P]\!]_{(B \cup \tilde{b}) \cap fn(P)} \qquad \text{(SeqRec)}$$

Sequential composition inductive cases. In Sect. 3.4.1 we provide the explanation and a number of examples of these cases:

$$\left[\!\left[\left(\sum_{i=1}^{k} P_i\right) \,;\, P\right]\!\right]_B \stackrel{\text{df}}{=} \left[\!\left[\sum_{i=1}^{k} \left(P_i \,;\, K[B \cap fn\,(P)]\right)\right]\!\right]_B \qquad \text{(SeqChoice)}$$

where K is a fresh PID (not added to the initial process)
$K[B \cap fn\,(P)] := [\![P]\!]_{B \cap fn(P)}$

$$\left[\!\left[\left(\prod_{i=1}^{k} P_i\right) ; P\right]\!\right]_B \stackrel{\text{df}}{=} \left[\!\left[\prod_{i=1}^{k} P_i\right]\!\right]_{B \cap \bigcup_{i=1}^{k} fn(P_i)} . \, [\![P]\!]_{B \cap fn(P)} \qquad \text{(SeqPar)}$$

$$[\![(\nu \tilde{r} : P) ; P']\!]_B \stackrel{\text{df}}{=} [\![\nu \tilde{r} : (P ; P')]\!]_B \qquad \text{(SeqRestr)}$$

After translating an EFCP to FCP, the resulting FCP specification it appears to be behavioural equivalent with the initial EFCP. Thus, we need to investigate the relationship between the transition systems generated by EFCPs and those generated by the corresponding FCPs, with the view to prove the correctnes of the proposed translation. In addition, the operational (rewrite rules) and denotational (translation to FCP) semantics appear to be consistent, however, they have not yet formally proved.

Moreover, as the examples in Section 3.4.1 show, the translation sifts the concurrency inside of the process to the initial term. Thus, both the EFCP and resulting FCP specification have the required level of concurrency. This is obtained by the Par rule. Based on this rule, if a process has parallelism, it generates new threads for each part that constitutes parallel composition. To that end, we need to consider that the order of the actions in the process must be preserved. Thus, new prefixes are added to link the threads with the initial term and preserve the ordering.

Another case that we need extra communication between threads to preserve the order of actions is in SeqChoice rule. In this case, this specification is not an FCP one. We have a sequential composition of a choice composition, for example of two terms (i.e., term1 and term2), in the left hand side and a process in the right hand side. Our aim is to convert this process to an FCP that has the same behaviour. In this case, either term1 or term2 will evolve, and then the process in the right hand side must evolve. For this reason, in each term of the choice composition we add a fresh PID and the process in the right hand side becomes a new thread that has the same PID with the one in the choice terms. Thus, a process that initially is not an FCP is converted to FCP and evolves in the

same way as the original EFCP specification. The SeqRestr is the same as in π-calculus. As it is explained in [88], it is an axiom that allows manipulation of term-structure. It expresses that a restriction can be moved so as to include in its scope a process in which the restricted name is not free.

3.5.2 An example of translation from EFCP to FCP

The following EFCP process models a client that communicates with a server.

$$\begin{aligned} C\lfloor url, ip\rfloor &:= \nu q : (\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor) \\ S\lfloor url'\rfloor &:= url'(ip', q') \,;\, \nu x : ((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \mid \\ &\qquad x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle) \,;\, S\lfloor url'\rfloor \\ \nu url'', ip'' &: (S\lfloor url''\rfloor \mid C\lfloor url'', ip''\rfloor) \end{aligned}$$

The server is located at some URL, $S\lfloor url'\rfloor$. A client can contact it by sending its IP address ip on the channel url. At the same time it sends a question, q, to the server, $\overline{url}\langle ip, q\rangle$. The client generates and sends a different question each time, thus q is a restricted name. The client's IP address and the question are received by the server and are stored as ip' and q', $url'(ip', q')$. The server runs two computational threads, which communicate with one another via a temporary internal channel x and produce an answer, and one of them sends the answer to the client on ip', $\nu x : ((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \mid x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle)$, at which point the server repeats its behaviour by calling $S\lfloor url'\rfloor$. The client receives the answer, $ip(a)$, and is able to contact the server again, $C\lfloor url, ip\rfloor$.

This specification is a safe EFCP satisfying **(NOCLASH)**. Now, we translate it to FCP in a stepwise manner. First, the declaration of C is translated according

to the rules of Definition 9:

$$
\begin{aligned}
&[\![C\lfloor url, ip\rfloor := \nu q : (\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor)]\!]_{\emptyset} && = \text{ by } \textit{Decl}\\
&C[url, ip] := [\![\nu q : (\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor)]\!]_{\{url,ip\}} && = \text{ by } \textit{Restr}\\
&C[url, ip] := \nu q : ([\![\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor]\!]_{\{url,ip,q\}}) && = \text{ by } \textit{SeqSend}\\
&C[url, ip] := \nu q : (\overline{url}\langle ip, q\rangle \;.\; [\![ip(a) \,;\, C\lfloor url, ip\rfloor]\!]_{\{url,ip\}}) && = \text{ by } \textit{SeqRec}\\
&C[url, ip] := \nu q : (\overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; [\![C\lfloor url, ip\rfloor]\!]_{\{url,ip\}}) && = \text{ by } \textit{Call}\\
&C[url, ip] := \nu q : \overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; C\lfloor url, ip\rfloor
\end{aligned}
$$

Finally, client process has been converted to FCP. It is now the server's turn to be translated to FCP. Again, the same procedure is followed.

$$
\begin{aligned}
[\![S\lfloor url'\rfloor \;:=\; & url'(ip', q') \,;\, \nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\emptyset} && = \text{ by } \textit{Decl}\\
S[url'] \;:=\; & [\![url'(ip', q') \,;\, \nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\{url'\}} && = \text{ by } \textit{SeqRec}\\
S[url'] \;:=\; & url'(ip', q') \;.\; [\![\nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\{url\}} && = \text{ by } \textit{SeqRestr}\\
S[url'] \;:=\; & url'(ip', q') \;.\; [\![\nu x : \big(((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle) \,;\, S\lfloor url'\rfloor\big)]\!]_{\{url\}} && = \text{ by } \textit{Restr}\\
S[url'] \;:=\; & url'(ip', q') \;.\; \nu x : [\![\big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& (x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle)\big) \,;\, S\lfloor url'\rfloor]\!]_{\{url,x\}} && = \text{ by } \textit{SeqPar}\\
S[url'] \;:=\; & url'(ip', q') \;.\; \nu x : [\![(\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle]\!]_{\{url,x\}} \;.\; [\![S\lfloor url'\rfloor]\!]_{\{url\}} && = \text{ by } \textit{Call}\\
S[url'] \;:=\; & url'(ip', q') \;.\; \nu x : [\![(\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle]\!]_{\{url,x\}} \;.\; S\lfloor url'\rfloor && = \text{ by } \textit{Par}
\end{aligned}
$$

$$S[url'] \;:=\; url'(ip', q') \;.\; \nu x : \overline{begin_1}\langle x, ip'\rangle \;.\; \overline{begin_2}\langle x\rangle \;.$$
$$end_1() \;.\; end_2() \;.\; S\lfloor url' \rfloor$$

Here $begin_1$, $begin_2$, end_1, end_2 are fresh public names, K_1 and K_2 are fresh PIDs, and $(K_1 \mid K_2,)$ is added to the initial process.

$$\begin{aligned}
K_1 \;:=\;& begin_1(x, ip') \;.\; [\![\nu r : \overline{x}\langle r\rangle \;;\; \tau \;;\; r(a) \;;\\
& \overline{ip'}\langle a\rangle \;;\; \overline{end_1}\langle\rangle \;;\; K_1]\!]_{\{ip', x\}} && = \text{ by } \textit{Restr}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : [\![\overline{x}\langle r\rangle \;;\; \tau \;;\; r(a) \;;\\
& \overline{ip'}\langle a\rangle \;;\; \overline{end_1}\langle\rangle \;;\; K_1]\!]_{\{ip', x, r\}} && = \text{ by } \textit{SeqSend}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; [\![\tau \;;\; r(a) \;;\\
& \overline{ip'}\langle a\rangle \;;\; \overline{end_1}\langle\rangle \;;\; K_1]\!]_{\{ip', r\}} && = \text{ by } \textit{SeqTau}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; [\![r(a) \;;\\
& \overline{ip'}\langle a\rangle \;;\; \overline{end_1}\langle\rangle \;;\; K_1]\!]_{\{ip', r\}} && = \text{ by } \textit{SeqRec}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\\
& [\![\overline{ip'}\langle a\rangle \;;\; \overline{end_1}\langle\rangle \;;\; K_1]\!]_{\{ip', a\}} && = \text{ by } \textit{SeqSend}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\\
& \overline{ip'}\langle a\rangle \;.\; [\![\overline{end_1}\langle\rangle \;;\; K_1]\!]_{\emptyset} && = \text{ by } \textit{SeqSend}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\\
& \overline{ip'}\langle a\rangle \;.\; \overline{end_1}\langle\rangle \;.\; [\![K_1]\!]_{\emptyset} && = \text{ by } \textit{Call}\\
K_1 \;:=\;& begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\\
& \overline{ip'}\langle a\rangle \;.\; \overline{end_1}\langle\rangle \;.\; K_1
\end{aligned}$$

$$
\begin{array}{lcll}
K_2 & := & begin_2(x) \,.\, [\![x(v)\,;(\tau \;+\; \tau)\,; & \\
 & & \overline{v}\langle a'\rangle\,;\overline{end_2}\langle\rangle\,;K_2]\!]_{\{x\}} & = \text{ by } \textit{SeqRec} \\
K_2 & := & begin_2(x) \,.\, x(v) \,.\, [\![(\tau \;+\; \tau)\,; & \\
 & & \overline{v}\langle a'\rangle\,;\overline{end_2}\langle\rangle\,;K_2]\!]_{\{v\}} & = \text{ by } \textit{SeqChoice} \\
K_2 & := & begin_2(x) \,.\, x(v) \,.\, [\![(\tau\,;K_3\lfloor v\rfloor \;+ & \\
 & & \tau\,;K_3\lfloor v\rfloor)]\!]_{\{v\}} & = \text{ by } \textit{Choice}
\end{array}
$$

Here K_3 is a fresh PID (not added to the initial process)

$$
\begin{array}{lcll}
K_2 & := & begin_2(x) \,.\, x(v) \,.\, ([\![\tau\,;K_3\lfloor v\rfloor]\!]_{\{v\}} \;+ & \\
 & & [\![\tau\,;K_3\lfloor v\rfloor]\!]_{\{v\}}) & = \text{ by } \textit{SeqTau} \\
K_2 & := & begin_2(x) \,.\, x(v) \,.\, (\tau \,.\, [\![K_3\lfloor v\rfloor]\!]_{\{v\}} \;+ & \\
 & & \tau \,.\, [\![K_3\lfloor v\rfloor]\!]_{\{v\}}) & = \text{ by } \textit{Call} \\
K_2 & := & begin_2(x) \,.\, x(v) \,.\, (\tau \,.\, K_3\lfloor v\rfloor \;+\; \tau \,.\, K_3\lfloor v\rfloor) &
\end{array}
$$

$$
\begin{array}{lcll}
K_3[v] & := & [\![\overline{v}\langle a'\rangle\,;\overline{end_2}\langle\rangle\,;K_2]\!]_{\{v\}} & = \text{ by } \textit{SeqSend} \\
K_3[v] & := & \overline{v}\langle a'\rangle \,.\, [\![\overline{end_2}\langle\rangle\,;K_2]\!]_{\emptyset} & = \text{ by } \textit{SeqSend} \\
K_3[v] & := & \overline{v}\langle a'\rangle \,.\, \overline{end_2}\langle\rangle \,.\, [\![K_2]\!]_{\emptyset} & = \text{ by } \textit{Call} \\
K_3[v] & := & \overline{v}\langle a'\rangle \,.\, \overline{end_2}\langle\rangle \,.\, K_2 &
\end{array}
$$

Finally, the resulting FCP is:

$$
\begin{aligned}
C[url, ip] &:= \nu q : \overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; C\lfloor url, ip\rfloor \\
S[url'] &:= url'(ip', q') \;.\; \nu x : \overline{begin_1}\langle x, ip'\rangle \;.\; \overline{begin_2}\langle x\rangle \;.\; end_1() \;.\; end_2() \;.\; S\lfloor url'\rfloor \\
K_1 &:= begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\; \overline{ip'}\langle a\rangle \;.\; \overline{end_1}\langle\rangle \;.\; K_1 \\
K_2 &:= begin_2(x) \;.\; x(v) \;.\; (\tau \;.\; K_3\lfloor v\rfloor \;+\; \tau \;.\; K_3\lfloor v\rfloor) \\
K_3[v] &:= \overline{v}\langle a'\rangle \;.\; \overline{end_2}\langle\rangle \;.\; K_2 \\
&\qquad \nu url'', ip'' : (S\lfloor url''\rfloor \mid C\lfloor url'', ip''\rfloor \mid K_1 \mid K_2)
\end{aligned}
$$

3.5.3 Size of the translation

One can easily check that every translation rule except (SeqChoice) yields a linear size result, and that (SeqChoice) yields at most quadratic result. This quadratic blow-up happens when it is necessary to pass a large number of bound names as parameters of a call, as shown in the following example.

$$
\begin{aligned}
K \quad &:= \quad a(\tilde{x}) \,;\, \left(\sum_{i=1}^{N} \tau\right) \,;\, \overline{b}\langle\tilde{x}\rangle \\
&K
\end{aligned}
$$

The translated process is:

$$
\begin{aligned}
K \quad &:= \quad a(\tilde{x}).\left(\sum_{i=1}^{N} \tau.K_1\lfloor\tilde{x}\rfloor\right) \\
K_1[\tilde{x}] \quad &:= \quad \overline{b}\langle\tilde{x}\rangle \;.\; \mathbf{0} \\
&K
\end{aligned}
$$

If $|\tilde{x}| = N$ then the size of the translated specification is quadratic, as N calls with N parameters each are created.

Note that this quadratic blow-up in (SeqChoice) is isolated, and the subsequent translation of these calls by the (Call) rule cannot create any further blow-up, and so the overall size of the translated process is at most quadratic.

Furthermore, one needs a rather artificial process for this quadratic blow-up to occur, and we conjecture that for practical EFCP models the translation will usually be linear.

3.6 Case study

In this section, the applicability of the proposed formalism and its translation to safe FCP is demonstrated using SpiNNaker [32] as a case study.

3.6.1 SpiNNaker architecture

SpiNNaker is a massively parallel architecture designed to model large-scale spiking neural networks in real-time [69]. Its design is based around *ad-hoc* multi-core System-on-Chips, which are interconnected using a two-dimensional toroidal triangular mesh [14, 69]. Neurons are modelled in software and their spikes generate packets that propagate through the on- and inter-chip communication fabric relying on custom-made on-chip multicast routers [33, 78]. The aim of SpiNNaker project is to simulate a billion spiking neurons in real time [3, 48, 85]. The SpiNNaker architecture is illustrated in Figure 3.1.

Every node of the network consists of a SpiNNaker Chip multiprocessor (CMP), which constitutes the basis of the system [28, 79]. It comprises 20 processing cores and SDRAM memory. For the cores, synchronous ARM9 processors were used because of their high power efficiency [85]. One of the processors is called monitor processor and its role is to perform system management tasks and to allow the user to track the on-chip activity. The other processors run independent event-driven neural processes and each of them simulates a group of neurons. Each processor core models up to around one thousand individual neurons.

The communication network-on-chip (NoC) provides an on- and off- chip packet switching infrastructure [98], see Figure 3.2. Its main task is to carry neural-event packets between the processors that can be located on the same or different chips. Also, it transports system configurations and monitoring information [79, 98]. The receiver of the data must be able to manage how long the sender keeps the data stable in order to complete a Delay-Insensitive communica-

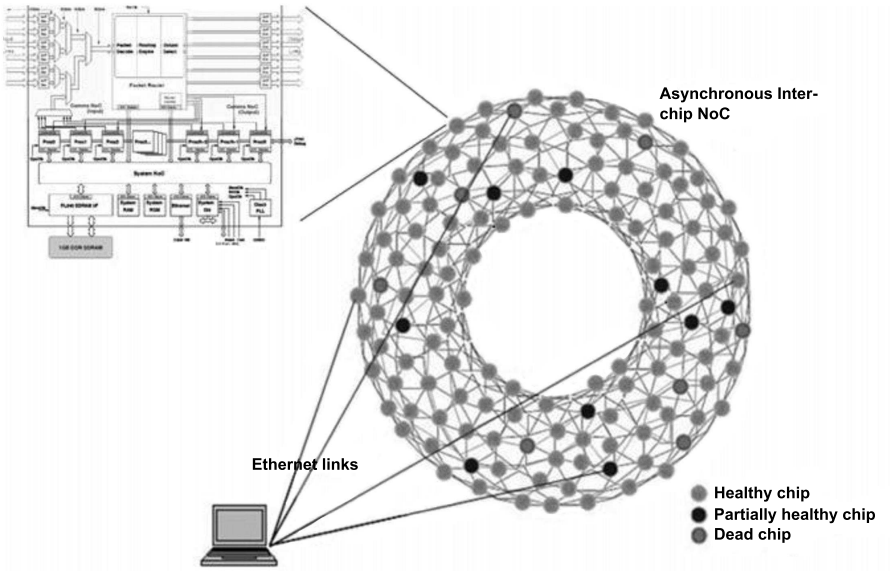

Figure 3.1: The SpiNNaker architecture [31].

tion. This is achieved by handshaking. The receiver uses an acknowledgement to show that data has been accepted. The acknowledgement follows a return-to-zero protocol [79, 98].

Figure 3.3 illustrates a SpiNNaker system composed of 25 SpiNNaker chips at a high level of abstraction. They are linked with each other by channels (e.g., *c1*, *c2*, ...). According to the routing protocol [98] of SpiNNaker's system, every chip can generate and propagate a datum. Every chip is connected to six other chips by bidirectional links as shown in Figure 3.3. This structure forms a Cartesian coordinate system. For instance, P_0 can communicate only with P_1, P_6, P_5, P_4, P_{24} and P_{20}. Thus, the communication happens in the first and third quadrant. Every chip has a pair of coordinates. These coordinates are needed for the routing plan of the system. It is possible for some chips to be faulty or congested. In such a case, an emergency routing plan is followed to bypass this kind of issues [82, 83]. Thus, the redundancy of the SpiNNaker chips enhances the fault tolerance of the system [41].

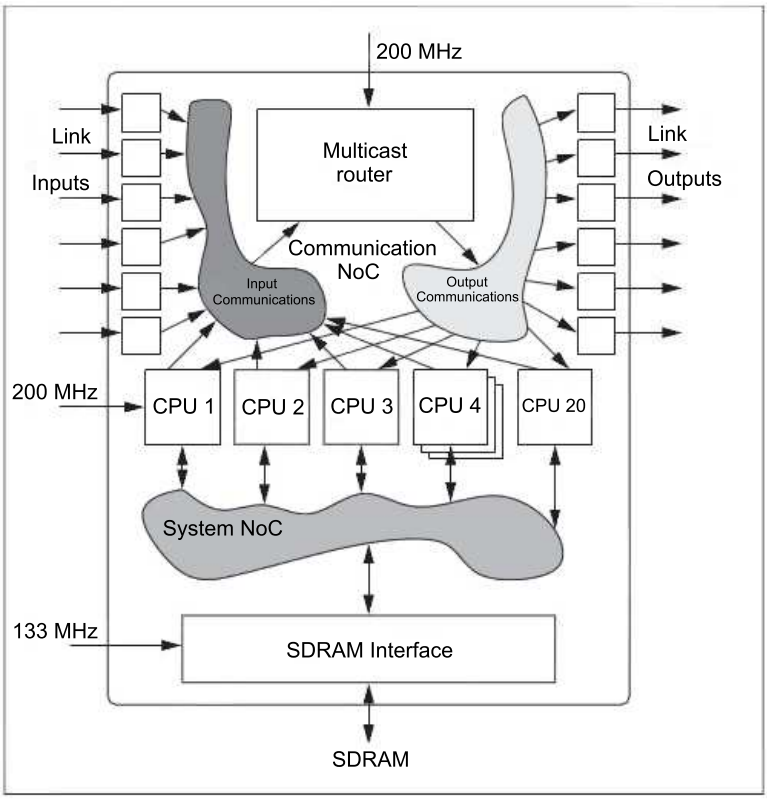

Figure 3.2: The SpiNNaker chip organization [98].

3.6.2 Modelling SpiNNaker interconnection network

The flow-control mechanism of the interconnection network (IN) of SpiNNaker is as follows. When a packet arrives to an input port, one or more output ports are selected, and the router tries to transmit the packet through them. If the packet cannot be forwarded, the router will keep trying, and after a given period of time it will also test the clockwise emergency route. It will try both the regular and the emergency route. Finally, if a packet stays in the router for longer than a given threshold (waiting time), the packet will be dropped to avoid deadlocks. To avoid livelocks, packets have an age field in their header. When two ages pass and the packet is still in the IN, it is considered outdated and dropped [69].

The following EFCP models a 5×5 SpiNNaker configuration. A healthy processor, *HP*, can execute either of the following scenarios:

- It can generate a new message, m, and process it by calling an auxiliary

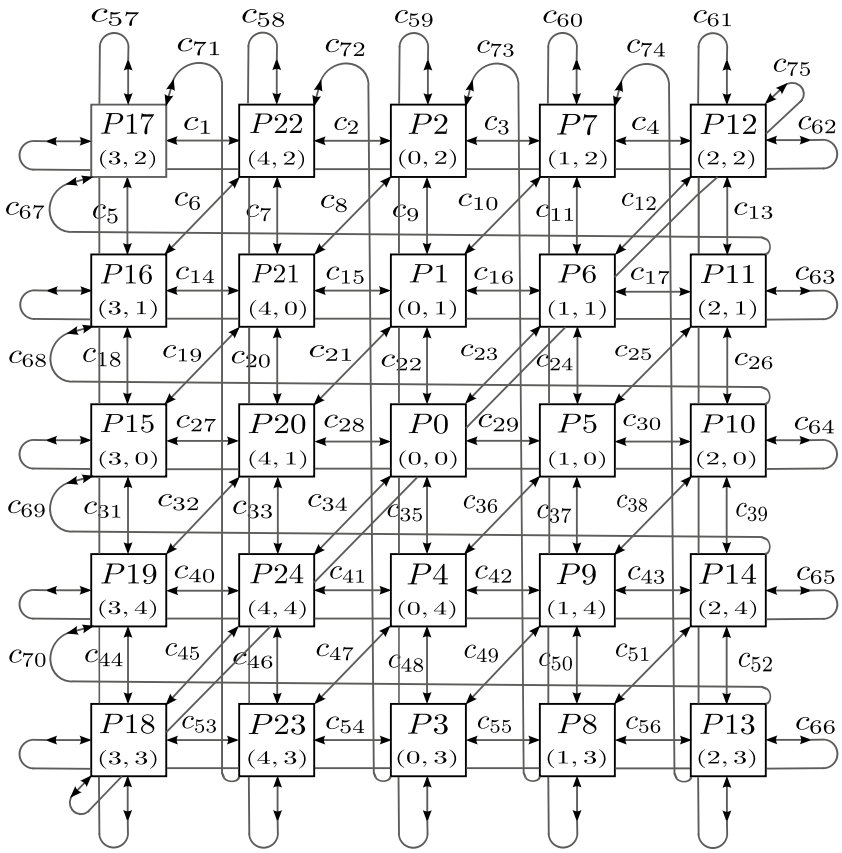

Figure 3.3: SpiNNaker network topology [98].

declaration *MSEND*.

- It can receive a message on any of its channels and process it using an auxiliary declaration *REC_MSEND*.
- It can become permanently faulty by calling an auxiliary declaration *FP*.

The definition of *HP* has six formal parameters corresponding to the six channels connecting it to the neighbours, see Fig. 3.3. These parameters are named after

points of the compass, e.g. 'n' stands for 'north', 'ne' stands for 'north-east', etc.

$$\begin{aligned} HP[n,ne,e,s,sw,w] \;:=\; & \nu m : MSEND\lfloor m,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor n,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor ne,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor e,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor s,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor sw,n,ne,e,s,sw,w\rfloor \;+ \\ & REC\_MSEND\lfloor w,n,ne,e,s,sw,w\rfloor \;+ \\ & FP\lfloor n,ne,e,s,sw,w\rfloor \end{aligned}$$

The auxiliary declarations are as follows:

MSEND$[m,n,ne,e,s,sw,w]$ sends message m on 0 or more of the channels and becomes *HP*$[n,ne,e,s,sw,w]$. In particular, the message can be consumed, forwarded or multicast. Clockwise emergency routes are used in case of negative acknowledgement *nack*.

$$\begin{aligned} MSEND[m,n,ne,e,s,sw,w] := \Big(& \\ & \Big(\tau \;+\; \overline{n}\langle m\rangle \,;\, n(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{ne}\langle m\rangle \,;\, ne(a)\big)\Big) \mid \\ & \Big(\tau \;+\; \overline{ne}\langle m\rangle \,;\, ne(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{e}\langle m\rangle \,;\, e(a)\big)\Big) \mid \\ & \Big(\tau \;+\; \overline{e}\langle m\rangle \,;\, e(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{s}\langle m\rangle \,;\, s(a)\big)\Big) \mid \\ & \Big(\tau \;+\; \overline{s}\langle m\rangle \,;\, s(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{sw}\langle m\rangle \,;\, sw(a)\big)\Big) \mid \\ & \Big(\tau \;+\; \overline{sw}\langle m\rangle \,;\, sw(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{w}\langle m\rangle \,;\, w(a)\big)\Big) \mid \\ & \Big(\tau \;+\; \overline{w}\langle m\rangle \,;\, w(a) \,;\, \big([a=ack]\,;\tau \;+\; [a=nack]\,;\overline{n}\langle m\rangle \,;\, n(a)\big)\Big) \\ \Big) \,;\, HP\lfloor n,ne,e,s,sw,w\rfloor & \end{aligned}$$

REC_MSEND$[c,n,ne,e,s,sw,w]$ receives a message on channel c and either

negatively acknowledges (*nack*) it to simulate congestion or positively acknowledges (*ack*) it and then consumes, forwards or multicasts it by calling *MSEND*.

$$REC\_MSEND[c,n,ne,e,s,sw,w] := c(m)\,;\,(\\ \overline{c}\langle nack\rangle\,;\,HP\lfloor n,ne,e,s,sw,w\rfloor + \overline{c}\langle ack\rangle\,;\,MSEND\lfloor m,n,ne,e,s,sw,w\rfloor\\ )$$

$FP[n,ne,e,s,sw,w]$ models a faulty process that does not send any messages and negatively acknowledges (*nack*) all the received messages.

$$FP[n,ne,e,s,sw,w] := (\\ n(m)\,;\,\overline{n}\langle nack\rangle + ne(m)\,;\,\overline{ne}\langle nack\rangle + e(m)\,;\,\overline{e}\langle nack\rangle +\\ s(m)\,;\,\overline{s}\langle nack\rangle + sw(m)\,;\,\overline{sw}\langle nack\rangle + w(m)\,;\,\overline{w}\langle nack\rangle\\ )\,;\,FP\lfloor n,ne,e,s,sw,w\rfloor$$

The initial term creates 25 concurrent instances of HP, $\prod_{i=1}^{25} HP\lfloor\ldots\rfloor$, and connects them by channels as shown in Figure 3.3:

$$HP\lfloor c_{22},c_{23},c_{29},c_{35},c_{34},c_{28}\rfloor \mid HP\lfloor c_9,c_{10},c_{16},c_{22},c_{21},c_{15}\rfloor \mid\\ HP\lfloor c_{59},c_{73},c_3,c_9,c_8,c_2\rfloor \mid HP\lfloor c_{48},c_{49},c_{55},c_{59},c_{72},c_{54}\rfloor \mid\\ HP\lfloor c_{35},c_{36},c_{42},c_{48},c_{47},c_{41}\rfloor \mid HP\lfloor c_{24},c_{25},c_{30},c_{37},c_{36},c_{29}\rfloor \mid\\ HP\lfloor c_{11},c_{12},c_{17},c_{24},c_{23},c_{16}\rfloor \mid HP\lfloor c_{60},c_{74},c_4,c_{11},c_{10},c_3\rfloor \mid\\ HP\lfloor c_{50},c_{51},c_{56},c_{60},c_{73},c_{55}\rfloor \mid HP\lfloor c_{37},c_{38},c_{43},c_{50},c_{49},c_{42}\rfloor \mid\\ HP\lfloor c_{26},c_{68},c_{64},c_{39},c_{38},c_{30}\rfloor \mid HP\lfloor c_{13},c_{67},c_{63},c_{26},c_{25},c_{17}\rfloor \mid\\ HP\lfloor c_{61},c_{75},c_{62},c_{13},c_{12},c_4\rfloor \mid HP\lfloor c_{52},c_{70},c_{66},c_{61},c_{74},c_{56}\rfloor \mid\\ HP\lfloor c_{39},c_{69},c_{65},c_{52},c_{51},c_{43}\rfloor \mid HP\lfloor c_{18},c_{19},c_{27},c_{31},c_{69},c_{64}\rfloor \mid$$

$$
\begin{aligned}
&HP\lfloor c_5, c_6, c_{14}, c_{18}, c_{68}, c_{63}\rfloor \mid HP\lfloor c_{57}, c_{71}, c_1, c_5, c_{67}, c_{62}\rfloor \mid \\
&HP\lfloor c_{44}, c_{45}, c_{53}, c_{57}, c_{75}, c_{66}\rfloor \mid HP\lfloor c_{31}, c_{32}, c_{40}, c_{44}, c_{70}, c_{65}\rfloor \mid \\
&HP\lfloor c_{20}, c_{21}, c_{28}, c_{33}, c_{32}, c_{27}\rfloor \mid HP\lfloor c_7, c_8, c_{15}, c_{20}, c_{19}, c_{14}\rfloor \mid \\
&HP\lfloor c_{58}, c_{72}, c_2, c_7, c_6, c_1\rfloor \mid HP\lfloor c_{46}, c_{47}, c_{54}, c_{58}, c_{71}, c_{53}\rfloor \mid \\
&HP\lfloor c_{33}, c_{34}, c_{41}, c_{46}, c_{45}, c_{40}\rfloor
\end{aligned}
$$

The above specification is an EFCP and below its translation to FCP is given. It has been obtained with the help of the developed tool EFCP2FCP. First of all, this EFCP must be translated to a safe EFCP. This is done automatically by the tool by replicating the process declarations, HP, $MSEND$, REC_MSEND, and FP, so that each of the 25 threads has its own copies of these declarations: HP^i, $MSEND^i$, REC_MSEND^i, FP^i, $i = 1 \ldots 25$. Also, the tool enforces the **(NOCLASH)** assumptions by renaming the formal parameters and bound names. However, below we disregard this renaming for the sake of clarity.

The translation of HP and REC_MSEND is straightforward as they do not use any special features of EFCP:

$$
\begin{aligned}
HP^i[n, ne, e, s, sw, w] \;:=\; & \nu m : MSEND^i\lfloor m, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor n, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor ne, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor e, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor s, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor sw, n, ne, e, s, sw, w\rfloor + \\
& REC\_MSEND^i\lfloor w, n, ne, e, s, sw, w\rfloor + \\
& FP^i\lfloor n, ne, e, s, sw, w\rfloor
\end{aligned}
$$

$$REC\_MSEND^i[c, n, ne, e, s, sw, w] := c(m)\ .\ (\\ \overline{c}\langle nack\rangle\ .\ HP^i\lfloor n, ne, e, s, sw, w\rfloor + \overline{c}\langle ack\rangle\ .\ MSEND^i\lfloor m, n, ne, e, s, sw, w\rfloor\\ )$$

The translation of FP can be obtained by applying the (SeqChoice) rule:

$$FP^i[n, ne, e, s, sw, w] :=\\ n(m).\overline{n}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor + ne(m).\overline{ne}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor +\\ e(m).\overline{e}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor + s(m).\overline{s}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor +\\ sw(m).\overline{sw}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor + w(m).\overline{w}\langle nack\rangle.K_{FP^i}\lfloor n, ne, e, s, sw, w\rfloor$$

$$K_{FP^i}[n, ne, e, s, sw, w] := FP^i\lfloor n, ne, e, s, sw, w\rfloor$$

Here K_{FP^i} is a fresh PID.

The translation of $MSEND$ is the most interesting one as it contains some local concurrency that is not allowed in FCPs (below K^i_j, L^i_j and M^i_j are fresh PIDs and $begin^i_j$ and end^i_j are fresh public names):

$$MSEND^i[m, n, ne, e, s, sw, w] :=\\ \overline{begin_1}\langle m, n, ne\rangle.\overline{begin_2}\langle m, ne, e\rangle.\overline{begin_3}\langle m, e, s\rangle.\\ \overline{begin_4}\langle m, s, sw\rangle.\overline{begin_5}\langle m, sw, w\rangle.\overline{begin_6}\langle m, w, n\rangle.\\ end_1().end_2().end_3().end_4().end_5().end_6().HP^i\lfloor n, ne, e, s, sw, w\rfloor$$

$$K^i_1 := begin_1(m, n, ne).\big(\\ \tau.L^i_1 + \overline{n}\langle m\rangle.n(a).([a = ack].\tau.M^i_1 + [a = nack].\overline{ne}\langle m\rangle.ne(a).M^i_1)\\ \big)$$

$$L^i_1 := \overline{end_1}\langle\rangle\ .\ K^i_1$$

$$M_1^i := L_1^i$$

$$K_2^i := begin_2(m, ne, e).\big($$
$$\tau.L_2^i + \overline{ne}\langle m\rangle.ne(a).([a = ack].\tau.M_2^i + [a = nack].\overline{e}\langle m\rangle.e(a).M_2^i)$$
$$\big)$$

$$L_2^i := \overline{end_2}\langle\rangle \;.\; K_2^i$$

$$M_2^i := L_2^i$$

$$K_3^i := begin_3(m, e, s).\big($$
$$\tau.L_3^i + \overline{e}\langle m\rangle.e(a).([a = ack].\tau.M_3^i + [a = nack].\overline{s}\langle m\rangle.s(a).M_3^i)$$
$$\big)$$

$$L_3^i := \overline{end_3}\langle\rangle \;.\; K_3^i$$

$$M_3^i := L_3^i$$

$$K_4^i := begin_4(m, s, sw).\big($$
$$\tau.L_4^i + \overline{s}\langle m\rangle.s(a).([a = ack].\tau.M_4^i + [a = nack].\overline{sw}\langle m\rangle.sw(a).M_4^i)$$
$$\big)$$

$$L_4^i := \overline{end_4}\langle\rangle \;.\; K_4^i$$

$$M_4^i := L_4^i$$

$$K_5^i := begin_5(m, sw, w).\big($$
$$\tau.L_5^i + \overline{sw}\langle m\rangle.sw(a).([a = ack].\tau.M_5^i + [a = nack].\overline{w}\langle m\rangle.w(a).M_5^i)$$
$$\big)$$

$$L_5^i := \overline{end_5}\langle\rangle \;.\; K_5^i$$

$$M_5^i := L_5^i$$

$$K_6^i := begin_6(m, w, n).\big($$
$$\tau + \overline{w}\langle m\rangle.L_6^i.w(a).([a = ack].\tau.M_6^i + [a = nack].\overline{n}\langle m\rangle.n(a).M_6^i)$$
$$\big)$$

$L_6^i := \overline{end_6}\langle\rangle \ . \ K_6^i$

$M_6^i := L_6^i$

The initial process is now as follows:

$$\prod_{i=1}^{25} HP^i\lfloor\ldots\rfloor \quad | \quad \prod_{i=1}^{25}\prod_{j=1}^{6} K_j^i$$

3.6.3 A tool implementation to automate the translation from EFCP to FCP

For the proposed translation the Efcp2Fcp tool has been developed to automate this process. It is a command line tool that can run in a Windows machine and has been implemented in C++. Efcp2Fcp has been developed inside the Punf platform. The following Figure. 3.4 show its architecture. It consists of 5 classes and 3 header files which contains 40 functions and the total lines of code is almost 3500.

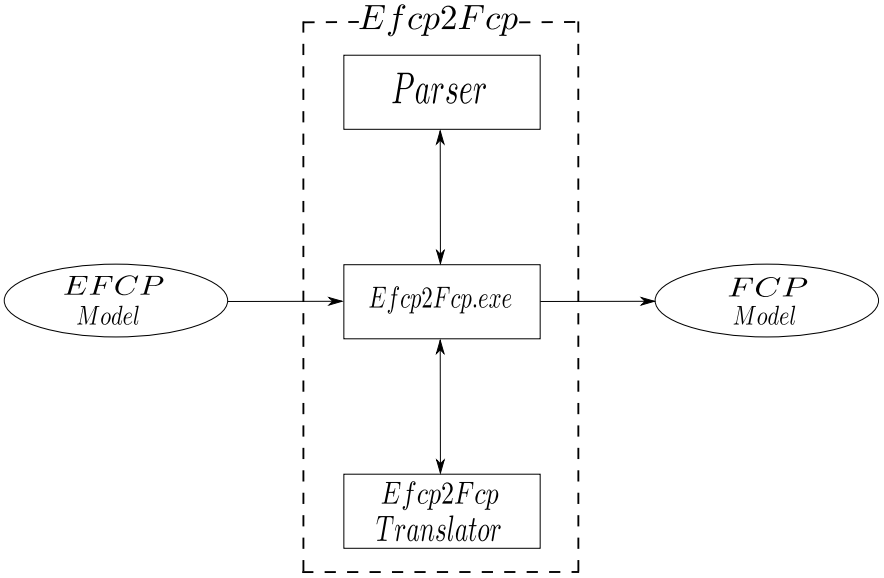

Figure 3.4: Efcp2Fcp Architecture.

Efcp2Fcp takes as an input a text file that contains an EFCP model. Then, a parser is used in order to check that whether the model is syntactically correct.

The parser has been implemented using the Bison parser generator. That was one of the most important steps of whole development because any error in the syntax we defined in the parser, leads to a faulty FCP model in the output. When the parser verifies that syntax of the EFCP specification is correct, the function that translates the EFCP input file to FCP is called. This function recursively uses the rules of EFCP to FCP translation (see Section 3.5) and converts the EFCP specification to an FCP one. When this process is finished, the output of this function (the FCP model) is stored in new text file. In the early stages of the development, the number of translation rules was less than the one is introduced in this thesis. Thus, the approach of developing the translation function in a recursive way made the implementation easier when new rules or modification in the existing rules had to be added or made.

During the testing, the software have stable behaviour. However, it has not been tested in large EFCP specification that consists of a big number of process to evaluate further its performance and stability. Moreover, because of its recursive functionality the tool can be extended, and new translation rules can be added if it is necessary based on future research.

3.6.4 Formal verification of SpiNNaker architecture

As outlined in the introduction, formal verification is an important motivation of this paper. It was performed as follows. First, the EFCP model of the 2x2 SpiNNaker network was automatically translated into an FCP model by the EFCP2FCP tool. Then the resulting FCP was then translated into a safe low-level Petri net using the FCP2PN tool [52]. Some small adaptations had to be done for the latter: FCP2PN requires choices to be *guarded,* i.e. each summand must start with a prefix, match or mismatch. This was achieved by inlining the calls to REC_MSEND^i and prefixing the first and last summands in the body of HP^i with τ. We also inlined the calls to L^i_j and M^i_j as an optimisation – the same effect could have been achieved automatically during the translation if rule (SeqChoice) were avoiding the creation of a new PID whenever the size of P does not exceed some pre-defined constant. The translation runtimes were negligible (<2sec) in both cases, and the resulting Petri net contained 14844 places, 38864

transitions and 292336 arcs.

Then deadlock checking was performed with the LoLA tool,[1] configured to assume safeness of the Petri net (`CAPACITY 1`), use the stubborn sets and symmetry reductions (`STUBBORN`, `SYMMETRY`), compress states using P-invariants (`PREDUCTION`), use a light-weight data structure for states (`SMALLSTATE`), and check for deadlocks (`DEADLOCK`).

The verification runtime was 3223sec, and LoLA reported that the model had a deadlock. In hindsight, this is quite obvious, as the model allows all the processors to become faulty, after which they stop generating new messages and the system quickly reaches a deadlock state.

3.6.5 Conclusion

The initial motivation of this research was the development of a formalism allowing for convenient modelling and formal verification of Reference Passing Systems. To that end, a new fragment of π-calculus, the Extended Finite Control Processes, is presented in this thesis. EFCPs is an extension of the well-known fragment of π-calculus, the Finite Control Processes. FCPs were used for formal modelling of reference passing systems; however, they cannot express scenarios involving 'local' concurrency inside a process. EFCPs remove this limitation. As a result, practical modelling of mobile systems becomes more convenient, e.g. multicast can be naturally expressed.

An EFCP system specification, due to the local concurrency its processes may contain, can yield a very large state space. Thus, to define the EFCP syntax the notion of finite processes is required. Such processes have special syntax ensuring that the number of actions they can execute is bounded in advance. To this end, also a more powerful sequential composition operator ' ; ' is used instead of prefixing. Furthermore, an almost linear translation from safe EFCP to safe FCP has been developed, which forms the basis of formal verification of RPSs. The purpose of translating EFCP to FCP is for the latter to be translated to safe low-level PN, for which efficient verification techniques can be applied.

In the translation we had to face two main challenges. Firstly, it has to elim-

[1] Available from http://service-technology.org/tools/lola.

inate the parallel composition operator iniside threads and the use of sequential composition. Since an FCP consists of sequential processes (threads), any thread of an EFCP that is not sequential must be converted to a sequential one. This has be done by shifting all the concurrency to the initial term. Moreover, sequential composition after the translation has been replaced by prefixing. To that end, to avoid blow up in size, new declarations are introduced during this process. Secondly, we have to ensure that the order of actions is preserved and that the context (binding of names) is correct. To address these issues, we realised that extra communication between threads may be required. As a result, new process definitions are introduced in two cases. The first is when local concurrency exists within a thread. The second case is when there is a sequential composition with a non-trivial left-had side.

Moreover, a formal definition of the translation that consists of several rules is defined and based on this formal definition a tool that automates the translation has been implemented. The SpiNNaker case study demonstrates that EFCPs allow for a concise expression of multicast communication, and is suitable for practical modelling. The SpiNNaker's EFCP model was translated to FCP and then to PN. Then, deadlock checking was performed to the PN model with the LoLA tool.

In our future work we intend to investigate the relationship between the transition systems generated by EFCPs and those generated by the corresponding FCPs, with the view to prove the correctness of the proposed translation. Moreover, it should be noted that for formally verifying the SpiNNaker architecture, we have used a small model (2x2). Although its small size, the resulting PN has significantly large size which leads to a big verification time as well. To that end, we would also like to evaluate the scalability of the proposed approach on a range of models and optimise the translation, e.g. by reducing the number of generated defining equations and by lifting it to non-safe processes.

Chapter 4

Diagnosability under Weak Fairness

4.1 Introduction

The *diagnosability* [1, 5, 36, 42, 43, 44, 47, 58, 59, 87, 90] of systems (e.g., autonomous systems [26]) has become an interesting topic to both artificial intelligence and control theory communities. For instance, a key feature of autonomous systems is their high adaptivity. They operate by sensing the environment and learn to make decisions on their actions without the need of any human interference. The procedure of describing some abnormal behaviour of a system is called *diagnosis*. In formal verification, diagnosability is a specification property that ascertains whether it is possible the detection of fault given a set of observations [5]. When we are able to deduce the occurrence of a fault after observing the system's behaviour for fairly long time, we say that the system is diagnosable [57]. On the contrary, it is often the case that we cannot conclude about the occurrence of fault. In this case, the system is not diagnosable. Thus, more sensors should be added in the system to be possible the detection of a fault.

Recent work [44] presented a diagnosis method that encompasses *weak fairness* [94]. There, concurrent systems are modelled by partially observable safe Petri nets, and diagnosis is carried out under the assumption that all executions of the Petri net are weakly fair, that is, the only infinite executions admitted are

those in which any transition *enabled* at some stage will be *disabled* at some later stage, i.e. either it will actually fire later in that execution, or else some conflicting transition will fire. Under this assumption, a given finite observation diagnoses a fault if no finite execution yielding this observation can be extended to a weakly fair fault-free execution. The work in [44] gave a procedure for deciding this diagnosis problem. It remained open for which systems this procedure reliably diagnoses faults, i.e. how to determine whether a system is *diagnosable* under the weak fairness assumption. In this thesis, this problem is addressed.

Note that a first definition of diagnosability under weak fairness was proposed in [1]. However, that definition is incompatible with the notion of diagnosis in [44] and contains a major flaw, as explained below.

The following contributions are made in this chapter:

- A notion of weakly fair (WF) diagnosability is developed, which corrects and supersedes the one in [1].
- We give two alternative characterisations of executions that *witness* violations of WF-diagnosability together with a proof of their equivalence.
- The special case where fault transitions are not WF is further investigated, i.e. a fault is a *possible* outcome in the system but not one that is *required* to happen. (The examples in Sect. 4.6 suggest that this is a reasonable assumption in practice.) Under this assumption, the notion of a witness can be significantly simplified.
- We develop a method for verifying WF-diagnosability in this case, and evaluate it experimentally.
- A general method for verifying WF-diagnosability, which allows faults to be WF.

The chapter is organised as follows: Sect. 4.2 discusses existing notions of diagnosability and explains why they are problematic for concurrent systems. Sect. 4.3 develops a new notion of WF-diagnosability and witnesses of its violation. Sect. 4.4 presents the construction of the verifier, which is evaluated in

Sect. 4.6. Sect. 4.7 presents a generalised method for verifying WF-diagnosability, which allows faults to be WF.

The work presented in this chapter was previously published as a conference paper at ACSD'14 [36], and a journal paper has been submitted to ACM TECS [35].

4.2 Petri nets and diagnosability

This section explains why the standard notion of diagnosability, as well as the notion of WF-diagnosability developed in [1], are problematic, which motivates the new definition, to be presented later.

Throughout the paper it is assumed that the system is modelled as a labelled Petri net (LPN) $\mathcal{N}$, where each transition is labelled with the performed action. The actions are partitioned into *observable* and *silent,* i.e. there is a labelling function ℓ mapping the LPN's transitions to $O \cup \{\varepsilon\}$, where O is an alphabet of *observable* actions and $\varepsilon \notin O$ is the empty word denoting the *silent* action. (Intuitively, observable actions correspond to controller commands and sensor readings, while the silent action models some internal activity that is not recorded by sensors.) This labelling function ℓ can be naturally extended to finite and infinite executions of the LPN, projecting them to words in O^* or O^ω. We assume that the LPN is free from deadlocks and divergencies, i.e. every execution of the LPN can be extended to an infinite one, and every infinite execution of the LPN has infinitely many observable transitions. Some of the transitions are designated as *faults;* w.l.o.g., we assume that none of them is observable. An example in Fig. 4.1 shows an LPN with the observable transitions t_3, t_4 and t_5 with $\ell(t_3) = a$, $\ell(t_4) = b$ and $\ell(t_5) = \mathit{tick}$ (the other transitions are unobservable). Note that we draw faults as black boxes, and the observable transitions are shaded.

The usual interleaving semantics is used in this paper; in particular, references to time in temporal modalities like 'eventually' and 'always' are w.r.t. the 'internal' clock that progresses when some transition of the LPN fires.

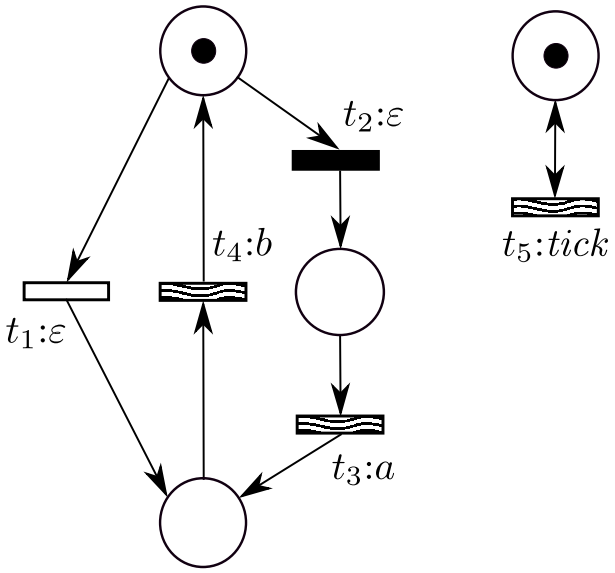

Figure 4.1: This LPN without t_5 would be diagnosable, but t_5 makes it undiagnosable. Making t_3 WF makes the LPN diagnosable.

4.2.1 Standard diagnosability

Given a finite execution σ of the LPN, the observer sees the outputs of the system $\ell(\sigma) \in O^*$, and needs to conclude whether some fault transition t has definitely occurred in σ. In a diagnosable system, once a fault has occurred, the observer is able to *eventually* detect this. That is, provided that the suffix of σ after the first occurrence of a fault in it is sufficiently long, the observer should be able to conclude that each infinite execution with a prefix having the same projection $\ell(\sigma)$ contains a fault, i.e. a fault has either already occurred or will definitely occur in the future. Let us first recall the definition of standard diagnosability:[1]

Definition 10 (Diagnosability)**.** An LPN is diagnosable iff for all its infinite traces σ and ρ such that $\ell(\sigma) = \ell(\rho)$, σ contains a fault iff ρ contains a fault.

In other words, a non-diagnosable LPN has two infinite executions having the same projection onto the observable actions and such that one of them contains

[1] This definition is taken from [58]. It is subtly different from the original definition in [87], but equivalent for finite state systems, and simpler to use in practice. (An LPN has finitely many reachable markings iff it is bounded.)

a fault and the other does not; such a pair of traces constitutes a *witness* of diagnosability violation.

For example, the LPN in Fig. 4.1 is not diagnosable. Indeed, the diagnoser can only conclude that the fault has occurred after observing a. However, the infinite execution $t_2 t_5^\omega$ contains a fault but never fires t_3. Hence, the pair of executions $(t_2 t_5^\omega, t_5^\omega)$ constitutes a witness of diagnosability violation. Nevertheless, if t_5 is removed, the LPN becomes diagnosable.

4.2.2 Weak fairness

The example in Fig. 4.1 exhibits a pathological property of this notion of diagnosability: a diagnosable system ceases to be such simply because some unrelated concurrent activity is added to the specification. In practice, it is often reasonable to assume that the system is keen to fire its enabled transitions, and *cannot perpetually ignore an enabled transition.* In other words, one can consider the LPN in Fig. 4.1 diagnosable, by declaring the infinite execution $t_2 t_5^\omega$ impossible.

To capture this idea formally, the notion of *weak fairness* is helpful [94]. Suppose the designer wants to disallow some of the transitions to be perpetually ignored when enabled. We call such transitions *weakly fair* (WF). An infinite execution σ of the LPN is called *weakly fair* (WF) if for each WF transition t, if t is enabled after some prefix of σ then the rest of σ contains some transition in $({}^\bullet t)^\bullet$, see Fig. 4.2. All finite executions are regarded as WF. We now can use the set of WF executions as the semantics of the LPN, i.e. other executions are considered impossible. Coming back to the example in Fig. 4.1, if t_3 is WF then the execution $t_2 t_5^\omega$ is not WF and thus impossible, and so the LPN becomes diagnosable.

It is tempting to derive the definition of WF-diagnosability simply by taking Def. 10 and restricting it to WF executions. In fact, such an approach was taken in [1], where an LPN $\mathcal{N}$ was said to be WF-diagnosable iff for all its infinite WF executions σ and ρ such that $\ell(\sigma) = \ell(\rho)$, σ contains a fault iff ρ contains a fault.

Unfortunately, this definition contains a major flaw, demonstrated by the example in Fig. 4.3. This LPN would be said to be diagnosable, while it is not possible for the observer to detect a fault in finite time, as one would have to

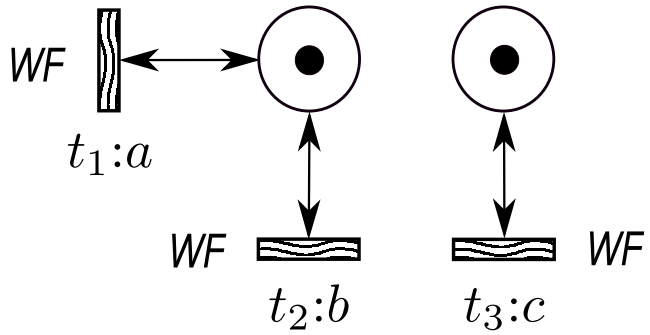

Figure 4.2: (i) The execution $(t_1t_2t_3)^\omega$ is WF as no enabled transition is perpetually ignored by it. (ii) The execution $(t_1t_2)^\omega$ is not WF as t_3 is enabled but all the transitions in $({}^\bullet t_3)^\bullet = \{t_3\}$ are perpetually ignored. (iii) The execution $(t_1t_3)^\omega$ is WF: even though t_2 is perpetually ignored, $t_1 \in ({}^\bullet t_2)^\bullet = \{t_1, t_2\}$ is fired.

observe the infinite trace a^ω to positively conclude that the fault has occurred.

4.3 Weakly fair diagnosability

To fix the problems exhibited in Sect. 4.2, we present a corrected definition of WF-diagnosability, where the possibility of detecting a fault in finite time is imposed. Intuitively, it states that each infinite WF execution containing a fault must have a finite prefix after which it is possible to conclude unambiguously that the fault has either occurred or will inevitably occur in future. Below we denote by '$<$' the prefix relation on sequences.

Definition 11 (WF-diagnosability)**.** An LPN is WF-diagnosable iff each infinite WF execution σ containing a fault has a finite prefix $\hat{\sigma}$ such that every infinite WF execution ρ with $\ell(\hat{\sigma}) < \ell(\rho)$ contains a fault.

The LPN in Fig. 4.3 is not WF-diagnosable according to Def. 11, as for each finite prefix (say, $t_1t_3^n$ for some $n \in \mathbb{N}$) of the infinite WF execution $t_1t_3^\omega$ containing a fault, there is a finite execution $(t_2t_3^n)$ with the same projection to observable actions, that can be extended to an infinite WF execution without a fault (e.g. $t_2t_3^n(t_3t_4)^\omega$). As it was mentioned in Sect. 4.2.2, the transitions that are expected to be weakly fair are denoted using the WF notation.

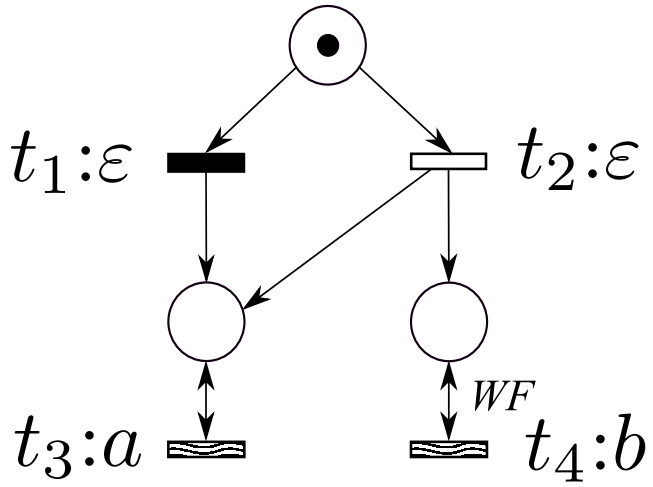

Figure 4.3: This LPN is WF-diagnosable according to the definition from [1], but not according to the corrected definition (Def. 11 and Lemma 4.3.1). Note that the observer cannot detect the fault in finite time.

In this example one can also identify a fault-free *infinite* execution $t_2 t_3^\omega$ that is in itself not WF, but each of its finite prefixes can be extended to an infinite fault-free WF execution. As we shall see, such an execution can always be found in a *bounded* LPN that is not WF-diagnosable.

Definition 12 (Witness for a bounded LPN)**.** Let $\mathcal{N}$ be a bounded LPN. A pair of infinite executions (σ, ρ) with $\ell(\sigma) = \ell(\rho)$ is called *witness* (of WF-diagnosability violation) if σ is WF and contains a fault, and every prefix of ρ can be extended to an infinite fault-free WF execution.

Lemma 4.3.1 (WF-diagnosability of a bounded LPN)**.** *A bounded LPN $\mathcal{N}$ is WF-diagnosable iff no witness of its WF-diagnosability violation satisfying the conditions of Def. 12 exists.*

Proof. If a witness satisfying Def. 12 exists then the condition of Def. 11 is violated, as for any prefix of σ one can choose a prefix of ρ with the same projection, which can be extended to a fault-free WF execution, i.e. $\mathcal{N}$ is not WF-diagnosable.

In the reverse direction: Suppose $\mathcal{N}$ is not WF-diagnosable. Then, according to Def. 11, there exists an infinite, WF, faulty execution σ such that for every finite prefix $\hat{\sigma} < \sigma$ there exists some infinite, WF, fault-free execution ρ with

$\ell(\hat{\sigma}) < \ell(\rho)$. From σ, we shall construct a pair of executions (σ', ρ') constituting a witness according to Def. 12.

Let K be the number of states (i.e. reachable markings) of $\mathcal{N}$. Let $m(\sigma, i)$ denote the marking generated by the i-th observable transition in σ; since $\mathcal{N}$ has no divergencies, it is well-defined for all $i \geq 1$. Moreover, let $s(\sigma, i, j)$ denote the interval of σ starting after the i-th observable transition and ending at the j-th observable transition, for all $0 < i < j$. Furthermore, let k be the number of observable transitions in σ before the first occurrence of a fault.

By the pigeonhole principle, some marking m must satisfy $m = m(\sigma, i)$ for infinitely many i, and thus one can construct an infinite, strictly ascending sequence of indices $(i_j)_{j \geq 0}$ such that $i_0 > k$, and all $j \geq 0$ satisfy (i) $m(\sigma, i_j) = m$, and (ii) $s(\sigma, i_j, i_{j+1}) \cap ({}^\bullet t)^\bullet \neq \emptyset$ for every WF transition t enabled in m (such a subsequence exists since σ is WF and m appears infinitely often). Let $\hat{\sigma}$ be the prefix of σ with $|\ell(\hat{\sigma})| = i_K$.

Now, let ρ be an infinite, WF, fault-free sequence with $\ell(\hat{\sigma}) < \ell(\rho)$. By the pigeonhole principle, there must be two indices $0 \leq j_1 < j_2 \leq K$ with $m(\rho, i_{j_1}) = m(\rho, i_{j_2}) = m'$.

We are now ready to conclude. Consider the execution σ', identical to σ up to $m(\sigma, i_{j_1})$ and then executing $s(\sigma, i_{j_1}, i_{j_2})^\omega$. This execution is infinite, contains a fault, and is WF by construction. Moreover, let ρ' be an infinite execution identical to ρ up to $m(\rho, i_{j_1})$ and then executing $s(\rho, i_{j_1}, i_{j_2})^\omega$. By construction, $\ell(\sigma') = \ell(\rho')$ but ρ' does not contain a fault. Also, every prefix of ρ' can be extended to a WF fault-free execution by going to the next occurrence of m' and then appending any suffix of ρ starting at an occurrence of m'. Thus, (σ', ρ') constitutes a witness. □

Def. 12 is difficult to use due to the necessity to consider every prefix of ρ. Lemma 4.3.2 provides an alternative characterisation for ρ. Its advantage is that instead of considering infinitely many prefixes of ρ, it considers a single well chosen one.

Lemma 4.3.2. *Let $\mathcal{N}$ be a bounded LPN and ρ be an infinite execution. Then the following two statements are equivalent:*

1. *every prefix of ρ can be extended to an infinite fault-free WF execution;*

2. *ρ is fault-free and has a prefix $\hat{\rho}$ such that ρ passes through the marking reached by $\hat{\rho}$ infinitely many times, and $\hat{\rho}$ can be extended to an infinite fault-free WF execution.*

Proof. (1) $\Rightarrow$ (2) Suppose every prefix of ρ can be extended to an infinite fault-free WF execution ($*$). Then ρ must be fault-free, as otherwise some prefix of ρ would contain a fault, contradicting ($*$). Since the LPN is bounded and ρ is infinite, ρ passes through some marking m infinitely many times. Let $\hat{\rho}$ be a prefix of ρ that reaches m first time. By ($*$), $\hat{\rho}$ can be extended to an infinite WF execution.

(1) $\Leftarrow$ (2) Suppose ρ is fault-free and has a prefix $\hat{\rho}$ such that ρ passes through the marking m reached by $\hat{\rho}$ infinitely many times, and $\hat{\rho}$ can be extended to an infinite fault-free WF execution $\hat{\rho}\rho'$. Since ρ passes through m infinitely many times, every prefix of ρ can be extended to a longer prefix reaching m. In turn, this longer prefix can be extended by ρ' to an infinite fault-free execution, as ρ' is enabled from m. Moreover, this execution is WF, as $\hat{\rho}\rho'$ is WF, and ρ' fully determines whether the execution is WF or not. □

We note that in certain practical cases, the witness definition can be simplified. In particular, we consider the case when no fault transition is WF: Then one can further simplify the requirements imposed on ρ in Lemma 4.3.2.

Lemma 4.3.3. *Let $\mathcal{N}$ be a bounded LPN where no fault transition is WF, and let ρ be an infinite execution. Then the following two statements are equivalent:*

1. *every prefix of ρ can be extended to an infinite fault-free WF execution;*

2. *ρ is fault-free.*

Proof. The proof of (1) $\Rightarrow$ (2) is as in the proof of Lemma 4.3.2. As for the other direction, suppose ρ is infinite and fault-free and take any finite prefix $\hat{\rho}$ of ρ such that $\hat{\rho}\rho' = \rho$. We construct an infinite, WF, fault-free continuation of $\hat{\rho}$. If ρ itself is WF then we are done. Otherwise there exists some WF transition t that is enabled at some point in ρ after which ρ contains no more transition from $({}^\bullet t)^\bullet$; note that t is not a fault by the assumption. But this means that firing t cannot disable any transition in the rest of the execution, so we can insert it anywhere

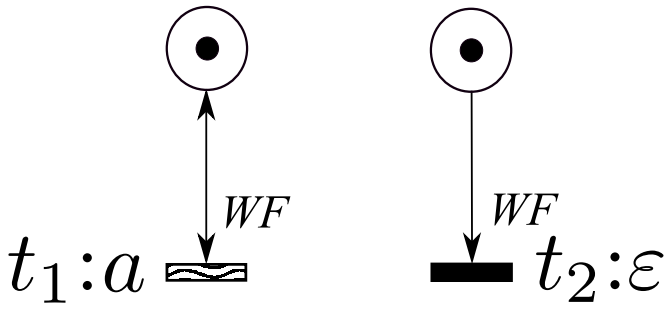

Figure 4.4: A bounded LPN illustrating that the assumption about faults being non-WF is essential for Lemma 4.3.3. Despite the presence of an infinite fault-free execution t_1^ω, this LPN is trivially WF-diagnosable, as the fault must occur in every infinite WF execution.

into ρ' without disabling the rest of this execution. The repeated application of this insertion process yields the required continuation of $\hat{\rho}$, and it always can be done in such a way that no enabled WF transition is perpetually ignored by the insertion process, and no transition from ρ' is indefinitely delayed by the newly inserted transitions. □

Lemma 4.3.3 provides another characterisation of ρ in witnesses provided that the net contains no WF fault transitions. This result is central for the WF-diagnosability verification method proposed in Sect. 4.4. Note also that this characterisation is quite similar to the (flawed) definition from [1], but with the following important differences: (i) it is limited to bounded LPNs without WF faults, and (ii) ρ is not required to be WF. As an example, a witness of WF-diagnosability violation for the LPN in Fig. 4.3 would be $(t_1 t_3^\omega, t_2 t_3^\omega)$; note that the latter execution is not WF, but that each of prefix can be extended to a WF execution.

It should be noted that the assumption that the faults are not WF is essential for Lemma 4.3.3. Indeed, consider the LPN in Fig. 4.4, where t_1^ω constitutes an infinite, fault-free execution as required by Lemma 4.3.3 (2) that has the same observations as the faulty execution $t_2 t_1^\omega$. Nonetheless, this LPN is trivially WF-diagnosable, as all its infinite WF executions contain the WF fault transition. Thus, the assumption about the absence of WF faults cannot be dropped in Lemma 4.3.3.

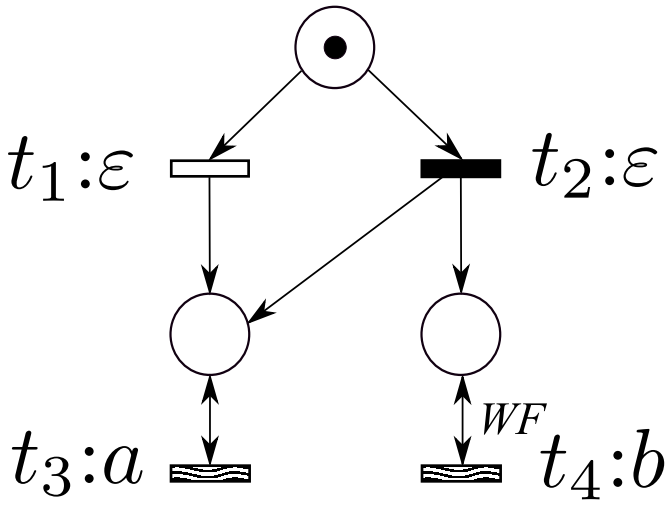

Figure 4.5: An LPN similar to that in Fig. 4.3, but with a different choice of a fault transition. It is not diagnosable but WF-diagnosable, as an occurrence of a fault enables t_4, which can be perpetually ignored under the non-WF semantics, but must eventually fire — thus diagnosing the fault — under the WF semantics.

4.4 Checking WF-diagnosability

In this section we show how checking WF-diagnosability can be re-formulated in terms of LTL-X [55, 80] model checking.

The proposed approach works for a bounded LPN $\mathcal{N}$. We perform various operations on $\mathcal{N}$ to obtain another bounded LPN $\mathcal{V}$, called the *verifier*, which we check against a *fixed* LTL-X formula (in particular, its size does not depend on $\mathcal{N}$). To achieve this, we exploit the ability of many existing model checkers to handle weak fairness directly.[1]

We first introduce the operations on $\mathcal{N}$ needed to obtain $\mathcal{V}$ (Sect. 4.4.1), then recall the approach for non-WF diagnosability (Sect. 4.4.2), and finally present the modifications necessary to handle WF-diagnosability for the special case where no fault transition is WF (Sect. 4.4.3). The general case of bounded nets is considered in Sect. 4.7. We use the net in Fig. 4.5 as a running example.

[1] The algorithm looking for an accepting (lasso-shaped) execution of a Büchi automaton can be modified in such a way as to ignore non-WF executions.

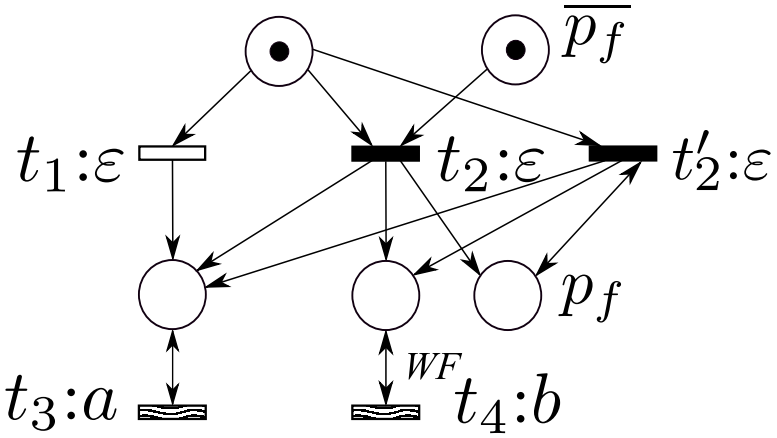

Figure 4.6: Fault tracking net $\mathcal{N}^{ft}$ for the LPN in Fig. 4.5. Two additional places, $\overline{p_f}$ and p_f, of which $\overline{p_f}$ is initially marked indicating that no fault has occurred so far, are added. A token moves from $\overline{p_f}$ to p_f, indicating that a fault has occurred. The transition f' is added for each fault transition f to simulate that a fault transition may fire several times.

4.4.1 Net operations

We are concerned with the state-based LTL-X verification. However, the definition of diagnosability in Sect. 4.3 is action-based, and thus has to be re-formulated in terms of states. The first operation is defined for this purpose.

Fault monitor: We use the construction proposed in [58], we will need to keep track of whether some execution contains a fault transition. Given $\mathcal{N}$, the net $\mathcal{N}^{ft}$ denotes $\mathcal{N}$ extended with two additional places $\overline{p_f}$ and p_f of which $\overline{p_f}$ is initially marked, indicating that no fault has happened so far. Then we make every fault transition move a token from $\overline{p_f}$ to p_f, indicating that a fault has occurred. Also, since a fault transition may fire several times in $\mathcal{N}$, another transition f' is added for each fault transition f, in order to simulate these subsequent firings in $\mathcal{N}^{ft}$. The construction is illustrated in Fig. 4.6, where it is applied to Fig. 4.5.

In terms of behaviour, $\mathcal{N}$ and $\mathcal{N}^{ft}$ are equivalent in a strong sense. Suppose that the transitions of $\mathcal{N}$ are injectively labelled, and the transitions of $\mathcal{N}^{ft}$ retain these labels, with the label of f and f' being the same. Then these two nets are strongly bisimilar. Moreover, if $\overline{p_f}$ in $\mathcal{N}^{ft}$ is unmarked then a fault occurred in the past.

Stubs: We will want to know whether an infinite execution perpetually ignores certain enabled WF transitions. Given a subset of $\mathcal{N}$'s WF transitions and a fresh initially marked place *stub_monitor*, we can turn these transitions into *stubs* by removing all their outgoing arcs and adding *stub_monitor* to their presets, as we proposed in [35, 36].

Stubs are not meant to be executed: in fact, the LTL-X formulae will make such executions 'irrelevant' by demanding that *stub_monitor* remains always marked. Then, a 'relevant' WF execution that keeps *stub_monitor* marked cannot enable a stub forever.

Removing transitions: We can remove a given subset of transitions from an LPN, together with their incoming and outgoing arcs [58].

Synchronising: Let $\mathcal{N}$ and $\mathcal{N}'$ be two LPNs with disjoint sets of places and transitions. Intuitively, the *synchronisation* of $\mathcal{N}$ and $\mathcal{N}'$ puts $\mathcal{N}$ and $\mathcal{N}'$ side-by-side, and then each observable transition t of $\mathcal{N}$ is fused (merged) with each transition t' of $\mathcal{N}'$ that has the same label (each fusion produces a new transition, and t and t' remain in the result) as was proposed in [58]. Thus the synchronised net has three types of transitions: those from $\mathcal{N}$ (e.g., t), those from $\mathcal{N}'$ (e.g., t'), and the fused ones (e.g., (t, t')).

4.4.2 Verifying ordinary diagnosability

We recall the verification of (non-WF) diagnosability from [58] and show that it is unsuitable for WF-diagnosability. Let $\mathcal{N}$ be the original LPN. The construction works in the following steps:

1. Let $\mathcal{N}^{ft}$ be the fault tracking net corresponding to $\mathcal{N}$.
2. Let $\mathcal{N}'$ be a copy of $\mathcal{N}$ with disjoint sets of places and transitions names.
3. Let $\mathcal{N}_s$ be the result of synchronising $\mathcal{N}^{ft}$ and $\mathcal{N}'$.
4. Remove from $\mathcal{N}_s$ all observable transitions of $\mathcal{N}^{ft}$.
5. Remove from $\mathcal{N}_s$ all observable and fault transitions of $\mathcal{N}'$.
6. Call the resulting net $\mathcal{V}$.

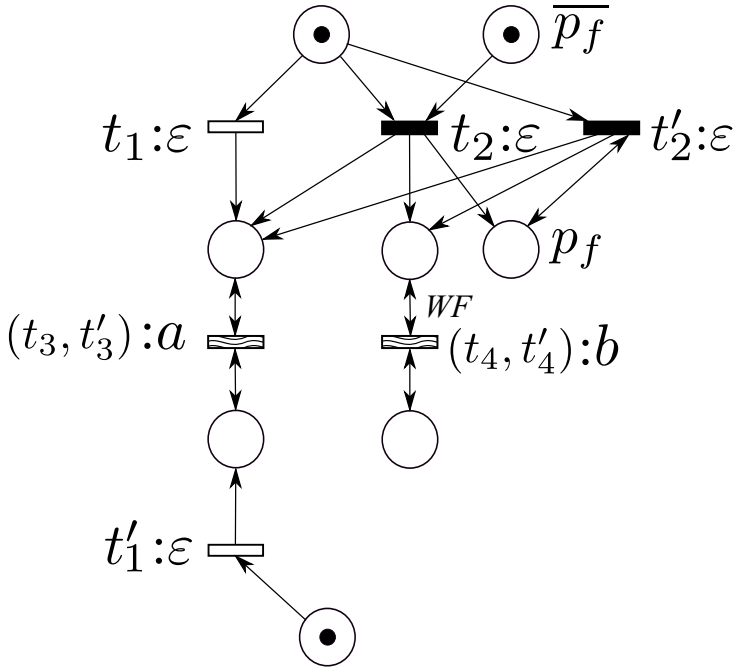

Figure 4.7: The (non-WF) verifier for the LPN in Fig. 4.5. Here, the $\mathcal{N}^{ft}$ is synchronised with $\mathcal{N}'$ making the $\mathcal{N}_s$. $\mathcal{N}'$ is a copy of $\mathcal{N}$. From $\mathcal{N}_s$ all the observable transition of $\mathcal{N}^{ft}$ and $\mathcal{N}'$ and the fault transitions of $\mathcal{N}'$ have been removed. The resulting net is the verifier $\mathcal{V}$. Here, transitions t_3 and t_4 are the fused transitions (t_3,t_3') and (t_4,t_4') respectively.

The original net and its copy have disjoint sets of places and transitions names. This is necessary because we need to distinguish the places and transitions that exist in each net and to avoid possible errors in verification tools.

Moreover, Note that after $\mathcal{V}$ has been built, it is no longer necessary to remember which actions are visible and which are not, and so we can disregard all the labelling and treat $\mathcal{V}$ as an unlabelled PN. This construction is illustrated in Fig. 4.7.

It turns out [58] that $\mathcal{N}$ is diagnosable iff the following LTL-X property holds for all traces of $\mathcal{V}$:

$$diag \stackrel{\text{df}}{=} \Box \overline{p_f},$$

i.e. one simply requires that no infinite trace of $\mathcal{V}$ contains an occurrence of a fault. Thus, place $\overline{p_f}$ always contains a token.

Conversely, a counterexample satisfying $\Diamond \neg \overline{p_f}$ is an infinite execution of $\mathcal{V}$ containing a fault; when projected to the parts corresponding to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, it gives a witness of (non-WF) diagnosability violation, i.e. two infinite executions of $\mathcal{N}$ that have the same projection on the set of observable actions but the first contains a fault while the second does not. Similarly, such a pair of executions corresponds to an infinite trace of $\mathcal{V}$, with the first being executed by the part of $\mathcal{V}$ corresponding to $\mathcal{N}^{ft}$, and the second (without occurrences of faults) being executed by the part of $\mathcal{V}$ corresponding to $\mathcal{N}'$.

Unfortunately, this construction is not appropriate for WF-diagnosability, even if the executions of the verifier are restricted to be WF. For example, consider the net in Fig. 4.5. The verifier proposed in [58] is shown in Fig. 4.7. It has an infinite execution containing a fault, $t_2 t_1' t_3^\omega$, which, when projected to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, yields a pair of traces constituting a witness of diagnosability violation. However, this verifier cannot be used for checking WF-diagnosability simply by restricting its executions to be WF, as the same execution $t_2 t_1' t_3^\omega$ is actually WF, since t_4 is not permanently enabled by it (in fact, it is a dead transition in the verifier). Therefore, this execution is a false negative (the original LPN is in fact WF-diagnosable). Note that when this WF execution of the verifier is projected to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, the resulting pair of traces will not constitute a witness of WF-diagnosability, as the former projection will be a non-WF execution of $\mathcal{N}^{ft}$ that perpetually ignores an enabled transition t_4.

Below, we amend $\mathcal{V}$ to fix this problem for bounded LPNs with no WF faults.

4.4.3 Verifier for non-WF fault transitions

In this section we introduce a general verifier for bounded LPNs. Let $\mathcal{N}$ be a bounded LPN, in which no fault transition is WF. We keep the basic idea of the verifier construction from Sect. 4.4.2, i.e. verifier $\mathcal{V}_{WF}$ will be the synchronisation of two nets, and a counterexample to the LTL-X formula will give a faulty execution σ in one net, and a fault-free execution ρ in the other net, such that (σ, ρ) is a witness.

The first important change is to check the formula only against WF executions. As seen in Sect. 4.4.2, this alone is not enough: The false counterexample

obtained for Fig. 4.5 comes from the fact that $\mathcal{V}_{WF}$ allows σ to perpetually ignore a transition (here: t_4) if ρ does not enable it. We use stubs to prevent this from happening. More precisely, $\mathcal{V}$ is constructed as follows:

1. Obtain the net $\mathcal{N}_s$ as in Sect. 4.4.2; its fused transitions are declared non-WF.
2. Turn in $\mathcal{N}_s$ the observable WF transitions of $\mathcal{N}^{ft}$ into stubs; they remain WF.
3. Remove from $\mathcal{N}_s$ all observable and fault transitions of $\mathcal{N}'$.
4. In $\mathcal{N}_s$, make the remaining transitions of $\mathcal{N}'$ non-WF.
5. Call the resulting net $\mathcal{V}_{WF}$.

Fig. 4.8 shows the verifier $\mathcal{V}_{WF}$ for the LPN in Fig. 4.5.

Now we can formulate WF-diagnosability of the original $\mathcal{N}$ as a fixed LTL-X formula on $\mathcal{V}_{WF}$ that has to be checked for infinite WF executions only:

$$diag_{WF} \stackrel{\text{df}}{=} \Box\, \overline{p_f} \;\vee\; \Diamond\, \neg stub\_monitor.$$

Thus a counterexample is an infinite WF execution containing a fault but no stubs.

Theorem 4.4.1 (Correctness of specialised WF Verifier)**.** *Let $\mathcal{N}$ be a bounded LPN where no fault transition is WF. Then $\mathcal{N}$ is WF-diagnosable iff all infinite WF executions of $\mathcal{V}_{WF}$ satisfy $diag_{WF}$.*

Proof. According to Lemma 4.3.1, $\mathcal{N}$ is WF-diagnosable iff no witness (σ, ρ) exists. Since no fault transition is WF, we can employ the simplified condition of Lemma 4.3.3.

First, suppose $diag_{WF}$ is false, i.e. $\mathcal{V}_{WF}$ has an infinite WF execution τ that contains a fault and no stubs. Let σ and ρ be the projections of τ to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, respectively. We claim that (σ, ρ) is a witness. Indeed, since $\mathcal{N}$ has no divergencies, τ must contain infinitely many observable transitions. Thus, both σ and ρ are infinite, and $\ell(\sigma) = \ell(\rho)$ holds; moreover, σ contains a fault but ρ does not. Finally, σ must be WF because τ is and no stubs are fired.

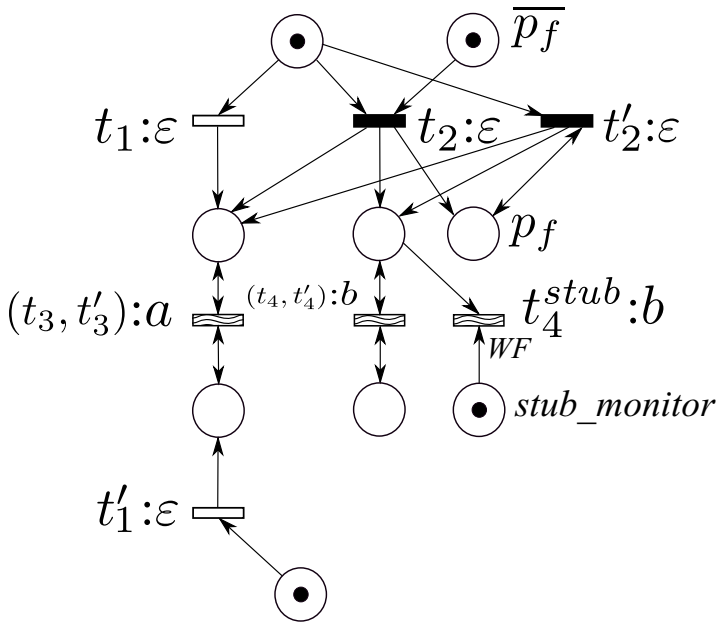

Figure 4.8: The WF verifier for the LPN in Fig. 4.5. We obtain the $\mathcal{N}_s$ as in Sect. 4.4.2. Now, we declare the fused transitions t_3 and t_4 as non-WF. The observable WF transitions of $\mathcal{N}^{ft}$ are turned into stubs, and they remain WF. All observable and fault transitions of $\mathcal{N}'$ have been removed and we make the remaining transitions of $\mathcal{N}'$ non-WF. Here, transitions t_3 and t_4 are the fused transitions (t_3,t_3') and (t_4,t_4') respectively.

For the reverse direction, it is fairly straightforward to see that any witness (σ, ρ) gives rise to an execution τ of $\mathcal{V}_{WF}$ violating $diag_{WF}$, even if ρ satisfies only the simplified condition from Lemma 4.3.3. Moreover, τ is WF because σ is. The fact that ρ is not necessarily WF does not play a role, as ρ is executed in the part of the verifier corresponding to $\mathcal{N}'$ and so contains no WF transitions by construction. □

4.5 A summary of the WF-verifier construction

In this section we revise the whole process of the $\mathcal{V}_{WF}$ construction and why the formula that has to be verified at the end is as such. We will use the same figures

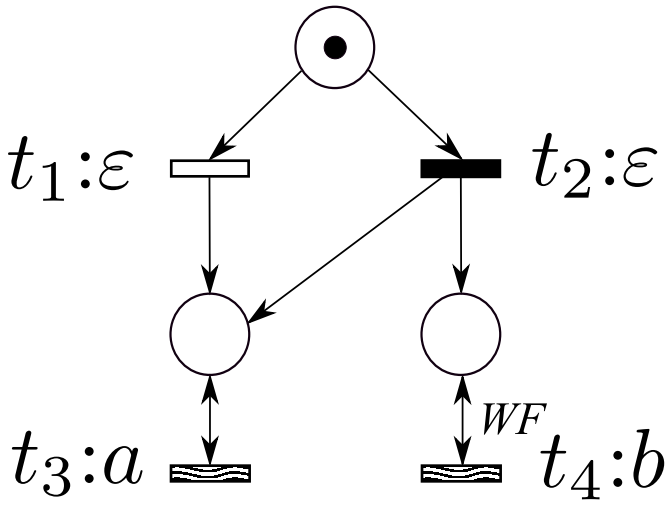

Figure 4.9: An LPN similar to that in Fig. 4.3, but with a different choice of a fault transition. It is not diagnosable but WF-diagnosable, as an occurrence of a fault enables t_4, which can be perpetually ignored under the non-WF semantics, but must eventually fire — thus diagnosing the fault — under the WF semantics.

as is Section 4.4. We use the net in Fig. 4.9 as a running example. Initially we provide the required steps for constructing the verifier, $\mathcal{V}$, that verifies ordinary diagnosability. In addition, we will explain why this construction is not suitable for verifying WF-diagnosability.

The first step of the $\mathcal{V}$ construction is to build the fault monitor. As proposed in [58], we will need to keep track of whether some execution contains a fault transition. Given $\mathcal{N}$, the net $\mathcal{N}^{ft}$ denotes $\mathcal{N}$ extended with two additional places $\overline{p_f}$ and p_f of which $\overline{p_f}$ is initially marked, indicating that no fault has happened so far. Then we make every fault transition move a token from $\overline{p_f}$ to p_f, indicating that a fault has occurred. Also, since a fault transition may fire several times in $\mathcal{N}$, another transition f' is added for each fault transition f, in order to simulate these subsequent firings in $\mathcal{N}^{ft}$. The construction is illustrated in Fig. 4.10, where it is applied to Fig. 4.9.

The next step is to make a copy, $\mathcal{N}'$, of Fig. 4.9. The original net and its copy have disjoint sets of places and transitions names. This is necessary because we need to distinguish the places and transitions that exist in each net and to avoid possible errors in verification tools. Then, we synchronise these two nets.

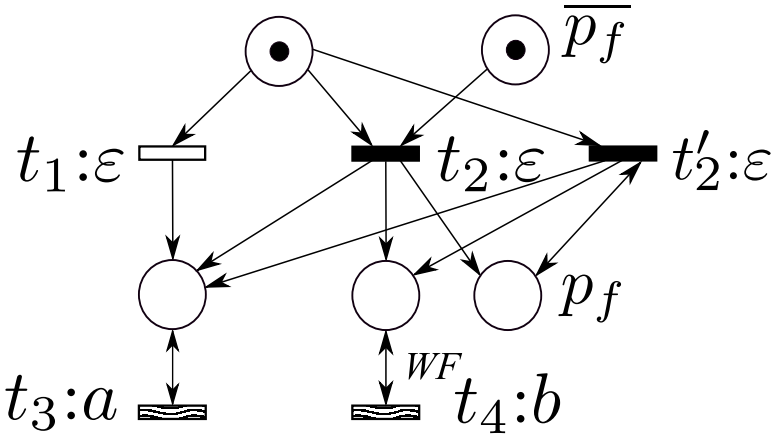

Figure 4.10: Fault tracking net $\mathcal{N}^{ft}$ for the LPN in Fig. 4.5. Two additional places $\overline{p_f}$ and p_f have been added of which $\overline{p_f}$ is initially marked, indicating that no fault has occurred so far. A token moves from $\overline{p_f}$ to p_f, indicating that a fault has occurred. The transition f' is added for each fault transition f to simulate that a fault transition may fire several times.

Intuitively, the *synchronisation* of $\mathcal{N}$ and $\mathcal{N}'$ puts $\mathcal{N}$ and $\mathcal{N}'$ side-by-side, and then each observable transition t of $\mathcal{N}$ is fused with each transition t' of $\mathcal{N}'$ that has the same label (each fusion produces a new transition, and t and t' remain in the result). Thus the synchronised net, $\mathcal{N}_s$, has three types of transitions: those from $\mathcal{N}$ (e.g., t), those from $\mathcal{N}'$ (e.g., t'), and the fused ones (e.g., (t,t')) [58]. Consequently, we remove the observable transition from $\mathcal{N}$ and $\mathcal{N}'$ and the fault transitions in $\mathcal{N}'$. As proposed in [58], we can remove a given subset of transitions from an LPN, together with their incoming and outgoing arcs. The resulting net is the $\mathcal{V}$ as shown in Fig. 4.11

Note that after $\mathcal{V}$ has been built, it is no longer necessary to remember which actions are visible and which are not, and so we can disregard all the labelling and treat $\mathcal{V}$ as an unlabelled PN.

It turns out [58] that $\mathcal{N}$ is diagnosable iff the following LTL-X property holds for all traces of $\mathcal{V}$:

$$diag \stackrel{\text{df}}{=} \Box\,\overline{p_f},$$

i.e. one simply requires that no infinite trace of $\mathcal{V}$ contains an occurrence of a fault. Thus, place $\overline{p_f}$ always contains a token.

Conversely, a counterexample satisfying $\Diamond\,\neg\overline{p_f}$ is an infinite execution of $\mathcal{V}$

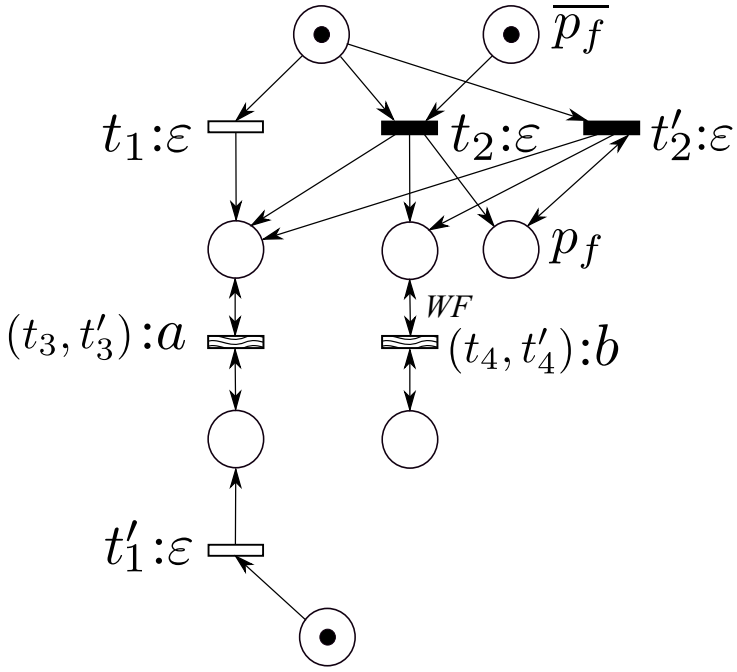

Figure 4.11: The (non-WF) verifier for the LPN in Fig. 4.5. Here, the $\mathcal{N}^{ft}$ is synchronised with $\mathcal{N}'$ making the $\mathcal{N}_s$. $\mathcal{N}'$ is a copy of $\mathcal{N}$. From $\mathcal{N}_s$ all the observable transition of $\mathcal{N}^{ft}$ and $\mathcal{N}'$ and the fault transitions of $\mathcal{N}'$ have been removed. The resulting net is the verifier $\mathcal{V}$. Here, transitions t_3 and t_4 are the fused transitions (t_3,t_3') and (t_4,t_4') respectively.

containing a fault; when projected to the parts corresponding to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, it gives a witness of (non-WF) diagnosability violation, i.e. two infinite executions of $\mathcal{N}$ that have the same projection on the set of observable actions but the first contains a fault while the second does not. Similarly, such a pair of executions corresponds to an infinite trace of $\mathcal{V}$, with the first being executed by the part of $\mathcal{V}$ corresponding to $\mathcal{N}^{ft}$, and the second (without occurrences of faults) being executed by the part of $\mathcal{V}$ corresponding to $\mathcal{N}'$.

Unfortunately, this construction is not appropriate for WF-diagnosability, even if the executions of the verifier are restricted to be WF. For example, consider the net in Fig. 4.5. The verifier proposed in [58] is shown in Fig. 4.7. It has an infinite execution containing a fault, $t_2t_1't_3^{\omega}$, which, when projected to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, yields a pair of traces constituting a witness of diagnosability violation. However, this verifier cannot be used for checking WF-diagnosability simply by

restricting its executions to be WF, as the same execution $t_2 t_1' t_3^\omega$ is actually WF, since t_4 is not permanently enabled by it (in fact, it is a dead transition in the verifier). Therefore, this execution is a false negative (the original LPN is in fact WF-diagnosable). Note that when this WF execution of the verifier is projected to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, the resulting pair of traces will not constitute a witness of WF-diagnosability, as the former projection will be a non-WF execution of $\mathcal{N}^{ft}$ that perpetually ignores an enabled transition t_4.

In this point, we amend $\mathcal{V}$ to fix this problem for bounded LPNs with no WF faults. We keep the basic idea of the verifier construction above, the $\mathcal{V}_{WF}$ will be again the synchronisation of two nets.

The first important change is to check the formula only against WF executions. As seen in Sect. 4.4.2, this alone is not enough: The false counterexample obtained for Fig. 4.5 comes from the fact that $\mathcal{V}_{WF}$ allows σ to perpetually ignore a transition (here: t_4) if ρ does not enable it. We use stubs to prevent this from happening. Given a subset of $\mathcal{N}$'s WF transitions and a fresh initially marked place *stub_monitor*, we can turn these transitions into *stubs* by removing all their outgoing arcs and adding *stub_monitor* to their presets, as we proposed in [35, 36].

Stubs are not meant to be executed: in fact, the LTL-X formulae will make such executions 'irrelevant' by demanding that *stub_monitor* remains always marked. Then, a 'relevant' WF execution that keeps *stub_monitor* marked cannot enable a stub forever.

More precisely, to construct the $\mathcal{V}$ we build the fault monitor as previously. We synchronize the fault monitor with a copy, $\mathcal{N}'$, of $\mathcal{N}$ but now the fused transitions are declared non-WF and we obtain as before the. The next step is to turn in the synchronised net, $\mathcal{N}_s$, the observable WF transitions of fault monitor into stubs that remain WF. As previously, we remove the observable transition from $\mathcal{N}$ and $\mathcal{N}'$ and the fault transitions in $\mathcal{N}'$. Then, we make the remaining transitions of $\mathcal{N}'$ non-WF. The resulting net is the weakly fair verifier, and is shown on Fig. 4.12.

Now we can formulate WF-diagnosability of the original $\mathcal{N}$ as a fixed LTL-X formula on $\mathcal{V}_{WF}$ that has to be checked for infinite WF executions only:

$$diag_{WF} \stackrel{\text{df}}{=} \Box\, \overline{p_f} \;\vee\; \Diamond \neg stub\_monitor.$$

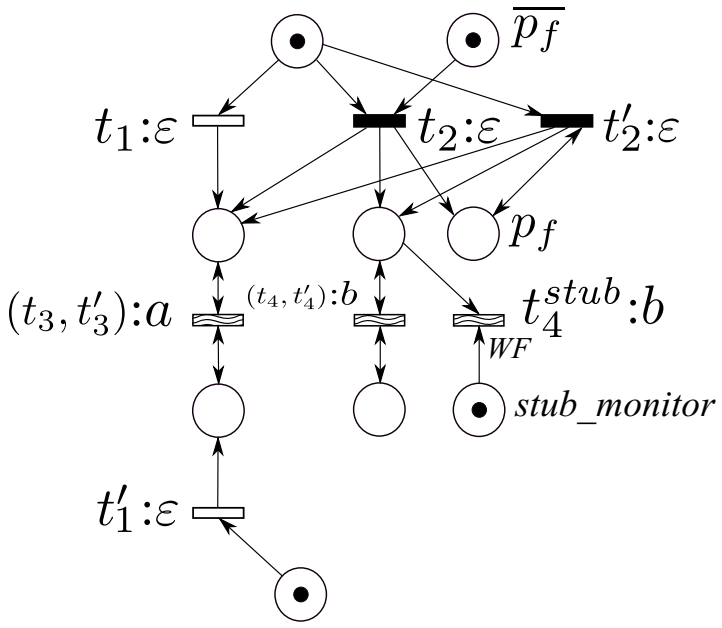

Figure 4.12: The WF verifier for the LPN in Fig. 4.5. We obtain the $\mathcal{N}_s$ as in Sect. 4.4.2. Now, we declare the fused transitions t_3 and t_4 as non-WF. The observable WF transitions of $\mathcal{N}^{ft}$ are turned into stubs, and they remain WF. All observable and fault transitions of $\mathcal{N}'$ have been removed and we make the remaining transitions of $\mathcal{N}'$ non-WF. Here, transitions t_3 and t_4 are the fused transitions (t_3,t_3') and (t_4,t_4') respectively.

The $diag_{WF}$ formula states that the $\mathcal{V}_{WF}$ is diagnosable if place $\overline{p_f}$ is always marked or eventually the place *stub_monitor* is empty. That is necessary because if the faulty transition t_2' fires the WF stub transition will be enabled and after firing the place *stub_monitor* will become empty indicating the occurrence of fault. Similarly if transitions t_2 fires then the place $\overline{p_f}$ becomes empty indicating the fault. Thus a counterexample is an infinite WF execution containing a fault but no stubs.

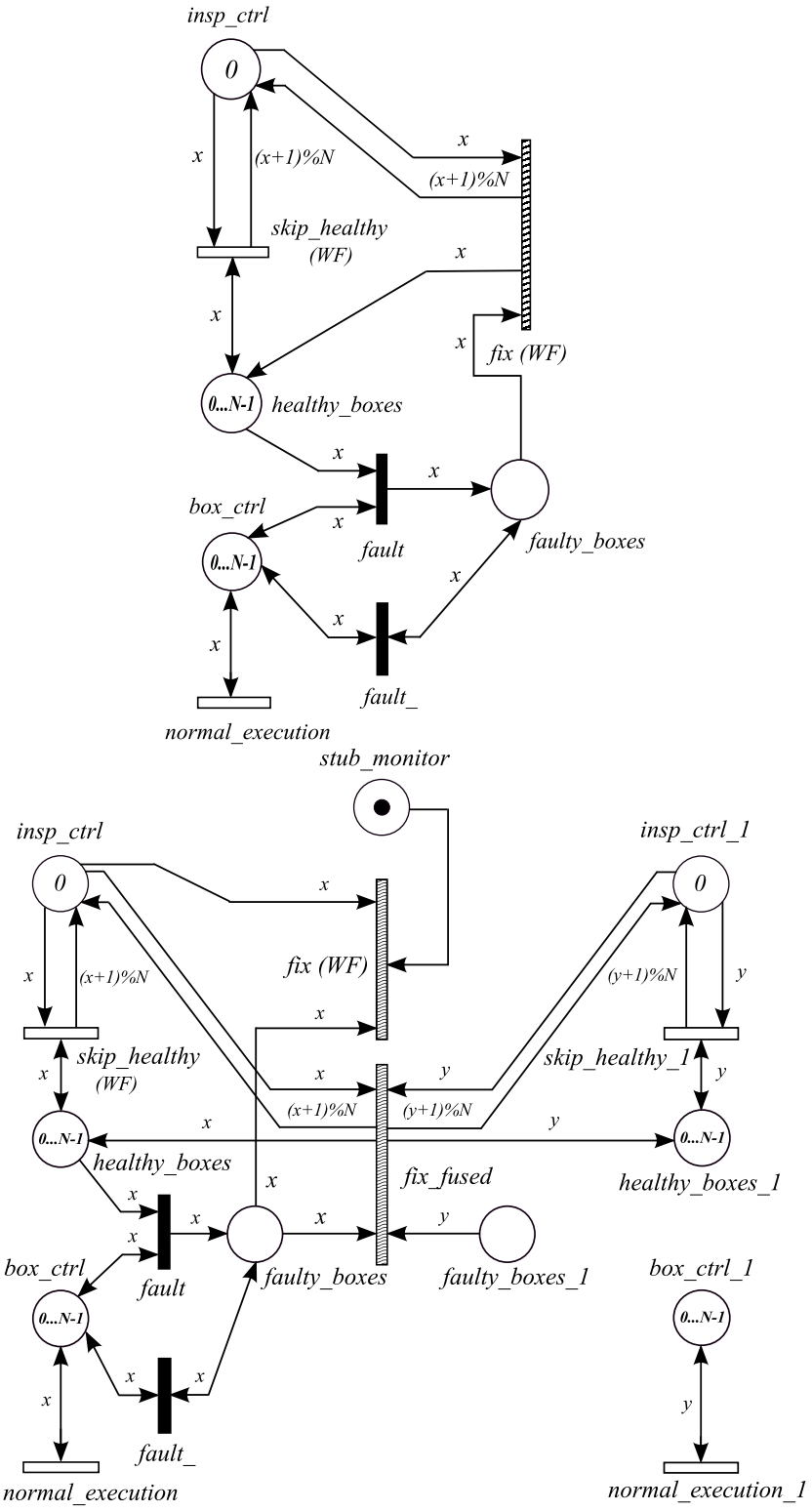

Figure 4.13: The CommBox (n) benchmark (top) and the corresponding verifier (bottom).

4.6 Experimental results

In this section we present experimental results for the proposed WF-diagnosability approach. Furthermore, we demonstrate that the proposed approach can easily be lifted from low-level Petri nets to high-level ones: both the used benchmarks and the corresponding verifiers were modelled using high-level LPNs.

For the verification, the MARIA (modular reachability analyser) tool [60] was used. Since MARIA supports modular verification, it was possible to exploit the modular structure of the verifier (recall that it is built by synchronising two LPNs, see Sect. 4.4) to significantly speed up the verification.

It should be noted that finding interesting benchmarks was a challenging task: Despite a lot of theoretical work done in the area of diagnosability, rather few practical experiments have been conducted. Moreover, we wanted benchmarks where weak fairness is essential, i.e. removing some transitions from the WF set would make the system undiagnosable. Hence, we designed the following two new families of scalable benchmarks.

CommBox (n) Fig. 4.13 shows a high-level LPN modelling the system comprising commutator boxes and an inspector, together with the verifier. It models n boxes commuting telephone calls. Normally, a box just handles telephone calls (the *normal_execution* transition), but occasionally it may register a fault (the *fault* transition) in a telephone line. Such an event, however, does not take the box out of action, and it still continues to commute calls (the *normal_execution* transition) and register further faults (the *fault_* transition). Nevertheless, the registered faults have to be considered and fixed, and so there is an inspector visiting the boxes on a round trip and fixing them if necessary (the *skip_healthy* and *fix* transitions). It is assumed that *fix* is the only observable transition, and one can be sure that a fault has occurred once it fires. Nevertheless, it is possible that the inspector indefinitely postpones visiting the boxes (i.e. its transitions are always preempted by, e.g., *normal_execution* which is always enabled), and so the system is undiagnosable. However, if the transitions modelling the inspector are WF, the system becomes diagnosable, as after a fault the *fix* transition is eventually executed.

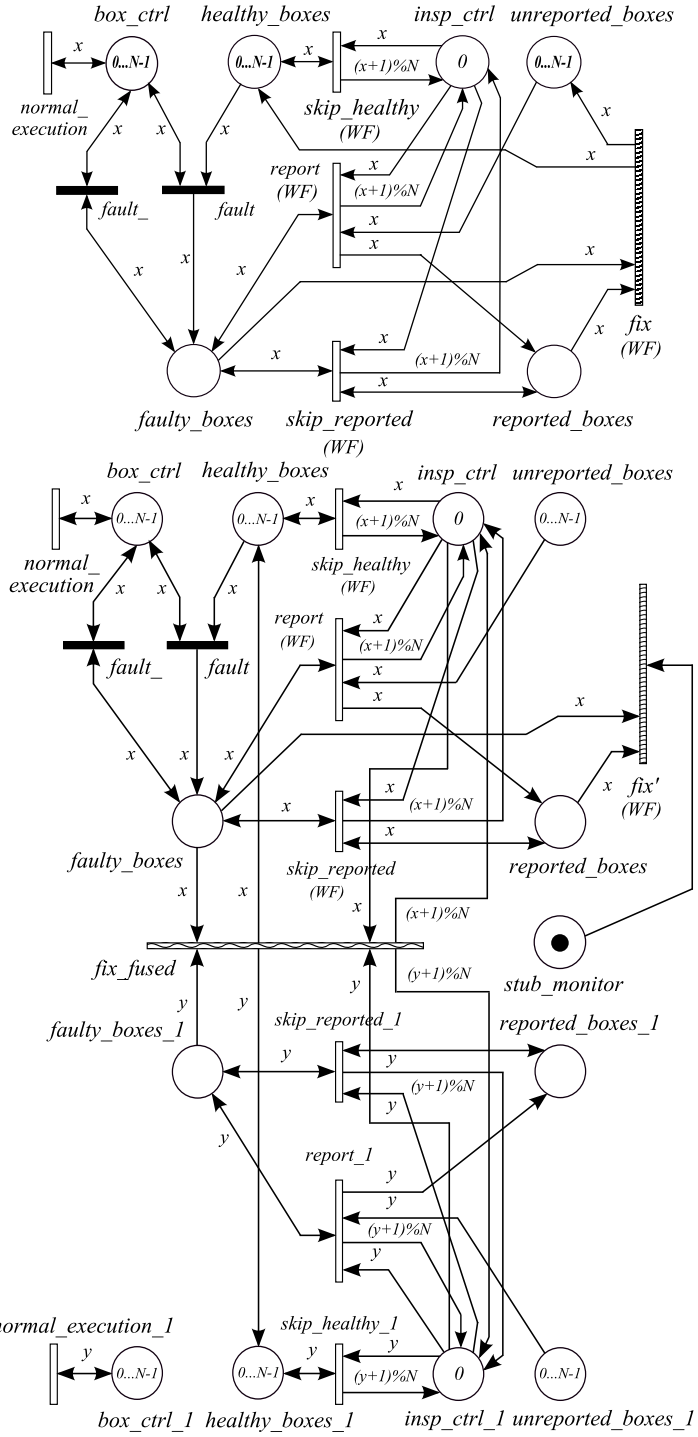

Figure 4.14: The CommBoxTech (n) benchmark (top) and the corresponding verifier (bottom).

Benchmarks	Vrf Time	Vrf Modular Time
CommBox (4)	<1	<1
CommBox (5)	4	1
CommBox (6)	12	4
CommBox (7)	38	14
CommBoxTech (4)	17	6
CommBoxTech (5)	101	33
CommBoxTech (6)	561	162
CommBoxTech (7)	2995	Bug

Table 4.1: Experimental results for CommBox and CommBoxTech benchmarks (all nets are diagnosable).

CommBoxTech (n) Fig. 4.14 shows an elaborated version of the above system, together with its verifier: The inspector reports the faults to a technician, who then fixes them. Again, the inspector's and technician's transitions must both be WF to make the system diagnosable.

The experimental results are summarised in Table 4.1, where the meaning of the columns is as follows (from left to right): name of the benchmark, verification time, and verification time using the modular representation of the verifier. (The time is measured in seconds.) All experiments were conducted on a PC with 64-bit Windows 7 operating system, an Intel Core i7 2.8GHz Processor with 8 cores and 4GB RAM (no parallelisation was used for the results in this table). The Maria tool has confirmed that the diagnosability property holds for these benchmarks. We also discovered a bug in Maria: for the CommBoxTech (7) benchmark there is a mismatch between the verification outcomes in the standard and modular modes.

We also wanted to check that the WF constraint is essential for diagnosability, i.e. that if even one transition is removed from the WF set then the system becomes undiagnosable. These results are summarised in Tables 4.2 and 4.3. The meaning of the columns is as follows (from left to right): name of the benchmarks, transitions that are WF enabled ('✓' means that the transition is WF enabled, otherwise is not, '×'), verification time, verification time using the modular representation of the verifier, and whether the system is diagnosable. Again, the

Benchmarks	WF enabled *skip_healthy*	WF enabled *fix*	Vrf Time	Vrf Modular Time	Diagnosable
CommBox (4)	✓	×	1	<1	×
	×	✓	1	<1	×
CommBox (5)	✓	×	1	1	×
	×	✓	2	1	×
CommBox (6)	✓	×	2	1	×
	×	✓	2	2	×
CommBox (7)	✓	×	2	2	×
	×	✓	3	2	×

Table 4.2: Experimental results for CommBox with reduced WF set.

time is measured in seconds.

The Maria tool confirmed that this is the case for the transitions *skip_healthy* and *fix* for the CommBox family, and for the transitions *skip_healthy, report* and *fix* for the CommBoxTech family. However, surprisingly, the CommBoxTech benchmarks remain diagnosable even when the *skip_reported* transition is removed from the WF set: This is in fact correct, as *skip_reported* can be enabled only after some fault has been reported, i.e. some fault will be diagnosed due to the *fix* transition even if *skip_reported* never fires.

4.7 General verifier for bounded LPNs

Let $\mathcal{N}$ be a bounded LPN whose fault transitions may or may not be WF. The consequences of having WF faults are exemplified by Fig. 4.4. Notice that in both $\mathcal{V}$ and $\mathcal{V}_{WF}$, the fault transitions are removed from the component corresponding to $\mathcal{N}'$. However, in Fig. 4.4, the fault transition is enabled in the initial marking and WF, hence unavoidable, so the LPN is trivially WF-diagnosable. However, both $\mathcal{V}$ and $\mathcal{V}_{WF}$ violate their respective LTL-X formulae in this example since they allow $\mathcal{N}'$ to ignore the permanently enabled WF fault transition.

To fix this problem, the construction of $\mathcal{V}_{WF}$ is extended as explained below. The generalised construction adds several places called *control monitors*. These will be added to the presets and postsets of certain transitions, which allows one

Benchmarks	WF enabled				Vrf Time	Vrf Modular Time	Diagnosable
	skip_healthy	*report*	*skip_reported*	fix			
CommBoxTech (4)	✓	✓	✓	×	2	1	×
	✓	✓	×	✓	17	6	✓
	✓	×	✓	✓	1	1	×
	×	✓	✓	✓	8	3	×
CommBoxTech (5)	✓	✓	✓	×	3	2	×
	✓	✓	×	✓	102	30	✓
	✓	×	✓	✓	2	1	×
	×	✓	✓	✓	42	14	×
CommBoxTech (6)	✓	✓	✓	×	3	2	×
	✓	✓	×	✓	560	147	✓
	✓	×	✓	✓	2	1	×
	×	✓	✓	✓	6	61	×
CommBoxTech (7)	✓	✓	✓	×	4	3	×
	✓	✓	×	✓	2853	Bug	✓
	✓	×	✓	✓	3	2	×
	×	✓	✓	✓	1099	4	×

Table 4.3: Experimental results for CommBoxTech with reduced WF set.

to distinguish different phases of an execution. While such a transformation does not change the behaviour of a standard Petri net, the presence of weak fairness requires a more elaborated construction. To illustrate this issue, suppose t is WF and t' is another transition with ${}^\bullet t \cap {}^\bullet t' = \emptyset$. Thus, if t is enabled then firing t' does not discharge an infinite WF execution from the obligation to fire a transition from $({}^\bullet t)^\bullet$. However, if the same control monitor place is added to the presets of t and t', the set of WF executions can change. To solve this problem we introduce the following notion.

Definition 13. A *WF-preserving control monitor mon w.r.t. a set of transitions* T is a set of fresh places $\{mon_0\} \cup \{\, mon_t \mid t \in T \text{ is WF} \,\}$.

We use the following net operations involving control monitors.

(Simple) control: Let T be a set of transitions and p, p' be two control monitor places (which may be the same). To *control* T by (p, p') means adding p to the preset and p' to the postset of each $t \in T$.

WF-preserving control: Let T be a set of transitions and mon, mon' be two WF-preserving control monitors. We say that T is controlled by (mon, mon') if we control each non-WF transition in T by (mon_0, mon_0') and each WF transition $t \in T$ by (mon_t, mon_t').

Activity monitor: Let T be a set of transitions and $T_{WF} \stackrel{\text{df}}{=} \{t \in T \mid t \text{ is WF}\}$. An *activity monitor* is used to make sure that an execution contains infinitely many occurrences of transitions from T. Formally, an activity monitor a is a tuple of WF-preserving control monitors $(idle^a, active^a)$ that controls T, where all places of $idle^a$ are initially marked. Moreover, we add a number of fresh WF transitions: one that transfers a token from $active_0^a$ to $idle_0^a$, and one for each $t \in T_{WF}$ that transfers a token from $active_t^a$ to $idle_t^a$. With a we associate the LTL-X formula

$$\phi_a \stackrel{\text{df}}{=} \Box \Diamond \left(active_0^a \vee \bigvee_{t \in T_{WF}} active_t^a \right)$$

expressing that an execution contains infinitely many occurrences of T.

The ideas behind the general verifier $\mathcal{V}_{WF}^{gen}$ capable of handling WF faults are as follows. It has three copies of the original LPN $\mathcal{N}$, corresponding to the executions σ, ρ, and the infinite fault-free WF continuation ρ' of $\hat{\rho}$ as in the proof of Lemma 4.3.2. The first two copies are transformed and synchronised as before and correspond to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, to ensure that σ is WF, ρ is fault-free and $\ell(\sigma) = \ell(\rho)$. The LTL-X formula, as before, ensures that σ contains a fault. The third copy, $\mathcal{N}''$, initially follows $\mathcal{N}'$, in the sense that any transition modifying the marking of $\mathcal{N}'$ also modifies the marking of $\mathcal{N}''$ in the same way, so that the markings of $\mathcal{N}'$ and $\mathcal{N}''$ are the same for some time. Moreover, a separate set of places $\bar{P}$ corresponding to those in $\mathcal{N}'$ is created, and it is ensured in a similar way that the marking of $\bar{P}$ is the same as that of $\mathcal{N}'$ and $\mathcal{N}''$ for the same period of time. However, at some non-deterministically chosen point of time, $\mathcal{N}''$ starts running completely independently from $\mathcal{N}'$ and the marking of $\bar{P}$ stops changing (the projection of the verifier's execution up to this point to $\mathcal{N}'$ corresponds to $\hat{\rho}$). The non-WF fault transitions of $\mathcal{N}''$ are removed, and its WF fault transitions are turned into stubs, and the LTL-X formula will ensure that these stubs do not fire and the projection of the execution onto this LPN is WF. Moreover, the formula will ensure that ρ periodically passes through the marking stored in $\bar{P}$ as required by Lemma 4.3.2 (2). For this, we employ the LTL-X formula

$$\phi_{mark} \stackrel{\text{df}}{=} \Box \Diamond \bigwedge_{(p', \bar{p})} \#_{tok}(p') = \#_{tok}(\bar{p}),$$

where p' runs through the places of $\mathcal{N}'$ and $\bar{p}$ is the place of $\bar{P}$ corresponding to p'. Note that this formula uses elementary propositions checking whether two places contain the same number of tokens; such comparisons are supported by mainstream model checkers, e.g. MARIA. Lastly, two *activity monitors* are added to this construction, to check whether the projections of any execution of the verifier satisfying the formula to $\mathcal{N}'$ and $\mathcal{N}''$ are infinite (this is unlike the construction of $\mathcal{V}_{WF}$ in Sect. 4.4.3, where the infinity of projections followed automatically from the assumptions about $\mathcal{N}$). This implies the infinity of the projection to $\mathcal{N}^{ft}$, due to the assumptions about $\mathcal{N}$.

We now describe the verifier's construction in more detail:

1. Construct the net $\mathcal{V}_{WF}$ as in Sect. 4.4.3, containing the two copies $\mathcal{N}^{ft}$ and $\mathcal{N}'$ and a place *stub_monitor*. Let $\mathcal{N}''$ be an additional copy of $\mathcal{N}$. Add a fresh set of places $\bar{P} \stackrel{\text{df}}{=} \{\, \bar{p} \mid p \text{ is a place of } \mathcal{N}' \,\}$.

2. Remove from $\mathcal{N}''$ all non-WF fault transitions, and turn the WF fault transitions into stubs (re-using *stub_monitor*); they remain WF.

3. Add an initially marked control monitor place *in_prefix* and a WF-preserving monitor *after_prefix* w.r.t. the non-stub transitions T of $\mathcal{N}''$. Control the transitions in T by (*after_prefix*, *after_prefix*). Moreover, add a fresh non-WF transition *switch* with the preset $\{$*in_prefix*$\}$ and the postset *after_prefix*.

4. Let T_{after} be the transitions of $\mathcal{N}'$, including the fused transitions. Add a set T_{in} containing a fresh non-WF copy t' for each $t \in T_{after}$ with the same label. For each place p of $\mathcal{N}'$ in ${}^\bullet t$ (resp. $t^\bullet$), add p and the corresponding places p'' of $\mathcal{N}''$ and $\bar{p} \in \bar{P}$ to the preset (resp. postset) of t'. Furthermore, T_{in} is controlled by (*in_prefix*, *in_prefix*) and T_{after} by (*after_prefix*$_0$, *after_prefix*$_0$).

5. Introduce an activity monitor a' that watches the fused transitions among T_{after}.

6. Introduce an activity monitor a'' that watches all the visible transitions of $\mathcal{N}''$.

7. Call the resulting net $\mathcal{V}^{gen}_{WF}$.

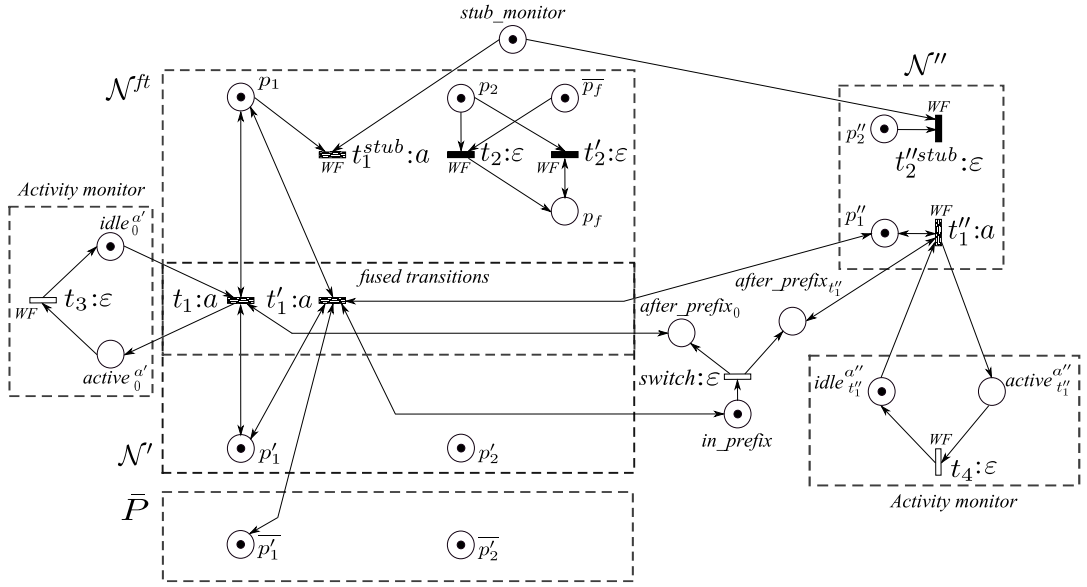

Figure 4.15: The general verifier $\mathcal{V}_{WF}^{gen}$ capable of handling WF faults for the LPN in Fig. 4.4.

This construction is illustrated in Fig. 4.15.

As before, we formulate diagnosability of $\mathcal{N}$ as an LTL-X formula that needs to hold for the infinite WF executions of $\mathcal{V}_{WF}^{gen}$:

$$diag_{WF}^{gen} \stackrel{\text{df}}{=} \Box\,\overline{p_f} \vee \Diamond\,\neg stub\_monitor \vee \neg\phi_{mark} \vee \neg\phi_{a'} \vee \neg\phi_{a''}.$$

The negation of this formula is

$$\neg diag_{WF}^{gen} = \Diamond\,\neg\overline{p_f} \wedge \Box\, stub\_monitor \wedge \phi_{mark} \wedge \phi_{a'} \wedge \phi_{a''}.$$

A counterexample thus has to commit a fault in $\mathcal{N}^{ft}$ (but not in $\mathcal{N}'$ as fault transitions are removed from there), not execute any stub transitions, contain infinitely many synchronised transitions and infinitely many visible transitions of $\mathcal{N}''$. Additionally, the $\mathcal{N}'$ part of $\mathcal{V}_{WF}^{gen}$ must pass through the marking stored in $\bar{P}$ infinitely often.

Theorem 4.7.1 (Correctness of general WF Verifier)**.** *A bounded LPN $\mathcal{N}$ is diagnosable iff the corresponding verifier $\mathcal{V}_{WF}^{gen}$ satisfies $diag_{WF}^{gen}$.*

Proof. ($\Rightarrow$) Suppose $\mathcal{V}^{gen}_{WF}$ does not satisfy $diag^{gen}_{WF}$, i.e. it has a WF execution τ satisfying $\neg diag^{gen}_{WF}$. Hence τ satisfies $\phi_{a'}$ and thus contains infinitely many synchronised transitions. Let σ and ρ be the projections of τ to $\mathcal{N}^{ft}$ and $\mathcal{N}'$, respectively (the latter projection includes all fused transitions, including both T_{in} and T_{after}). We claim that (σ, ρ) is a witness. Indeed, as in the proof of Thm. 4.4.1 we obtain that σ is an infinite WF faulty execution of $\mathcal{N}$. It remains to prove that ρ satisfies the condition of Lemma 4.3.2 (2). Clearly, ρ is fault-free by construction. Also, $\phi_{a'}$ implies that τ contains *switch*. Let $\hat{\rho}$ be a finite prefix of ρ before the occurrence of *switch*, and m be the marking of $\mathcal{N}'$ reached by $\hat{\rho}$. (A copy of m is stored in $\bar{P}$ and is never changed after this point in time.) Then ρ goes through m infinitely often because τ satisfies ϕ_{mark}. Let ρ' be the projection of τ to the transitions of $\mathcal{N}''$. These are controlled by *after_prefix*, hence describe an execution starting at m. That execution is infinite because $\phi_{a''}$ is satisfied; it is WF because τ is, and it is fault-free because no stub has fired. Thus, (σ, ρ) is indeed a witness.

($\Leftarrow$) Let (σ, ρ) be a witness according to Def. 12. According to Lemma 4.3.2, ρ goes infinitely often through some marking m, and a prefix $\hat{\rho}$ reaching m can be extended to an infinite WF fault-free execution $\hat{\rho}\rho'$. Such a witness can be transformed into an execution τ of $\mathcal{V}^{gen}_{WF}$ satisfying $\neg diag^{gen}_{WF}$ as follows. Prefix $\hat{\rho}$ is simulated by the transitions in T_{in}. Upon reaching m, the verifier can fire *switch*. Up to this point, $\mathcal{N}'$ and $\mathcal{N}''$ have the same marking, which is m. Then, transitions from T_{after} are used to simulate the continuation of $\hat{\rho}$ in ρ, and $\mathcal{N}''$ is used to simulate its infinite WF continuation ρ'. By assumption, σ and ρ' are infinite and WF, hence so is τ; thus no stub needs to be fired, and $\phi_{a'} \wedge \phi_{a''}$ are satisfied. Since σ contains a fault, τ satisfies $\Diamond \neg \overline{p_f}$. Also, ρ passes through m infinitely often, thus satisfying ϕ_{mark}. Hence $diag^{gen}_{WF}$ is violated by τ. □

4.7.1 Conclusion

In this chapter we have identified a major flaw in the previous definition of WF-diagnosability in the literature, and proposed a corrected notion. Based on weak diagnosis, a first definition of diagnosability under weak fairness was proposed in [1]. However, that definition is incompatible with the notion of diagnosis in [44]

and contains a major flaw. It is often the case that due to the presence of some independent concurrent action in a system, it is not possible this system to be diagnosed in a finite time. In addition, we provide two alternative characterisations of executions that witness violations of WF-diagnosability together with a proof of their equivalence.

Furthermore, the special case where fault transitions are not WF is further investigated, i.e. a fault is a *possible* outcome in the system but not one that is *required* to happen. The examples in Sect. 4.6 suggest that this is a reasonable assumption in practice. Under this assumption, the notion of a witness can be significantly simplified.

Moreover, under a simplifying assumption that the fault transitions are non-WF, we have presented an efficient technique for verifying WF-diagnosability based on a reduction to LTL-X model checking. The proposed approach works for a bounded LPN $\mathcal{N}$. We perform various operations on $\mathcal{N}$ to obtain another bounded LPN $\mathcal{V}$, the verifier, which we check against a *fixed* LTL-X formula. The construction of our verifier is based on the proposed in [58] which is based on Petri nets unfoldings. We have shown in Section 4.4.2 that the verifier in [58] is unsuitable for WF-diagnosability. Consequently, to address this issue in our proposed verifier we introduce and use stub transitions and a place the stub_monitor. Stubs are transitions that are not meant to be executed: in fact, our LTL-X formulae will make such executions 'irrelevant' by demanding that the place stub_monitor remains always marked. Moreover, an important advantage of this method is that the LTL-X formula is fixed — in particular, the WF assumption does not have to be expressed as a part of it (which would make the formula length proportional to the size of the specification), but rather the ability of existing model checkers to handle weak fairness directly is exploited. Furthermore, the construction has been generalised to arbitrary bounded LPNs.

Furthermore, we created two families of scalable benchmarks, where the weak fairness is essential for diagnosability. The proposed WF-diagnosability verification method has been tested on these benchmarks, and the experimental results demonstrate its feasibility in practice. For the verification, the MARIA (modular reachability analyser) tool [60] was used. Since MARIA supports modular verification, it was possible to exploit the modular structure of the verifier to

significantly speed up the verification. It should be noted that finding interesting benchmarks was a challenging task: Despite a lot of theoretical work done in the area of diagnosability, rather few practical experiments have been conducted. Moreover, we wanted benchmarks where weak fairness is essential, i.e. removing some transitions from the WF set would make the system undiagnosable. In Section 4.6, the CommBox and CommBoxTech are presented, together with their verifier. The former one models a system comprising commutator boxes and an inspector, and the latter is an elaborated version of the former, where the inspector reports the faults to a technician, who then fixes them. For future work we intend to develop a theory of diagnosability of systems with strong fairness.

Chapter 5

Conclusions

This thesis introduces a formal specification language that is suitable for modelling reference passing systems. Also, it provides its translation to an existing formalism for which an efficient verification technique has been developed in [52]. Moreover, a notion of weakly fair diagnosability [35, 36] which corrects and supersedes the one in [1], is presented and an efficient method for formally verifying weakly fair diagnosability is developed.

In Chapter 3, a new fragment of π-calculus, the EFCP, is presented. The initial motivation of this research was the development of a formalism allowing for convenient modelling and formal verification of RPS. The validation of such systems is almost always limited to simulation/testing, as their formal verification is very difficult due to either the inability of the traditional verification techniques to express reference passing (at least in a natural way) or by poor scalability of the existing verification techniques for RPSs. This is very unfortunate: As many safety-critical systems must be resilient (and hence reconfigurable), they are often RPSs and thus have very complicated behaviour. Hence, for such systems the design errors are both very likely and very costly, and formal verification must be an essential design step.

Thus, new more efficient formal languages should be developed that can specify reference passing systems and make their formal verification feasible. The π-calculus is the most well-known formalism suitable for RPS specification. EFCPs is an extension of the well-known fragment of π-calculus, the Finite Control Processes. FCPs were used for formal modelling of RPS; however, they cannot express

scenarios involving 'local' concurrency inside a process. Extended finite control processes remove this limitation. This is essential because the processes constitute RPSs often have 'local' concurrency. For instance, systems belong to IoT, cloud computing, routing protocols in multi-core processor systems etc., share multicasting as a common feature. As a result, practical modelling of mobile systems becomes more convenient.

An EFCP system specification, due to the local concurrency its processes may contain, can yield a very large state space. Thus, to define the EFCP syntax the notion of finite processes is required. Such processes have special syntax ensuring that the number of actions they can execute is bounded in advance. To this end, a more powerful sequential composition operator ' ; ' is used instead of prefixing. Furthermore, an almost linear translation from safe EFCP to safe FCP has been developed, which forms the basis of formal verification of RPSs. The purpose of translating EFCP to FCP is for the latter to be translated to safe low-level PN, for which efficient verification techniques can be applied.

In the translation we had to face two main challenges. Firstly, it has to eliminate the parallel composition operator iniside threads and the use of sequential composition. Since an FCP consists of sequential processes (threads), any thread of an EFCP that is not sequential must be converted to a sequential one. This has be done by shifting all the concurrency to the initial term. Moreover, sequential composition after the translation has been replaced by prefixing. To that end, to avoid blow up in size, new declarations are introduced during this process. Secondly, we have to ensure that the order of actions is preserved and that the context (binding of names) is correct. To address these issues, we realised that extra communication between threads may be required. As a result, new process definitions are introduced in two cases. The first is when local concurrency exists within a thread. The second case is when there is a sequential composition with a non-trivial left-had side.

Moreover, a formal definition of the translation that consists of several rules is defined and based on this formal definition the Efcp2Fcp tool that automates the translation has been implemented. The SpiNNaker case study demonstrates that EFCPs allow for a concise expression of multicast communication, and is suitable for practical modelling. The SpiNNaker's EFCP model was translated

to FCP and then to PN. Then, deadlock checking was performed to the PN model with the LoLA tool.

In our future work we intend to investigate the relationship between the transition systems generated by EFCPs and those generated by the corresponding FCPs, with the view to prove the correctness of the proposed translation. Moreover, it should be noted that for formally verifying the SpiNNaker architecture, we have used a small model (2x2). Although its small size, the resulting PN has significantly large size which leads to a big verification time as well. To that end, we would also like to evaluate the scalability of the proposed approach on a range of models and optimise the translation, e.g. by reducing the number of generated defining equations and by lifting it to non-safe processes. Moreover, we intend to develop a translation of Petri net to EFCP and the related tool. Thus, possible Petri net counter-examples, which may be created during model checking of RPS, can be translated back to the level of EFCP facilitating the design process of RPS. In addition, we would like to define a property language for specifying key correctness properties of RPS at the level of EFCP. Based on this RPS logic, another potential work is to translate properties of this new logic to equivalent properties of Petri nets.

In Chapter 4 diagnosability under the weak fairness assumption is considered, which intuitively states that no transition from a given set can stay enabled forever. We have identified a major flaw in the previous definition of WF-diagnosability in the literature, and proposed a corrected notion. Based on weak diagnosis, a first definition of diagnosability under weak fairness was proposed in [1]. However, that definition is incompatible with the notion of diagnosis in [44] and contains a major flaw. It is often the case that due to the presence of some independent concurrent action in a system, it is not possible this system to be diagnosed in a finite time. Furthermore, we provide two alternative characterisations of executions that witness violations of WF-diagnosability together with a proof of their equivalence.

In addition, we characterise executions that witness violations of WF diagnosability. We further investigate the special case where fault transitions are not WF, that is, a fault is a possible outcome in the system but not one that is required to happen. Under this assumption, the notion of a witness can be

significantly simplified. Moreover, under a simplifying assumption that the fault transitions are non-WF, an efficient technique for verifying WF-diagnosability based on a reduction to LTL-X model checking, which is based on PN unfoldings [58], has been presented. The proposed approach works for a bounded LPN $\mathcal{N}$. We perform various operations on $\mathcal{N}$ to obtain another bounded LPN $\mathcal{V}$, the verifier, which we check against a *fixed* LTL-X formula. The construction of our verifier is based on the proposed in [58].

An important advantage of this method is that the LTL-X formula is fixed — in particular, the WF assumption does not have to be expressed as a part of it (which would make the formula length proportional to the size of the specification), but rather the ability of existing model checkers to handle weak fairness directly is exploited. Furthermore, the construction has been generalised to arbitrary bounded LPNs.

To evaluate the proposed technique, two families of scalable benchmarks were developed, where the weak fairness is essential for diagnosability. The proposed WF-diagnosability verification method has been tested on these benchmarks, and the experimental results demonstrate its feasibility in practice. For the verification, the MARIA (modular reachability analyser) tool [60] was used. Since MARIA supports modular verification, it was possible to exploit the modular structure of the verifier to significantly speed up the verification. It should be noted that finding interesting benchmarks was a challenging task: Despite a lot of theoretical work done in the area of diagnosability, rather few practical experiments have been conducted. Moreover, we wanted benchmarks where weak fairness is essential, i.e. removing some transitions from the WF set would make the system undiagnosable. In Section 4.6, the COMMBOX and COMMBOXTECH are presented, together with their verifier. The former one models a system comprising commutator boxes and an inspector, and the latter is an elaborated version of the former, where the inspector reports the faults to a technician, who then fixes them. For future work we intend to develop a theory of diagnosability of systems with strong fairness. Moreover, we would like to apply and evaluate the WF-diagnosability in systems that comply with the Bell-Lapadula security policy (i.e., federated clouds [99]).

References

[1] A. Agarwal, A. Madalinski, and S. Haar. Effective verification of weak diagnosability. In *Proc. SAFEPROCESS'12*. IFAC, 2012. vii, 2, 11, 59, 60, 61, 63, 65, 68, 90, 93, 95

[2] S. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978. 15

[3] K. Asanovic, J. Beck, J. Feldman, N. Morgan, and J. Wawrzynek. A supercomputer for neural computation. In *Neural Networks, World Congress on Computational Intelligence*, volume 1, pages 5–9. IEEE, 1994. 46

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008. 17, 19, 20, 21, 22

[5] S. Bavishi and E. Chong. Automated fault diagnosis using a discrete event systems framework. In *Proceedings of the International Symposium on Intelligent Control*, pages 213–218. IEEE, 1994. 5, 59

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Tech Rep. MITRE Corporation, 1973. 10

[7] A. Benveniste, E. Fabre, S. Haar, and C. Jard. Diagnosis of asynchronous discrete event systems: a net unfolding approach. *Automatic Control IEEE Transactions on*, 48(5):714–727, 2003. 2, 9

[8] E. Best, R. Devillers, and M. Koutny. The Box algebra=Petri nets+process expressions. *Information and Computation*, 178(1):44–100, 2002. 8, 32

[9] E. Best and C. Fernandez. Non-sequential processes. volume 13 of *EATCS Monographs*, pages 95–115. Springer New York, 1988. 9

[10] W. Brauer, W. Reisig, and G. Rozenberg. Advances in petri nets. In *Proceedings of an Advanced Course on Petri Nets, Central Models and Their Properties*. Springer-Verlag, 1987. 7, 28

[11] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. 15

[12] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the 1991 International Conference on VLSI*, pages 49–58, 1991. 15

[13] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. 15

[14] J. Camara, M. Moreto, E. Vallejo, R. Beivide, J. Miguel-Alonso, C. Martinez, and J.Navaridas. Mixed-radix twisted torus interconnection networks. In *Parallel and Distributed Processing Symposium IPDPS*, pages 1–10. IEEE, 2007. 46

[15] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin Heidelberg, 1998. 4

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194. Springer-Verlag, 2001. 15, 16

[17] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. 1, 17

[18] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999. 1, 14

[19] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373. Springer Verlag New York Inc., 1990. 15

[20] M. Dam. Model checking mobile processes. *Inf. Comp.*, 129(1):35–51, 1996. 1, 3, 4, 23, 31

[21] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, 1965. 19

[22] E. W. Dijkstra. The origin of concurrent programming. chapter Cooperating Sequential Processes, pages 65–138. Springer-Verlag New York, Inc., 2002. 19

[23] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN, pages 37–56. Springer-Verlag New York, Inc., 2001. 11

[24] J. Esparza and K. Heljanko. *Unfoldings: A Partial-Order Approach to Model Checking*. Springer, 2008. 11

[25] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *FMSD*, 20(3):285–310, 2002. 11

[26] J. Ezekiel and A. Lomuscio. An automated approach to verifying diagnosability in multi-agent systems. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 51–60. IEEE Computer Society, 2009. 59

[27] E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *J. of Discr. Event Syst.*, pages 33–84, 2005. 11

[28] P. Farber and K. Asanovic. Parallel neural network training on multi-spert. In *Algorithms and Architectures for Parallel Processing, ICAPP, 3rd International Conference on*, pages 659–666. IEEE, 1997. 46

[29] G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003. 7

[30] M. J. Fischer and R. E. Ladner. Propositional modal logic of programs. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC, pages 286–294. ACM, 1977. 18

[31] S. Furber and A. Brown. Biologically inspired massively parallel architectures computing beyond a million processors. In *Application of Concurrency to System Design, 2009. ACSD'09*, pages 3–12. IEEE, 2009. vii, 47

[32] S. Furber and S. Temple. Neural systems engineering. In J. Fulcher and L. Jain, editors, *Computational Intelligence: A Compendium*, volume 115 of *Studies in Computational Intelligence*, pages 763–796. Springer Berlin Heidelberg, 2008. 46

[33] S. Furber, S. Temple, and A. Brown. On-chip and inter-chip networks for modeling large-scale neural systems. In *Circuits and Systems. ISCAS Proceedings*, pages 21–24. IEEE, 2006. 46

[34] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin Heidelberg, 2001. 21, 22

[35] V. Germanos, S. Haar, V. Khomenko, and S. Schwoon. Diagnosability under weak fairness. *ACM Transactions on Embedded Computing Systems*. Special Issue on Best Papers from ACSD'14, submitted paper. 2, 11, 12, 61, 71, 79, 93

[36] V. Germanos, S. Haar, V. Khomenko, and S. Schwoon. Diagnosability under weak fairness. In *Proc. ACSD'14*, pages 132–141. IEEE Computing Society Press, 2014. 2, 6, 11, 12, 59, 61, 71, 79, 93

[37] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., 1996. 17

[38] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982. 9

[39] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984. 9, 10

[40] U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983. 9

[41] M. Gomez, N. Nordbotten, J. Flich, P. Lopez, A. Robles, J. Duato, T. Skeie, and O. Lysne. A routing methodology for achieving fault tolerance in direct networks. *Computers, IEEE Transactions on*, 55(4):400–415, 2006. 47

[42] S. Haar. Qualitative diagnosability of labeled Petri nets revisited. In *Proc. CDC'09 & CCC'09*, pages 1248–1253. IEEE Control Syst. Soc., 2009. 11, 59

[43] S. Haar, A. Benveniste, E. Fabre, and C. Jard. Partial order diagnosability of discrete event systems using Petri nets unfoldings. In *CDC'03*, 2003. 11, 59

[44] S. Haar, C. Rodríguez, and S. Schwoon. Reveal your faults: It's only fair! In *Proc. ACSD'13*, pages 120–129. IEEE Computer Society Press, 2013. 2, 6, 11, 12, 59, 60, 90, 95

[45] K. Heljanko, V. Khomenko, and M. Koutny. Parallelisation of the Petri net unfolding algorithm. In *Proc. TACAS'02*, volume 2280 of *LNCS*, pages 371–385. Springer, 2002. 11

[46] M. Huth and M. Ryan. *Logic in Computer Science Modelling and Reasoning about Systems.* CUP, 2004. 16, 18, 21

[47] S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete event systems. In *IEEE Trans. on Autom. Control*, 2001. 10, 59

[48] X. Jin, S. Furber, and J. Woods. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *Neural Networks, IJCNN World Congress on Computational Intelligence*, pages 2812–2819. IEEE, 2008. 46

[49] V. Khomenko. PUNF homepage. URL: http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/tools.html, 2012. 11

[50] V. Khomenko and V. Germanos. Modelling and analysis mobile systems using π-calculus (EFCP). In *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, volume 9410 of *Lecture Notes in Computer Science*. Spinger Verlag, 2015. 5, 8, 12

[51] V. Khomenko, M. Koutny, and A. Niaouris. Applying Petri net unfoldings for verification of mobile systems. In *Proc. of MOCA*, Bericht FBI-HH-B-267/06, pages 161–178. University of Hamburg, 2006. 7

[52] V. Khomenko, R. Meyer, and R. Hüchting. A polynomial translation of π-calculus (FCP) to safe Petri nets. *Logical Methods in Computer Science*, 9(3):1–36, 2013. 2, 9, 12, 23, 25, 28, 31, 56, 93

[53] M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989. 19

[54] M. Z. Kwiatkowska, T. Rodden, V. Sassone, and editors. From computers to ubiquitous computing by 2020. In *Proc. of Philosophical Transactions of the Royal Society*, volume 366, 2008. 3

[55] L. Lamport. What good is temporal logic. In *Information Processing '83*, volume 4709. Elsevier Science, 1983. 18, 21, 69

[56] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In S. Even and O. Kariv, editors, *Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer Berlin Heidelberg, 1981. 19

[57] F. Lin and W. Wonham. On observability of discrete-event systems. *Inf Sci*, 44(3):173–198, 1988. 6, 59

[58] A. Madalinski and V. Khomenko. Diagnosability verification with parallel LTL-X model checking based on Petri net unfoldings. In *Control and Fault-Tolerant Systems (SysTol), 2010 Conference on*, pages 398–403. IEEE Computer Society Press, 2010. 11, 59, 62, 70, 71, 72, 73, 76, 77, 78, 91, 96

[59] A. Madalinski, F. Nouioua, and P. Dague. Diagnosability verification with Petri net unfoldings. In *KES'09*, LNCS/LNAI. Springer, 2009. 11, 59

[60] maria. MARIA: The modular reachability analyzer, 2005. URL: `http://www.tcs.hut.fi/Software/maria/ index.en.html`. 82, 91, 96

[61] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV'92*, pages 164–177, jul 1992. 11

[62] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 15

[63] R. Meyer, V. Khomenko, and R. Hüchting. A polynomial translation of π-calculus (FCP) to safe Petri nets. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012 Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 440–455. Springer Berlin Heidelberg, 2012. 2, 8, 12, 31

[64] R. Meyer, V. Khomenko, and T. Strazny. A practical approach to verification of mobile systems using net unfoldings. In *Proc. of ATPN*, volume 5062 of *LNCS*, pages 327–347. Springer, 2008. 7

[65] R. Meyer, V. Khomenko, and T. Strazny. A practical approach to verification of mobile systems using net unfoldings. *Fundam. Inf.*, 94(3–4):439–471, 2009. 4, 5, 23, 27, 35

[66] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999. 22, 26

[67] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Inf. Comp.*, 100(1):1–40, 1992. 4

[68] B. Moret. Decision trees and diagrams. *ACM Comput. Surv.*, 14(4):593–623, 1982. 16

[69] J. Navaridas, M. Luján, J. Miguel-Alonso, L. A.Plana, and S. Furber. Understanding the interconnection network of SpiNNaker. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 286–295. ACM, 2009. 46, 48

[70] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981. 9

[71] F. Peschanski, H. Klaudel, and R. Devillers. A Petri net interpretation of open reconfigurable systems. In *Proc. of Petri nets'11*, volume 6709, pages 208–227. Springer, 2011. 7

[72] C. A. Petri. Kommunikation mit Automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1, 1966. 28

[73] V. V. Pham. *Modelling and Analysing Open Reconfigurable Systems*. PhD thesis, University of Evry, IBISC, 2014. 7

[74] M. Pistore. *History Dependent Automata*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999. 7

[75] C. Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, pages 54–64. Springer Verlag, 1991. 15

[76] C. Pixley, G. Beihl, and E. P. Skewes. Automatic derivation of fsm specification to implementation encoding. In *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, 1991. 15

[77] C. Pixley, S. Jeong, and G. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th ACM. IEEE Design Automation Conference*, pages 620–623. IEEE Computer Society Press, 1992. 15

[78] L. Plana, J. Bainbridge, S. Furber, S. Salisbury, S. Yebin, and W. Jian. An on-chip and inter-chip communications network for the SpiNNaker massively-parallel neural net simulator. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM IEEE International Symposium on*, pages 215–216. IEEE Computer Society, 2008. 46

[79] L. Plana, S. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS infrastructure for a massively parallel multiprocessor. *Design Test of Computers, IEEE*, 24(5):454–463, 2007. 46, 47

[80] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57. IEEE Computer Society, 1977. 20, 69

[81] F. Pommereau. SNAKES homepage. URL: `http://code.google.com/p/python-snakes`, 2014. 7

[82] V. Puente and J. Gregorio. Immucube: scalable fault-tolerant routing for k-ary n-cube networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):776–788, 2007. 47

[83] V. Puente, C. Izu, R. Beivide, J. A. Gregorio, F. Vallejo, and J. M.Prellezo. The adaptive bubble router. *J. Parallel Distrib. Comput.*, 61(9):1180–1208, 2001. 47

[84] J. Queille and J. Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Inf.*, 19(3):195–220, 1983. 19

[85] A. Rast, S. Yang, M. Khan, and S. Furber. Virtual synaptic interconnect using an asynchronous network-on-chip. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*, pages 2727–2734. IEEE, 2008. 46

[86] W. Reisig. *Petri nets: An Introduction*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1985. 14

[87] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete events systems. *IEEE Trans. on Autom. Control*, 40(9):1555–1575, 1995. 1, 5, 9, 10, 59, 62

[88] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. CUP, 2001. 3, 22, 24, 26, 41

[89] C. Schröter and V. Khomenko. Parallel LTL-X model checking of high-level Petri nets based on unfoldings. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 109–121. Springer Berlin Heidelberg, 2004. 11

[90] A. Schumann and Y. Pencolé. Scalable diagnosability checking of event-driven systems. In *Proc. IJCAI'07*, pages 575–580, 2007. 10, 59

[91] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin Heidelberg, 1998. 7, 8, 10

[92] M. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *Proc. LICS'86*, 1st Symposium in Logic in Computer Science, pages 322–331. IEEE Computer Society Press, 1986. 21, 22

[93] B. Victor and F. Moller. The mobility workbench - a tool for the π-calculus. In *PROC. OF CAV'94*, volume 818, pages 428–440. Springer, 1994. 7

[94] W. Vogler. Fairness and partial order semantics. *Inf. Process. Lett.*, 55(1):33–39, 1995. 59, 63

[95] D. von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *European Symposium on Research in Computer Security*, pages 225–243. Springer Verlag, 2004. 10

[96] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995. 28

[97] R. Wolfgang. *Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985. 1, 5, 7, 14, 28

[98] J. Wu and S. Furber. A multicast routing scheme for a universal spiking neural network architecture. *Comput. J.*, 53(3):280–288, 2010. vii, 46, 47, 48, 49

[99] W. Zeng, M. Koutny, and P. Watson. Verifying secure information flow in federated clouds. In *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014*, pages 78–85. IEEE, 2014. 96