Pós-Graduação em Ciência da Computação

Rodrigo Benedito Otoni

A Strategy for Local Analysis of Determinism

Recife

2018

Rodrigo Benedito Otoni

A Strategy for Local Analysis of Determinism

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Augusto Cezar Alves Sampaio
Cooreintadora: Ana Lúcia Caneca Cavalcanti

Recife
2018

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O88s Otoni, Rodrigo Benedito
A strategy for local analysis of determinism / Rodrigo Benedito Otoni. – 2018.
81 f.: il., fig.

Orientador: Augusto Cezar Alves Sampaio.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências.

1. Engenharia de software. 2. Linguagem de programação. I. Sampaio, Augusto Cezar Alves (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2018-102

Rodrigo Benedito Otoni

A Strategy for Local Analysis of Determinism

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 17/07/2018.

BANCA EXAMINADORA

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática / UFPE

_____

Prof. Dr. Rohit Gheyi
Departamento de Sistemas e Computação / UFCG

_____

Prof. Dr. Augusto Cézar Alves Sampaio
Centro de Informática / UFPE
(**Orientador**)

I dedicate this work to my parents, without whom none of this would have been possible.

ACKNOWLEDGEMENTS

I would like to thank all those that supported me during the course of my master's degree.

I am deeply indebted to my supervisor, Professor Augusto Sampaio, and my Co-Supervisor, Professor Ana Cavalcanti, for the opportunity to work with them, for all the help and counselling, and for always pushing me to my limit.

My experience at the Centro de Informática was a wonderful one, allowing me to grow both professionally and personally, and for that I thank all faculty members and colleagues with whom I had the pleasure to work with. In special, I thank professors Alexandre Mota, and Gustavo Carvalho, for their collaboration and support, and Madiel Conserva Filho, Joabe Jesus Júnior, André Didier, Pedro Antonino, Filipe Arruda, and Flávia Falcão, for all the helpful discussions.

For the financial support given, I thank FACEPE, which provided me with a master's scholarship (grant IBPG-0074-1.03/16), and INES, which aided me with the payment of event registration fees (grants CNPq/465614/2014-0, and FACEPE/APQ/0388-1.03/14).

For introducing me to the life of research, for always providing me with a friendly advice when needed, and for setting me up to do a master's in the first place, I thank Professor Leila Silva, to whom I will be forever grateful.

Last, but definitely not least, I thank my parents, for the unconditional support they have always given me, and for always driving me to do me best in all circumstances.

ABSTRACT

Nondeterminism is an inevitable constituent of any theory that describes concurrency. For the validation and verification of concurrent systems, it is essential to investigate the presence or absence of nondeterminism, just as much as it is in the case of properties such as deadlock and livelock. CSP is a well established process algebra that offers rich semantic models, capable of capturing a wide range of sources of nondeterminism. The approach taken by the main tool for practical use of CSP, the model checker FDR, it to check for determinism through global analysis, which limits its scalability. In this dissertation we propose a local analysis strategy to check for determinism in specifications written in a practical subset of CSP. Our goal is to provide an efficient and scalable method of checking for determinism. We use a compositional approach in which we start from basic deterministic processes and check whether any of the composition operators used in the specification can introduce nondeterminism. The use of controlled subsets of selected notations is a common feature of local analysis, with the subset of CSP captured by our strategy containing most of the main operators of CSP, and thus being capable of modelling real world systems. Furthermore, our strategy is sound, according to our empirical evaluation, but not complete; giving up completeness is also a usual compromise of compositional approaches to analysis, as a way to improve efficiency. We present here our strategy, the prototype developed to allow its automation, and the results of a number of experiments. There are two main aspects of our strategy: its metadata, and its algorithms. After a process of the CSP specification is checked to be deterministic, we gather metadata about it. The metadata stores all the information of a process that is relevant to our strategy, and is the only element used when checking further compositions. For each composition operator available in our subset of CSP, we have developed a specific algorithm to check if the composition is deterministic. By the use of metadata, we remove the need to check the operands at each composition, relying only on the information previously gathered, and thus achieving an efficient compositional approach. A number of case studies, both toy problems and systems described in the literature, have been performed. We compared our prototype with FDR in all the experiments. For most examples our prototype is capable of analysing instances that FDR is not able to, due to lack of memory resulting from the state explosion. In some cases, our prototype is capable of analysing instances up to three orders of magnitude higher. For most instances in which both tools provide a result, besides the trivial ones, our prototype is more efficient than FDR, with some cases where FDR takes more than twenty minutes to reach a result, and our prototype requires only a few seconds.

Keywords: Model Checking. CSP. FDR. Performance. Experiments.

RESUMO

Não determinismo é um constituinte inevitável de qualquer teoria que descreva concorrência. Para a validação e verificação de sistemas concorrentes, é essencial que se investigue a presença ou a ausência de não determinismo, tanto quanto de outras propriedades, como deadlock e livelock. CSP é uma álgebra de processos bem estabelecida e que oferece ricos modelos semânticos, capazes de capturar uma grande variedade de fontes de não determinismo. A abordagem utilizada pela principal ferramenta de CSP, o verificador de modelos FDR, é verificar determinismo através de uma análise global, o que limita a sua escalabilidade. Nesta dissertação nós propomos uma estratégia de análise local de determinismo para especificações escritas em um subconjunto de CSP. Nosso objetivo é prover um método eficiente e escalável de verificação de determinismo. Nós usamos uma abordagem composicional, partindo de processos determinísticos básicos, e verificando se os operadores de composição usados na especificação podem introduzir não determinismo. O uso de subconjuntos controlados de notações é comum em estratégias de análise local, sendo que o subconjunto de CSP capturado por nossa estratégia contêm os principais operadores de CSP, possibilitando a modelagem de sistemas reais. A nossa estratégia é correta, segundo nossos experimentos, mas não completa; abrir mão de completude é uma decisão comum em estratégias de análise composicional, como uma forma de aumentar a eficiência. Nós apresentamos aqui a nossa estratégia, o protótipo desenvolvido para permitir a sua automação, e os resultados de vários experimentos. Nossa estratégia tem dois elementos principais: os seus metadados, e os seus algoritmos. Após um processo de uma especificação ser verificado como determinístico, nos coletamos metadados sobre ele. Os metadados armazenam todas as informações do processo que são relevantes para a estratégia, sendo o único elemento utilizado nas verificações seguintes. Para cada operador de composição disponível em nosso subconjunto de CSP, nós desenvolvemos um algoritmo específico para verificar se a composição é determinística. Pelo uso dos metadados, nós removemos a necessidade de verificar os operandos a cada composição, o que nos leva a uma abordagem composicional eficiente. Vários estudos de caso foram realizados, nos quais nós comparamos nosso protótipo com FDR. Para a maior parte dos experimentos nosso protótipo é capaz de analisar instâncias que FDR não consegue, devido a falta de memória causada pela explosão de estados. Em alguns casos, nosso protótipo é capaz de analisar instâncias até três ordens de magnitude maiores. Para a maioria das instâncias nas quais ambas as ferramentas geram um resultado, além das triviais, nosso protótipo é mais eficiente que FDR, com alguns casos em que FDR demora mais que vinte minutos para chegar a um resultado, e o nosso protótipo requer apenas alguns segundos.

Palavras-chaves: Verificação de Modelos. CSP. FDR. Performance. Experimentos.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

This dissertation proposes a local analysis strategy for determinism. The strategy checks specifications written in a subset of a well known process algebra, Communicating Sequential Process, or CSP for short, and is capable of efficiently analysing a number of real world problems. Its main advantage is its scalability, achieved by our compositional approach. Our experiments with the prototype that automates its application indicate the value of our contribution.

In this chapter we present the motivation of our work, in Section 1.1, together with our objectives, in Section 1.2, an overview of our strategy, in Section 1.3, and an outline of the dissertation, in Section 1.4.

1.1 Motivation

Concurrent systems are more common each passing day. These systems are harder to develop and to analyse than sequential ones. Nevertheless, the never ending quest to increase computational speed, be it through the use of multiple CPUs in one machine, or through distributed machines, together with the need to accurately model the concurrent world that we live in, make them, and all their traits, necessary (WATT, 2004).

When dealing with concurrency, we not only need to tackle the added complexity of having multiple components, each with its own individual state, but we also we need to consider problems that are exclusive to concurrent systems. The three classical properties of concurrency are deadlock, livelock, and nondeterminism.

The analysis of the classical properties of concurrency is crucial in the specification and design of concurrent systems. Deadlock and livelock are usually considered undesirable in all circumstances. Nondeterminism, however, is to be expected in abstract models of a wide range of systems, but may indicate problems in concrete designs. Verification techniques to investigate the presence or absence of all these properties in a model are essential for validation and verification of concurrent systems.

Deadlock and livelock have been investigated in depth, and there are very efficient tools available for their analysis. Among those, there is a well established model checker, FDR (GIBSON-ROBINSON et al., 2014), and prototypes implementing a number of techniques to check for deadlock, be it with CSP (RAMOS; SAMPAIO; MOTA, 2009; ANTONINO; SAMPAIO; WOODCOCK, 2014; ANTONINO; GIBSON-ROBINSON; ROSCOE, 2016a), CCS (FRANCESCA et al., 2011), or other notations (BENSALEM et al., 2011), and livelock (CONSERVA FILHO et al., 2016; CONSERVA FILHO et al., 2018), many using compositional approaches.

Of the three classical properties, nondeterminism is the one that has been less studied. It is, however, specially important in notations for refinement, where it is used for ab-

straction, being an inevitable constituent of any theory that describes concurrency where some form of arbitration is present (ROSCOE, 2010).

To model concurrent systems, process algebras are a possible choice, since they allow for high level modelling and formal reasoning about specifications. CSP is a well established process algebra that is accompanied by a set of robust tools that allow its practical use both in academia and in industry. In particular, CSP is capable of modelling both explicit and implicit nondeterminism, such as the ones that can be introduced by parallelism, internal communications, or renaming. Its versatility in modelling nondeterminism, together with its tool support, makes CSP a good choice for the analysis of determinism.

FDR (GIBSON-ROBINSON et al., 2014) is the main tool for practical use of CSP; it is a model checker that takes as input specifications in CSP_M, a machine readable version of CSP, and can, among other things, check for the presence of deadlock, livelock, and nondeterminism. Other tools for CSP, or CSP dialects, like ProB (LEUSCHEL; BUTLER, 2003), and PAT (SUN et al., 2009), also implement analysis strategies for these classical properties. The approach taken by all these tools for checking nondeterminism is, however, based on global analysis, where the entire model is expanded and exhaustively checked. In this dissertation, we propose a local analysis strategy for determinism, in order to improve both performance and scalability.

Local analysis has already been adopted for the verification of deadlock and livelock. Here, we present a local strategy for the verification of determinism in models written using a subset of CSP that includes most of its main operators, with some restrictions on how they can be used. Our strategy follows the CSP semantic model of failures, but is not strongly attached to any aspect of the language, and can possibly be extended to other formalisms. As far as we know, this is the first approach to local analysis of determinism, not only in the context of CSP, but also of any other formal modelling notation.

1.2 Objectives

The goal of this work is the creation of a local analysis strategy for determinism. To guide the development, six objectives are considered, listed below.

1. The strategy must be sound;
2. The strategy must be complete;
3. The analysis needs to be efficient and scalable;
4. As much of standard CSP as possible must be covered;
5. Real world systems must be verifiable;
6. Automation must be available.

Soundness is essential to allow the applicability of our strategy, since it provides guarantees to engineers about their designs. Together with soundness, completeness is also an important property for our verification strategy, since it allows the identification of nondeterministic systems, preventing the presence of false negatives. To allow for scalability, a compromise regarding these two properties must be made. We have relaxed our requirement of completeness to improve efficiency, while maintaining soundness. This is a common design decision among local analysis strategies, with many being sound, but not complete; examples include the works of Antonino, Sampaio and Woodcock (2014), for deadlock analysis, and of Conserva Filho et al. (2016), for livelock analysis. Our claim of soundness is only backed by empirical evaluation, without formal proofs, and this remains as an important weakness, to be addressed in future work.

Besides soundness and completeness, it is essential for our strategy to be efficient and scalable. This is the main contribution of our work, since there are already tools that guarantee correctness and are relatively efficient, besides implementing global analysis. We aim to match the guarantees provided by other tools, but provide a more efficient and, specially, a more scalable way of conducting analysis of determinism.

As is common in local analysis, we aim to cover a subset of our selected language. We provide a subset suitable for analysis of a number real world problems; this expressiveness is exemplified by some of the case studies developed. By focusing on a controlled subset of our choice, we can minimize considerably the verification effort.

Finally, the strategy must allow for automation, which is essential for its practical use. The prototype developed for this purpose automates the whole application of the strategy, including parsing, creation of metadata, and use of all algorithms. With our tool, we provide push button verification.

1.3 Strategy Overview

Our strategy receives as input a specification written in an accepted subset of CSP and outputs a message informing if the specification is deterministic or not. If there is the possibility of nondeterminism, it reports the specific point. The accepted subset includes most of the main operators of CSP, such as prefixing, external choice, internal choice, interleaving, generalized parallelism, and hiding; the complete subset is presented in Section 3.1. To illustrate the application of our strategy, we consider the following specification.

$$
\begin{aligned}
GetBox &= selectBox \rightarrow pickUpBox \rightarrow GetBox \\
MoveBox &= moveToDepot \rightarrow dropBox \rightarrow moveToDesk \rightarrow MoveBox \\
BrokenMoveBox &= moveToDepot \rightarrow moveToDesk \rightarrow dropBox \rightarrow BrokenMoveBox \\
WorkingRobot &= GetBox \,\Box\, MoveBox \\
BrokenRobot &= WorkingRobot \,\Box\, BrokenMoveBox
\end{aligned}
$$

In this simple specification we model a robot that receives boxes in a desk and moves them to a given depot, a common activity in a post office, for example. It consists of five processes, which are the main structures of CSP: *GetBox*, *MoveBox*, *BrokenMoveBox*, *WorkingRobot*, and *BrokenRobot*. The first three processes model the individual activities of the robot, with *GetBox* and *MoveBox* dealing with the normal behaviour, and *BrokenMoveBox* depicting an unexpected one. The process *WorkingRobot* models a robot that always works as expected, while *BrokenRobot* models a robot that can fail to deliver a box the depot. We assume that the robot is directed by an external system, and always picks up a box before moving to the depot.

To conduct the analysis, we divide the processes in two categories: Basic Processes, which are assumed to be deterministic, and Composite Processes, which are the result of the composition of Basic Processes and other Composite Processes. In our example, the processes *GetBox*, *MoveBox*, and *BrokenMoveBox*, are Basic Processes; they simply communicate certain events and then recurse. Processes *WorkingRobot* and *BrokenRobot* are Composite Processes, because they are the result of a composition of other processes, using the external choice operator.

Since the Basic Processes are expected to be deterministic, our strategy only needs to check if the compositions in the specification are sources of nondeterminism. The subset of CSP used guarantees that most Basic Processes will be deterministic by definition. When a guarantee cannot be obtained, an external verification needs to carried out before the application of the strategy.

When checking a composition, we use the metadata of its operands as inputs. The metadata of a process stores all the relevant information needed for our analysis, such that we do not need to analyse it when it is used in compositions. The first step of our strategy is to create the metadata of the Basic Processes, which is used to analyse the first compositions. The metadata, M, of the processes *GetBox*, *MoveBox*, and *BrokenMoveBox*, is shown below; the formal definition of the metadata, including the types of its internal structures, are presented in Section 3.2.

$$M(GetBox) = \left\langle \left\{ \left(\left\langle \left\{ \left\langle \begin{array}{l} selectBox, \\ pickUpBox, 0 \end{array} \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(MoveBox) = \left\langle \left\{ \left(\left\langle \left\{ \left\langle \begin{array}{l} moveToDepot, \\ dropBox, \\ moveToDesk, 0 \end{array} \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(BrokenMoveBox) = \left\langle \left\{ \left(\left\langle \left\{ \left\langle \begin{array}{l} moveToDepot, \\ moveToDesk, \\ dropBox, 0 \end{array} \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

The metadata consists of a sequence of sets of tuples, which have size three. The first element of a tuple stores the structure of a Basic Process, the second element stores synchronizations, and the third element stores information regarding internal compositions. The metadata of a Basic Process only has one set, with one tuple, since no composition has been made, and the second and third elements of the tuple are always the empty set. For $M(GetBox)$, $M(MoveBox)$, and $M(BrokenMoveBox)$, we can see that the metadata simply stores the sequence of events specified in the definition of those processes, with zero indicating a recursion.

With the metadata of the Basic Processes calculated, we can now check if the composition in *WorkingRobot* is deterministic. The process *WorkingRobot* is a composition of *GetBox* and *MoveBox* using the external choice operator of CSP. To check any composition, we apply a specific algorithm, depending on the operator in use, passing as arguments the metadata of its operands. In the case of *WorkingRobot*, the algorithm will receive $M(GetBox)$ and $M(MoveBox)$ as input.

After conducting its analysis, the algorithm will conclude that *WorkingRobot* is deterministic. This conclusion can be reached by verifying that the first event of each operand that is present in the choice offered to the environment, *selectBox* and *moveToDepot*, are different, and thus each path can be selected accurately; the complete definition of this algorithm is presented in Section 3.3.1.

The strategy then proceeds to create the related metadata, $M(WorkingRobot)$, for future use; $M(WorkingRobot)$ is shown below. The sequence has three sets, each with one tuple. The first two were extracted from $M(GetBox)$ and $M(MoveBox)$, and represent the behaviour of their respective processes. The last set represents the choice introduced by the composition, with the first element of its tuple containing only $SKIP$, which is the process that does nothing, used to avoid introducing new behaviour, the second element being the empty set, since there are no internal synchronizations, and the third element containing the positions 1 and 2, in $M(WorkingRobot)$, of the possible behaviours.

We create a tree structure in $M(WorkingRobot)$, as can be seen in Figure 1, by concatenating the metadata of *selectBox* and *moveToDepot*, and adding a new composite node. A detailed description of the structure of our metadata, and of the graphical notation used, is given in Section 3.2.

$$M(WorkingRobot) = \left\langle \begin{array}{l} \left\{ \left(\left\langle \left\{ \left\langle \begin{array}{l} selectBox, \\ pickUpBox, 0 \end{array} \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle \begin{array}{l} moveToDepot, \\ dropBox, \\ moveToDesk, 0 \end{array} \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle SKIP \right\rangle \right\} \right\rangle, \varnothing, \left\{ 1, 2 \right\} \right) \right\} \end{array} \right\rangle$$

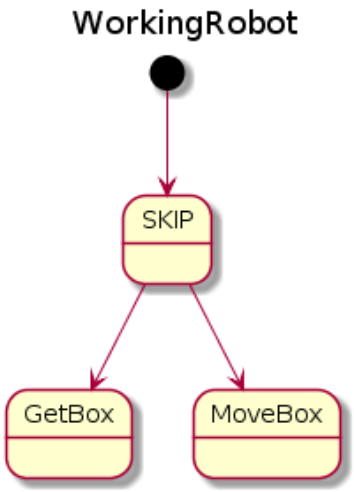

Figure 1 – Graphical representation of $M(WorkingRobot)$.

With $M(WorkingRobot)$, we can now check *BrokenRobot*. We use the same algorithm, since we have an external choice again, and its inputs are $M(WorkingRobot)$ and $M(BrokenMoveBox)$; we note that a more natural definition of *BrokenRobot* can be achieved with an internal choice, but we use an external choice here to illustrate aspects of our strategy. When checking this composition, however, the algorithm identifies the possibility of nondeterminism. This arises from the presence of *moveToDepot* in the initial events of both operands. In *BrokenRobot*, after performing *moveToDepot*, the environment can not control if the robot will perform *MoveBox* or *BrokenMoveBox*. At this point our strategy stops, returning that the external choice between *WorkingRobot* and *BrokenMoveBox* is a possible source of nondeterminism; no new metadata is created.

The automation provided by our prototype allows for quick and efficient application of the strategy. It calculates the metadata of the processes during its execution and applies the required algorithms at each composition. When applied to the given example, it outputs "Potential nondeterminism in $WorkingRobot \Box BrokenMoveBox$, the initial events of *WorkingRobot* and *BrokenMoveBox* can introduce nondeterminism".

1.4 Dissertation Outline

This section presents the structure of this dissertation. In Chapter 2 we discuss the necessary background of our work: the process algebra CSP, and its definitions of determinism. All the features of CSP used, as well as the various forms of defining determinism in CSP, and the definition chosen for our strategy, are presented.

In Chapter 3 we explain our strategy in detail. Initially we present the requirements needed to apply our strategy, which is followed by the definition the metadata used. Finally, we present the algorithms for each supported composition.

The case studies used, together with our experimental results, are discussed in Chapter 4. Beyond the problems used, and the results obtained, we also discuss the experimental infrastructure, and the features of our prototype. Finally, in Chapter 5, we summarise the contributions of our work. Additionally, we provide a comparative overview of related works dealing with local analysis, and propose ideas for future research.

2 BACKGROUND

We present here the background material related to our work: the process algebra CSP, in Section 2.1, and its notion of determinism, in Section 2.2.

2.1 CSP

CSP is a process algebra that can be used to describe systems as interacting components. These components, called processes, are independent entities that interact among themselves, and with the environment. The interactions, called events, are atomic, instantaneous, and synchronous messages. The main constructs of the CSP language are presented in Section 2.1.1, and its semantic models are discussed in Section 2.1.2; further information can be found in the works of Hoare (1985) and Roscoe (2010).

2.1.1 CSP Syntax

The processes in CSP are defined in terms of events. The set of all possible events is called Σ. We also define the set of events that a process P can engage in, its alphabet, denoted by α_P, with $\alpha_P \subseteq \Sigma$. To build a specification, processes can be composed in many different forms. Some of the main operators of CSP can be found in Figure 2.

Process ::= "$SKIP$"	Terminating process
| "$STOP$"	Deadlock process
| "DIV"	Divergent process
| Event " $\rightarrow$ " Process	Prefixing
| Condition "&" Process	Guard
| "if" Condition "then" Process "else" Process	Conditional
| Process " ; " Process	Sequential Composition
| Process " $\Box$ " Process	External Choice
| Process " $\sqcap$ " Process	Internal Choice
| Process " $\|\|\|$ " Process	Interleaving
| Process "$[\![$" X "$]\!]$" Process	Generalized Parallel
| Process " $\setminus$ " X	Hiding
| Process "$[\![$" R "$]\!]$"	Renaming

Figure 2 – Some of the operators of CSP.

CSP has three primitive processes, $SKIP$, $STOP$, and DIV, which can be used in the definition of other processes. The process $SKIP$ does nothing and terminates successfully, with the termination being marked by the special event $\checkmark \notin \Sigma$, which is always the last event communicated by any terminating process. The process $STOP$ stands for a canonical deadlock; it is not capable of any communication. Finally, the process DIV

models a divergent behaviour, standing for an infinite cycle of internal events, which are events that cannot be observed in the environment in which they are inserted.

A prefixing $a \rightarrow P$, with $a \in \Sigma$, is initially capable of performing the event a, and then behaves like the process P. Events can be compound to communicate data. For instance, $c.5$ is the event that represents the transmission of the value 5 through the channel c. Communications can be explicitly used for input or output. The composition $c?x \rightarrow P(x)$, for example, first receives a value through channel c, which is stored in the variable x, and then proceeds to behave as $P(x)$; in this case, we write $P(x)$ to indicate that the value of x can be used in P. An example of output can be seen in $in?x \rightarrow out!x \rightarrow SKIP$, which receives a value through channel in, outputs this value through channel out, and then terminates; the "!" symbol is semantically equivalent to the "." symbol.

The use of guards and conditionals is written $g \ \& \ P$, and if g then P else Q, respectively. The former behaves as P if g is true, and as $STOP$ otherwise. The latter behaves as P if g is true, and as Q otherwise. Sequential composition is written as $P \ ; Q$, and it behaves as P, until P terminates successfully, in which case it behaves as Q.

There are two operators that define choice in CSP. The process $P \ \Box \ Q$ is the external choice between P and Q, resolved in favour of either of them when the environment synchronizes on their initials, which are the sets of events that they initially offer. In contrast, we have the internal choice, $P \sqcap Q$, in which the environment has no control over how the choice is resolved, as the choice is taken by the process itself, potentially leading to nondeterminism.

To model parallelism in CSP we have various options. The process $P \ ||| \ Q$ is the interleaving of P and Q; in this composition P and Q execute in parallel, but independently. Another form of composition can be achieved with generalised parallelism, $P[\![X]\!]Q$, in which P and Q agree on $X \subseteq \Sigma$. The events in X occur for both operands, and the events outside of X occur independently; if $X = \varnothing$, the composition behaves as an interleaving.

Among other parallelism operators, we have, for example, alphabetized parallel, and linked parallel; these operators are not discussed here, since they are not present in our strategy. We note, however, that the extra forms of parallelism can be expressed in terms of operators we do consider (ROSCOE, 2010).

To achieve abstraction, we can encapsulate some events of a process. To do that we can write $P \setminus X$, which hides the events in the set X, with $X \subseteq \Sigma$, from the environment. It is important to point out that, although not visible, the hidden events still take place, leading to synchronizations not visible to the environment.

To rename the events of a process P, we write $P[\![R]\!]$, where R is a series of mappings of the form $a \leftarrow b$. The process $P[\![a \leftarrow b]\!]$ behaves as P, but all occurrences of a are replaced by b, so, for $P = a \rightarrow SKIP$, we have $P[\![a \leftarrow b]\!] = b \rightarrow SKIP$.

As an example, we present a specification of a railway network described by Schneider (1999). It is composed by a series of segments of tracks, with a signal between every

two adjacent segments used to control the flow of trains. The segments are organised in overlapping pairs, as shown in Figure 3(a), with segment pairs P_1, P_2 and P_3. In its initial state, the railway can have a number of trains in specific segments. A safety requirement is that no two trains should be in adjacent segments.

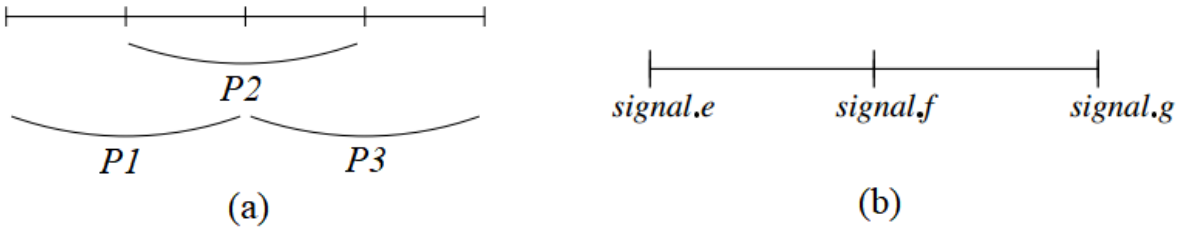

Figure 3 – Three overlapping pairs of segments (a), and signals of a pair of segments (b); modified from the original of Schneider (1999).

Each pair of segments has three signals, as indicated in Figure 3(b): e, which indicates a train entering the pair; f, which indicates the train moving from the first to the second segment; and g, which indicates the train leaving the pair. A pair of segments is modelled as a process that can communicate three events, $signal.e$, $signal.f$, and $signal.g$, corresponding to the signals e, f, and g. To deal with all possible initial states of a pair, three processes are defined. *Pair_Empty* specifies a pair that is initially empty, *Pair_First*, a pair in which a train is initially in its first segment, and *Pair_Second*, a pair in which a train is initially in its second segment.

$$Pair\_Empty = signal.e \rightarrow signal.f \rightarrow signal.g \rightarrow Pair\_Empty$$
$$Pair\_First = signal.f \rightarrow signal.g \rightarrow signal.e \rightarrow Pair\_First$$
$$Pair\_Second = signal.g \rightarrow signal.e \rightarrow signal.f \rightarrow Pair\_Second$$

To model a network we compose a number of instances of pairs of segments in parallel, with each instance having its own signals defined according to its position in the network. In Figure 4 we present an example of a cyclic network that has four pairs and four segments, with the last and the first segments being adjacent to each other. Figure 4(a) gives an overview of the complete network, in which there is initially a single train in the segment demarcated by signal.0 and signal.1. Figure 4(b) shows the four segment pairs, from $Pair0$ to $Pair3$. In this figure, the segment pairs are expressed in continous lines. For example, $Pair0$ in Figure 4(b) represents the pair of segments demarcated by $signal.0$ and $signal.1$, and $signal.1$ and $signal.2$.

The CSP processes that describe the four segment pairs are presented in Figure 5. These processes define the initial state of each segment pair. Therefore, although $Pair0$ is formed of the two segments demarcated by $signal.0$ and $signal.1$, and $signal.1$ and $signal.2$, the process is written as $Pair0 = signal.1 \rightarrow signal.2 \rightarrow signal.0 \rightarrow Pair0$, because the

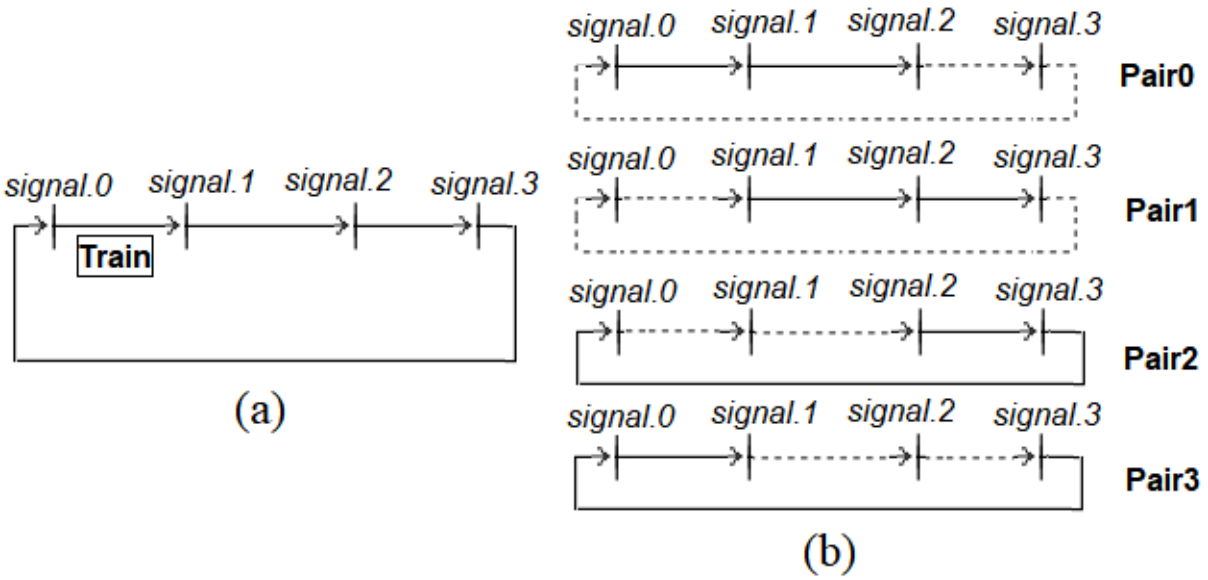

Figure 4 – Graphical representation of a network (a), and its pairs (b).

train is in the first segment of this pair, and the next relevant event it must communicate is $signal.1$, indicating the train is moving from the first to the second segment of $Pair0$. So $Pair0$ follows the form of $Pair\_First$, previously explained. Similarly, $Pair1$ and $Pair2$ are modelled as $Pair\_Empty$, since the train is not in any of their segments. Finally, $Pair3$ is modelled as $Pair\_Second$, as the train is in the second segment of this pair. The composition of the pairs is made using the generalised parallel operator, since the signals of a pair need to synchronise with the signals of its adjacent pairs.

$$
\begin{aligned}
&Pair0 = signal.1 \rightarrow signal.2 \rightarrow signal.0 \rightarrow Pair0\\
&Pair1 = signal.1 \rightarrow signal.2 \rightarrow signal.3 \rightarrow Pair1\\
&Pair2 = signal.2 \rightarrow signal.3 \rightarrow signal.0 \rightarrow Pair2\\
&Pair3 = signal.1 \rightarrow signal.3 \rightarrow signal.0 \rightarrow Pair3\\
&SyncSet1 = \{signal.1, signal.2\}\\
&SyncSet2 = \{signal.0, signal.2, signal.3\}\\
&SyncSet3 = \{signal.0, signal.1, signal.3\}\\
&RailwayNetwork = ((Pair0 [\![SyncSet1]\!] Pair1) [\![SyncSet2]\!] Pair2) [\![SyncSet3]\!] Pair3
\end{aligned}
$$

Figure 5 – CSP model of the network in Figure 4

The process $RailwayNetwork$ can initially communicate only $signal.1$, and afterwards $signal.2$, $signal.3$, $signal.0$, $signal.1$, $signal.2$ and so on. Each pair synchronises its first two signals with the pair on its left and its last two signals with the pair on its right, so when the network communicates $signal.1$, it means that a train is, simultaneously: moving from segment 1 to segment 2 of $Pair0$, entering segment 1 of Pair 1, and leaving $Pair3$.

2.1.2 CSP Semantics

CSP has both an operational and a denotational semantics (ROSCOE, 2010). The operational semantics of a process is given in terms of a Labelled Transition System, LTS for short, which models a specification as a set of states, with transitions between them. This approach is used, for example, in FDR, which creates the LTS of a CSP specification to conduct its analysis.

To reason about specifications, denotational semantics are usually preferred. There are three well established denotational semantic models for CSP: traces, failures, and failures-divergences. Other models exist, such as the stable revivals model used for deadlock analysis by Antonino, Sampaio and Woodcock (2014).

The traces model is the simplest one available. In it, a process P is represented by $traces(P)$, which is the set that contains all sequences of events in which P can engage. The trace semantics of the operators used in our strategy can be seen in Definition 1.

Definition 1 (Trace Semantics) *Let P and Q be two CSP processes, and $a \in \Sigma$.*

$$
\begin{aligned}
&traces(SKIP) = && \{\langle\rangle, \langle\checkmark\rangle\} \\
&traces(STOP) = && \{\langle\rangle\} \\
&traces(a \rightarrow P) = && \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces(P)\} \\
&traces(g\ \&\ P) = && traces(P),\ if\ g\ is\ true,\ traces(STOP)\ otherwise \\
&traces(if\ g\ then\ P\ else\ Q) = && traces(P),\ if\ g\ is\ true,\ traces(Q)\ otherwise \\
&traces(P\ ;\ Q) = && (traces(P) \cap \Sigma^*) \cup \\
& && \{s^\frown t \mid s^\frown\langle\checkmark\rangle \in traces(P) \wedge t \in traces(Q)\} \\
&traces(P \,\Box\, Q) = && traces(P) \cup traces(Q) \\
&traces(P \sqcap Q) = && traces(P) \cup traces(Q) \\
&traces(P \,|||\, Q) = && \bigcup\{s \,|||\, t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
&traces(P[\![X]\!]Q) = && \bigcup\{s \underset{X}{\parallel} t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
&traces(P \setminus X) = && \{t \setminus X \mid t \in traces(P)\}
\end{aligned}
$$

where $s^\frown t$ stands for the concatenation of the sequences s and t, Σ^* represents all possible finite sequences of events in Σ, and $t \setminus X$ represent a subsequence of t, without the events of X; the functions $|||$, and $\underset{X}{\parallel}$, can be found in definitions 2, and 3, respectively.

Definition 2 (Interleaving of Sequences) *Let s and t be sequences of events, with $a, b \in \Sigma$. Then the interleaving $s \,|||\, t$ is defined as:*

$$
\begin{aligned}
\langle\rangle \,|||\, s &= \{s\} \\
s \,|||\, \langle\rangle &= \{s\} \\
\langle a\rangle^\frown s \,|||\, \langle b\rangle^\frown t &= \{\langle a\rangle^\frown u \mid u \in s \,|||\, \langle b\rangle^\frown t\} \cup \{\langle b\rangle^\frown u \mid u \in \langle a\rangle^\frown s \,|||\, t\}
\end{aligned}
$$

Definition 3 (Parallelism of Sequences) *Let s and t be sequences of events, $X \subseteq \Sigma$, $x, x' \in X$, and $y, y' \notin X$. Then the parallelism $s \underset{X}{\parallel} t$ is given by:*

$$
\begin{aligned}
s \underset{X}{\parallel} t &= t \underset{X}{\parallel} s \\
\langle\rangle \underset{X}{\parallel} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{\parallel} \langle x\rangle &= \varnothing \\
\langle\rangle \underset{X}{\parallel} \langle y\rangle &= \{\langle y\rangle\} \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle y\rangle^\frown t &= \{\langle y\rangle^\frown u \mid u \in \langle x\rangle^\frown s \underset{X}{\parallel} t\} \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x\rangle^\frown t &= \{\langle x\rangle^\frown u \mid u \in s \underset{X}{\parallel} t\} \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x'\rangle^\frown t &= \varnothing \text{ if } x \neq x' \\
\langle y\rangle^\frown s \underset{X}{\parallel} \langle y'\rangle^\frown t &= \{\langle y\rangle^\frown u \mid u \in s \underset{X}{\parallel} \langle y'\rangle^\frown t\} \cup \{\langle y'\rangle^\frown u \mid u \in \langle y\rangle^\frown s \underset{X}{\parallel} t\}
\end{aligned}
$$

The process $STOP$ has only the empty trace, since it models a deadlock. The process $SKIP$, on the other hand, can perform the trace with ✓. The semantics of prefixing, guard, conditional, external choice, and internal choice, are quite straightforward. With sequential composition, we must point out that ✓ marks the concatenation point of the sequences of P and Q, but ✓ itself does not belong to any trace in traces(P ; Q); $traces(P) \cap \Sigma^*$ represents the traces of P that do not contain ✓.

The traces of interleaving and generalized parallelism, although not trivial, are not complex, as they simply represent the intertwine of sequences. The traces of hiding are also simple, since they entail removing the hidden events from all the sequences that are traces of the process.

The traces model allows us to capture what a process can do, but not what it is unable to do. One example of this is that with traces we cannot differentiate between external and internal choice, even though they can lead to different behaviours: an internal choice may refuse events that are accepted by an external choice. To capture what a process cannot do, we use the failures model.

In the failures model, a process P is represented by the pair ($traces(P)$, $failures(P)$), with $failures(P)$ being a set of pairs (s, X), where s is a trace of P and X is a set of events that P can refuse after performing s. This model captures not only how a process can behave, but also how it may refuse to behave. The failures semantics of the CSP operators used in our strategy can be seen in Definition 4.

The processes $SKIP$ and $STOP$ refuse all events of Σ, with $SKIP$ accepting only ✓, after the empty trace. The semantics of prefixing, guard, conditional, sequential composition, and internal choice are, like in the traces model, quite straightforward. External

Definition 4 (Failures Semantics) *Let P and Q be two CSP processes, and $a \in \Sigma$.*

$$
\begin{array}{ll}
\mathit{failures}(SKIP) = & \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma \cup \{\checkmark\}\} \\
\mathit{failures}(STOP) = & \{(\langle\rangle, X) \mid X \subseteq \Sigma \cup \{\checkmark\}\} \\
\mathit{failures}(a \rightarrow P) = & \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle^\frown s, X) \mid (s, X) \in \mathit{failures}(P)\} \\
\mathit{failures}(g\ \&\ P) = & \mathit{failures(P), if\ g\ is\ true, failures(STOP)\ otherwise} \\
\mathit{failures}(\mathit{if}\ g\ \mathit{then}\ P\ \mathit{else}\ Q) = & \mathit{failures(P), if\ g\ is\ true, failures(Q)\ otherwise} \\
\mathit{failures}(P\ ;\ Q) = & \{(s, X) \mid s \in \Sigma^* \wedge (s, X \cup \langle\checkmark\rangle) \in \mathit{failures}(P)\} \cup \\
& \{(s^\frown t, X) \mid s^\frown\langle\checkmark\rangle \in \mathit{traces}(P) \cup (t, X) \in \mathit{failures}(Q)\} \\
\mathit{failures}(P \,\Box\, Q) = & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}(P) \cap \mathit{failures}(Q)\} \cup \\
& \{(t, X) \mid (t, X) \in \mathit{failures}(P) \cup \mathit{failures}(Q) \wedge t \neq \langle\rangle\} \cup \\
& \{(\langle\rangle, X) \mid X \subseteq \Sigma \wedge \langle\checkmark\rangle \in \mathit{trace}(P) \cup \mathit{traces}(Q)\} \\
\mathit{failures}(P \sqcap Q) = & \mathit{failures}(P) \cup \mathit{failures}(Q) \\
\mathit{failures}(P \,|||\, Q) = & \bigcup\{(s \,|||\, t, Y \cup Z) \mid Y \setminus \checkmark = Z \setminus \checkmark \wedge \\
& (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q)\} \\
\mathit{failures}(P[\![X]\!]Q) = & \bigcup\{(s \underset{X}{\parallel} t, Y \cup Z) \mid Y \setminus (X \cup \{\checkmark\}) = Z \setminus (X \cup \{\checkmark\}) \wedge \\
& (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q)\} \\
\mathit{failures}(P \setminus X) = & \{(t \setminus X, Y) \mid (t, X \cup Y) \in \mathit{failures}(P)\}
\end{array}
$$

choice, however, is more complex, since it now takes into account the events that cannot be engaged in, after the choice is resolved. The semantics of interleaving, generalized parallelism, and hiding are uplifted from the traces model, adding considerations for the refusal of events.

To enhance the failures model, we have the failures-divergences model, the more complete of the classical ones. This model is capable of capturing divergent behaviour in a specification. It is built upon the failures model, similarly as the failures model is built upon the traces model. As this model is not used in our strategy, it is not presented here. In the next section, we discuss determinism using the failures model.

2.2 Determinism

A deterministic system can be thought of as one that always produces the same output, given a fixed input. Nondeterminism happens when we cannot control the output of a system with its input. An example of nondeterminism can be seen in Figure 6, where the system initially can perform e, but it has no control if it will be able to synchronize on either a or b afterwards.

CSP has different definitions for determinism, depending on the semantic model being used. The traces model, since it does not capture what a process cannot do, is not rich enough to allow us to determine if the process is deterministic or not, so it cannot be used for this end. For the example of Figure 6, the traces model can only capture that the traces $\langle e, a\rangle$, and $\langle e, b\rangle$ are valid, but it cannot capture that after e, either a, or b, can be refused, depending on how the choice is resolved.

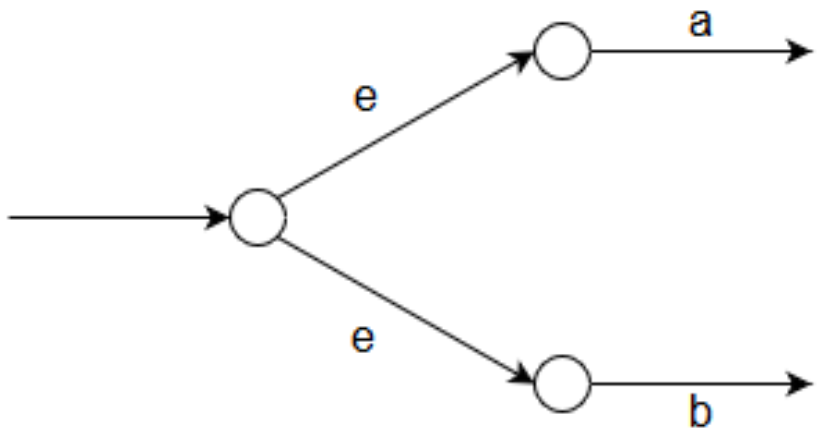

Figure 6 – Graphical example of nondeterminism.

The failures model is rich enough to capture nondeterminism. Its definition of determinism is reproduced in Definition 5. It states that, after a trace tr, a process cannot have the possibility of both accepting and refusing any given event a.

Definition 5 (Determinism in the failures model) *The process P is deterministic if, and only if,* $\forall\, tr : traces(P), a : \Sigma \bullet \neg(tr^\frown\langle a\rangle \in traces(P) \wedge (tr, \{a\}) \in failures(P))$.

This definition captures the essence of determinism: a process cannot have the possibility of both accepting and refusing an event at any given state, which can lead to different observable behaviours given the same input. To further illustrate the definition, consider the concrete examples 1, 2, and 3, based on the specification shown in Figure 5.

Example 1 The process $Ex1a = Pair1 \,\Box\, Pair2$ is deterministic, since $Pair1$ and $Pair2$ are deterministic and the intersection of their initials is empty. Without initial events in common, the environment has a clear choice between $Pair1$ and $Pair2$, which, with their traces and failures, do not violate Definition 5.

The process $Ex1b = Pair1 \,\Box\, Pair3$, on the other hand, is nondeterministic, because $signal.1$ is in the initials of both $Pair1$ and $Pair3$, so, by performing $signal.1$, the environment has no control over how the external choice is resolved, allowing $Ex1b$ to both accept or refuse $signal.2$ afterwards, depending on whether $Pair1$ or $Pair3$ is chosen; the trace $\langle signal.1, signal.2\rangle$ and the failure $(\langle signal.1\rangle,\{signal.2\})$, for instance, are in the semantics of $Ex1b$, which therefore does not satisfy the restriction in Definition 5. □

Example 2 The composition in $Ex2 = Pair1 \,|||\, Pair2$ is nondeterminitic because we have an event, $signal.3$, after which $Pair1$ and $Pair2$ behave differently. In terms of Definition 5, we note that $\langle signal.1, signal.2, signal.3, signal.1\rangle$ is a trace of $Ex2$, and $(\langle signal.1, signal.2, signal.3\rangle,\{signal.1\})$ is a failure of $Ex2$. □

Example 3 The hiding of event $signal.1$ in $Ex1a$, $Ex1a \setminus \{signal.1\}$, leads to nondeterministic behaviour, since the environment looses control over the choice between $Pair1$ and $Pair2$. One possible trace of the composition is $\langle signal.2, signal.3, signal.2\rangle$, if the choice is resolved in favour of $Pair1$, and one failure is $(\langle signal.2, signal.3\rangle,\{signal.2\})$, if the choice is resolved in favour of $Pair2$. □

With the failures model, we can capture the main aspect of nondeterminism, discussed so far. A different source of nondeterminism, however, is divergence, which is captured by the failures-divergences model. Its depiction of nondetermism is built upon Definition 5, with added considerations for divergent behaviour.

We adopt the definition of determinism in the failures model for the development of our strategy. There are two reasons for this decision. The first is that, as previously stated, the failures model captures the essence of determinism, which is the property we focus on, and not divergence. Second, there are already tools that verify divergence in a compositional way (CONSERVA FILHO et al., 2018), and can complement our strategy. The two-step approach for verification of determinism in the failures-divergences model, of first ensuring that the specification is divergence-free, and then checking for determinism in the failures model, can already be taken for global analysis, with automation being provided by tools like FDR (ROSCOE, 2010). Our strategy can be part of local analysis alternative for this two-step verification.

The global analysis of determinism in the failures model implemented by FDR is based on Lazic's algorithm for determinism checking (ROSCOE, 2010). This algorithm requires two copies of the given process in parallel in order conduct the analysis, which requires the creation of the whole state space of this parallel composition, which is even larger than that of the original process. In the next chapter we describe our approach, in which determinism can be predicted by analysing only a restricted subset of the state space of the process being checked.

3 STRATEGY FOR LOCAL ANALYSIS OF DETERMINISM

Our analysis of a process is compositional. We constructively check the process, starting from its basic components. If the possibility of nondeterminism is found at any point, the analysis stops and indicates the component that may be the source of nondeterministic behaviour. We collect information of the processes analysed, in the form of metadata, so that it is readily available when necessary, avoiding unnecessary recalculation.

Our strategy is sound, but not complete. When nondeterminism is indicated, we may have found a true nondeterminism, or it may be an inconclusive result, but when the strategy indicates that a process is deterministic, it is certainly the case. Local approaches to the analysis of classical concurrency properties tend to give up completeness in favour of efficiency gains; see, for example, the works of Antonino, Sampaio and Woodcock (2014), Antonino et al. (2014), Antonino, Gibson-Robinson and Roscoe (2016a), and Antonino, Gibson-Robinson and Roscoe (2016b) for deadlock analysis, and Conserva Filho et al. (2016), and Conserva Filho et al. (2018) for livelock analysis. To illustrate the incompleteness of our approach, we present a concrete example.

Example 4 We consider the following processes.

$$Ex4a = a \rightarrow b \rightarrow Ex4a \qquad Ex4c = Ex4a \sqcap Ex4b$$
$$Ex4b = c \rightarrow d \rightarrow SKIP \qquad Ex4d = Ex4c[\![\{a, c\}]\!]STOP$$

The process $Ex4c$ is nondeterministic, due to its internal choice. The process $Ex4d$, on the other hand, is deterministic, since it is equivalent to STOP. When analysing $Ex4d$, however, our strategy indicates the possibility of nondeterminism in $Ex4c$ and stops. □

Our claim of soundness is not yet backed by a formal proof. We intend for the strategy to have this property, and the experiments provide evidence in this direction, but currently there is no guarantee. In this chapter we assume that our approach is sound, with the lack of soundness being addressed in Chapter 5.

We present the subset of CSP that our strategy works with in Section 3.1, and the metadata it gathers after each composition in Section 3.2. The algorithms used to check for determinism are detailed in Section 3.3.

3.1 Process Structure and Restrictions

In our strategy the processes are separated in two categories, Basic Processes and Composite Processes, defined in Figure 7. The elements Event, Condition, ProcessName, and

SetOfEvents are the syntactic categories of the possible events, logical conditions, names of processes, and sets of events of CSP, respectively. We assume that all processes are divergence free, since we do not intend to capture nondeterminism introduced by divergence; this assurance can be achieved by conducting an appropriate verification prior to the application of our strategy, and checking for divergence can also be compositional (CONSERVA FILHO et al., 2018).

Process ::= BasicProcess | CompositeProcess

BasicProcess ::= "$SKIP$" | "$STOP$" | ProcessName
| Event " → " BasicProcess
| Condition "&" BasicProcess
| "if" Condition "then" BasicProcess "else" BasicProcess
| BasicProcess " ; " BasicProcess
| BasicProcess " □ " BasicProcess

CompositeProcess ::= ProcessName " ; " ProcessName
| ProcessName " □ " ProcessName | ProcessName " ⊓ " ProcessName
| ProcessName " ||| " ProcessName | ProcessName "[[" SetOfEvents "]]" ProcessName
| ProcessName " \ " SetOfEvents

Figure 7 – BNF of the subset of CSP considered.

Due to the nature of the set of operators that can be used to create Basic Processes, most of them are deterministic by definition. The processes $SKIP$ and $STOP$ are deterministic, and prefixing, guard, conditional, and sequential composition cannot introduce nondeterminism; if a Basic Process is composed exclusively of these elements, no verification is needed. The external choice operator, however, can lead to nondeterminism. If this operator is used in a Basic Process, it is necessary to check if the process is deterministic before applying the strategy, using an external verification. We include external choice as a composition operator for both Basic and Composite processes, however, to be able to cover a larger class of processes with our strategy. For example, a memory cell, as shown below, cannot be modelled without external choice for Basic Processes.

$$\begin{aligned} MemoryCell(value) = \; & in?newValue \rightarrow MemoryCell(newValue) \\ & \Box \\ & out!value \rightarrow MemoryCell(value) \end{aligned}$$

Note that Basic Processes consist of a structured, possibly branching, sequence of events, ending in either $SKIP$, $STOP$, or recursion; we allow only tail recursion, with a Basic Process not being able to reference any other process. Composite Processes, on the contrary, consist of a single composition, with its operands being references to other processes. This distinction is used to enforce the idea of components being put together.

In our strategy, no verification is carried out of the Basic Processes. The Composite Processes, which are the result of compositions between Basic Processes or other Composite Processes, are our focus.

The subset of CSP that our strategy can analyse includes most of the main operators of the language. They are, however, restricted on how they can be used. Prefixing, guards, and conditionals, can only be used in Basic Processes, while internal choice, interleaving, generalized parallel, and hiding are restricted to Composite Processes. To allow for greater expressiveness, sequential composition and external choice can be used both in Basic Processes and in Composite Processes, with only the added burden of checking, before applying the strategy, if the external choice can lead to nondeterminism in Basic Processes.

3.2 Metadata

For each process in a given specification, upon it being guaranteed to be deterministic, be it by definition or verification, we gather metadata, M, about it. This metadata is a sequence of sets of tuples, with the tuples being of size three. Each tuple in M represents a Basic Process, and the structure of a sequence of sets reflects the compositions. The first element of each tuple stores the syntactic structure of a Basic Process, the second element stores synchronisations among processes, which arise from parallel compositions, and the third element stores the order in which the compositions are made.

Example 5 The metadata of the processes $Ex4a$ and $Ex4b$, from Example 4, and of $Ex5 = Ex4a \,\Box\, Ex4b$, is shown below.

$$M(Ex4a) = \left\langle \left\{ \left(\left\langle \left\{ \left\langle \, a, b, 0 \, \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(Ex4b) = \left\langle \left\{ \left(\left\langle \left\{ \left\langle \, c, d, SKIP \, \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(Ex5) = \left\langle \begin{array}{l} \left\{ \left(\left\langle \left\{ \left\langle \, a, b, 0 \, \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle \, c, d, SKIP \, \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \langle SKIP \rangle \right\} \right\rangle, \varnothing, \left\{ \, 1, 2 \, \right\} \right) \right\} \end{array} \right\rangle$$

The metadata of all Basic Processes consist of a sequence with a singleton set; it has one tuple, since no composition among processes is present. In that tuple, the second and third elements are always the empty set, because there can be no synchronisations or compositions in a Basic Process. We can see that $M(Ex4a)$ and $M(Ex4b)$ store the sequence of events of their respective processes, $a \rightarrow b \rightarrow Ex4a$, and $c \rightarrow d \rightarrow SKIP$, in the first element of their tuples; the zero represents a recursion. The second element is

the empty set, since there are no synchronizations at this point, as is the third, because no compositions were made. The general form of the first element of each tuple is always a sequence of sets of sequences; this is further discussed in Example 7.

The sequence of $M(Ex5)$ contains three sets, since we are dealing with a Composite Process. The first two are the sets of $M(Ex4a)$ and $M(Ex4b)$, and the third has a composition tuple, which contains only $SKIP$ in its first element, and models the choice introduced by the composition operator; we use the term "composition tuple" to reference a tuple that does not store structure of a Basic Process. The third element of the composition tuple has the positions, in $M(Ex5)$, of the two alternative behaviours: 1 and 2, in this case. The sequence of sets of the metadata creates a tree structure, storing the order of the compositions, with the root being the last set. A graphical representation of $M(Ex5)$ can be seen in Figure 8; the graphical notation used is presented in detail after the next example. □

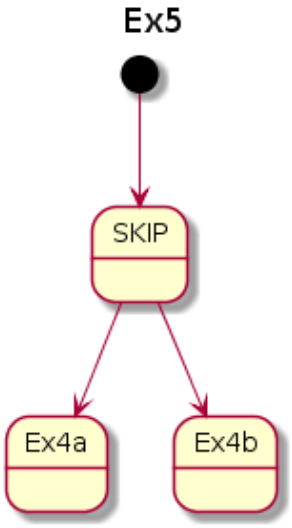

Figure 8 – Graphical representation of $M(Ex5)$.

The second element of each tuple, the synchronisation set, is a set of pairs, with each pair having an integer value as its first component, and a set of events as its second component. The events in the pair are the ones being synchronised, and the integer values identify which processes participate in the synchronisation.

Example 6 We consider the following processes.

$$Ex6a = a \rightarrow STOP \qquad Ex6c = b \rightarrow Ex6c$$
$$Ex6b = Ex6a [\![\{a\}]\!] Ex4a \qquad Ex6d = Ex6b [\![\{b\}]\!] Ex6c$$

With the generalised parallel operator, we add the synchronised events to all tuples of the metadata, except the new composition tuple, the last one in the sequence, as shown in

$M(Ex6b)$. The base, unsigned, integer value uniquely identifies the synchronisation, and its signal is used to differentiate between the two process arguments of the parallelism; a fresh integer is used for each new synchronisation, with its positive value indicating the left operand, and the negative counterpart indicating the right operand. Note that, differently from $M(Ex5)$, the tuples of the operands of $Ex6b$ are joined in a single set; in the metadata, tuples in the same set model parallelism.

$$M(Ex6b) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle a, STOP \right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ a \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle a, b, 0 \right\rangle, \right\} \right\rangle, \left\{ \left(-1, \left\{ a \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \left\langle \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 1 \right\} \right) \right\} \right\rangle$$

If $Ex6b$ is used in a composition with generalised parallel, as in $Ex6d$, we add the new synchronised events to its tuples. When more than one non-composition tuple has a synchronisation set with the same integer, 2 in the case of $M(Ex6d)$, only one of those need to synchronise with a counterpart indexed with the opposite integer, -2 in this case. The graphical representation of $M(Ex6b)$ and $M(Ex6d)$ can be seen in figures 9 and 10; the dashed vertical lines in the figures are used to represent parallel regions.

$$M(Ex6d) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle a, STOP \right\rangle, \right\} \right\rangle, \left\{ \begin{pmatrix} 1, \left\{ a \right\} \end{pmatrix}, \\ \begin{pmatrix} 2, \left\{ b \right\} \end{pmatrix} \end{Bmatrix}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle a, b, 0 \right\rangle, \right\} \right\rangle, \left\{ \begin{pmatrix} -1, \left\{ a \right\} \end{pmatrix}, \\ \begin{pmatrix} 2, \left\{ b \right\} \end{pmatrix} \end{Bmatrix}, \varnothing \right) \end{Bmatrix}, \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \left\{ \left(2, \left\{ b \right\} \right) \right\}, \left\{ 1 \right\} \right), \\ \left(\left\langle \left\{ \left\langle b, 0 \right\rangle, \right\} \right\rangle, \left\{ \left(-2, \left\{ b \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \left\{ \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 2 \right\} \right) \right\} \right\rangle$$

□

The graphical notation we use to represent the metadata is UML state diagrams (FAKHROUTDINOV, 2015). With this notation, we present the metadata as the tree structure that its sequence models, with Basic Processes being the nodes and their relations representing a wrapper for composed processes; besides the Basic Processes, the composite nodes, whose body contains only $SKIP$, are also shown.

To represent parallelism, we use orthogonal states, with one region for each operand; an example of an orthogonal state can be seen in Figure 9. Since our goal is to present the structure of the metadata itself, the internal structure of the nodes, that is, their sequences of events, is omitted.

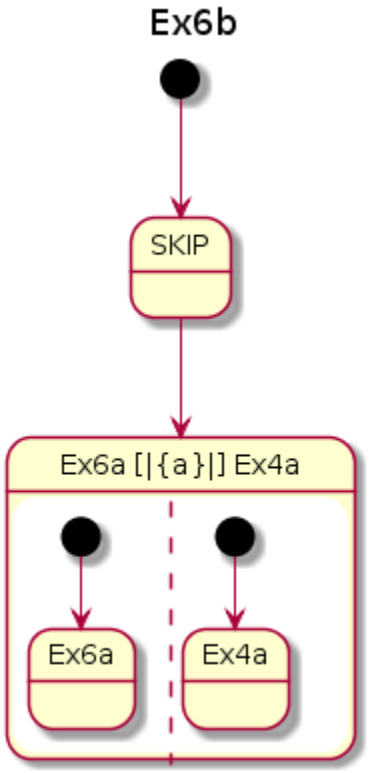

Figure 9 – Graphical representation of $M(Ex6b)$.

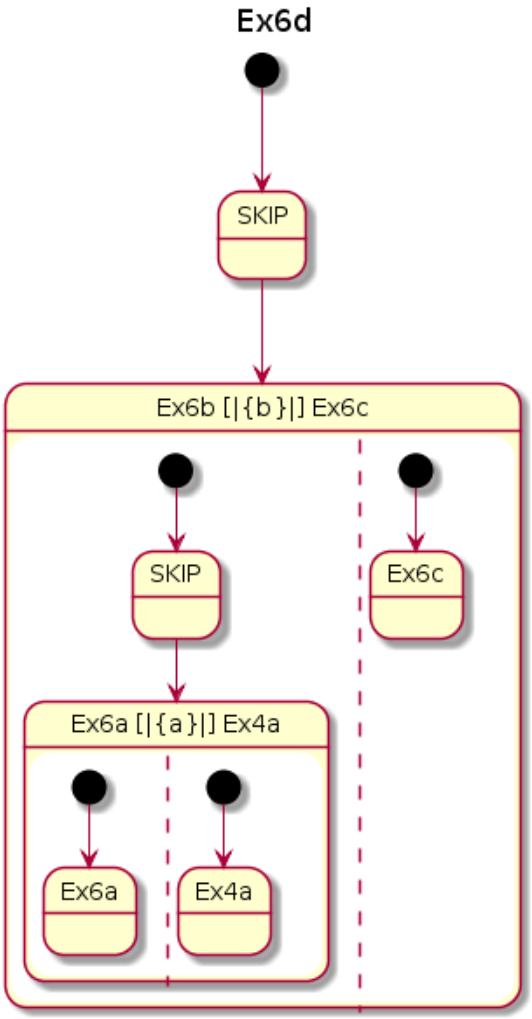

Figure 10 – Graphical representation of $M(Ex6d)$.

We now present the formal definition of our metadata. First, it is important to record how a process behaves after a sequence of events. To this end, we add an indicator that represents the final behaviour of the sequence as its last element. The indicator can be $SKIP$ or $STOP$, to represent their respective behaviours, or a natural number, that works as a pointer to another sequence of events, with zero representing a recursion. We call the set that contains these new sequences Valid Sequences, $VSeq$.

Definition 6 (Valid Sequences (VSeq))

$$VSeq = \{a : \Sigma^*, b : \mathbb{N} \cup \{SKIP, STOP\} \bullet a^\frown\langle b\rangle\}$$

To hold the structure of a Basic Process, we combine elements of $VSeq$ in a sequence of sets. This sequence represents the internal structure of the process, with multiple elements in a set representing an external choice, and pointers in a valid sequence being used to model sequential composition and recursion. We call the set that contains theses sequences of sets Valid Structures, $VStruct$.

Definition 7 (Valid Structures (VStruct))

$$VStruct = Seq(\mathbb{P}(VS))$$

where Seq stands for the sequence type constructor.

Example 7 We consider the following processes.

$$Ex7a = a \rightarrow Ex7a$$
$$Ex7b = a \rightarrow Ex7b \,\Box\, b \rightarrow SKIP$$
$$Ex7c = (a \rightarrow Ex7c \,\Box\, b \rightarrow SKIP)\,;\, c \rightarrow SKIP$$
$$Ex7d = (a \rightarrow Ex7d \,\Box\, b \rightarrow SKIP)\,;\, (c \rightarrow SKIP \,\Box\, d \rightarrow STOP)\,;\, e \rightarrow SKIP$$

The processes $Ex7a$, $Ex7b$, $Ex7c$, and $Ex7d$ are built in an incremental way, so that we can see the evolution of their valid structures. For $Ex7a$, our starting point, we have a structure similar to that of the previous examples, a sequence with one set, which contains an element of $VSeq$, $\langle\{\langle a, 0\rangle\}\rangle$; zero symbolises a recursion.

For $Ex7b$ we have $\langle\{\langle a, 0\rangle, \langle b, SKIP\rangle\}\rangle$, since we add a new sequence to the set, with multiple sequences in a set representing an external choice between behaviours. For $Ex7c$ we have $\langle\{\langle c, SKIP\rangle\}, \{\langle a, 0\rangle, \langle b, 1\rangle\}\rangle$, by adding a new set to represent the sequential behaviour, with the $SKIP$ that triggers the composition becoming a pointer to the new behaviour; the initial behaviour of the Basic Process is always given by the last set. For $Ex7d$ we have $\langle\{\langle e, SKIP\rangle\}, \{\langle c, 1\rangle, \langle d, STOP\rangle\}, \{\langle a, 0\rangle, \langle b, 2\rangle\}\rangle$, since we add the external choice with $d \rightarrow STOP$ in the second set, create a new set for the new sequential composition, and update the pointers.

Initially, $Ex7d$ has a choice between the events a and b. Event a leads to a recursion, index 0, while event b leads to a choice between the events c and d, with the pointer 2 to the second set of the sequence. If event c happens then event e followed by $SKIP$ will happen, by following the pointer 1 to the first set of the sequence, and if event d happens, we have a deadlock. □

To record which events in a tuple must take part in an internal synchronisation, we use, in their second element, pairs of integers and sets of events. With these pairs we can know which events in the valid structure of the tuple are part of a synchronisation, when applying the verification algorithms. We call the set that contains all possible sets of these pairs Syncronisation Sets, *SyncSets*; the use of *SyncSets* can be seen in Example 6.

Definition 8 (Syncronisation Sets (SyncSets))

$$SyncSets = \mathbb{P}(\mathbb{Z}^1 \times \mathbb{P}(\Sigma))$$

where $\mathbb{Z}^1$ stands for non-zero integers.

Finally, we define the Enhanced Traces, *eTraces*. It is a set whose elements are tuples of size three, with their first element being a *VStruct* sequence, the second one being a *SyncSets* set, and the third being a set of positive naturals.

Definition 9 (Enhanced Traces of P (eTraces(P)))

$$eTraces(P) = \left\{ \begin{array}{l} et : VStruct \times SyncSets \times \mathbb{N}^+ \mid \forall\, s \in fullSeq(first(et)) \bullet \\ front(s) \in traces(P) \land P/front(s) \equiv_{\mathrm{F}} last(s) \end{array} \right\}$$

where P/t represents the behaviour of P after it has performed the trace t, and $\equiv_{\mathrm{F}}$ indicates equivalence in the failures model. The function fullSeq receives an element of VStruct, and returns a set with all the sequences that start in the last set of its argument and ends in $SKIP$, $STOP$, or recursion, with the pointers being removed after used; $fullSeq(\langle\{\langle c, SKIP\rangle\}, \{\langle a, 0\rangle, \langle b, 1\rangle\}\rangle) = \{\langle a, 0\rangle, \langle b, c, SKIP\rangle\}$, with the value 0 standing for a recursive call; the order of the sets is inverted for efficiency reasons, and $last(0)$ stands for $last(P)$, since 0 represents a recursion.

The set *eTraces* is defined for a given Basic Process P, and its elements are all possible tuples that can represent P. The predicate in Definition 9 ensures that each sequence of events that can be derived, through all valid pointers, from the valid structure of the tuple contains a valid trace of P and leads the process to $SKIP$, $STOP$, or a recursion. With these restrictions, we can use *eTraces* as our building blocks to define metadata.

With *eTraces*, we can now give the complete definition of our metadata. For a process P, we define $Metadata(P)$ as a set containing sequences of subsets of $eTraces(P)$. The tuples in a same set represent Basic Processes in parallel, and the multiple sets, guided by the pointers, which are the third element of the tuples, model external choice and sequential composition; one pointer represents sequential composition, and many pointers represent an external choice.

Definition 10 (Metadata of P (Metadata(P)))

$$Metadata(P) = Seq(\mathbb{P}(eTraces(P)))$$

where Seq stands for the sequence type constructor.

To map a process P to its correct metadata, we use the semantic function M, declared below. The type of M is a dependent type (BOVE; DYBJER, 2009). It maps a process P to a metadata for this process. The range of M is specialised for each process P to which M is applied; this is why we need a dependent type declaration. So, although the type of the source of M is Process, we need to declare a variable $(P : Process)$ so that P can be used to define the type of the range of M for P, which is $Metadata(P)$.

Definition 11 (Semantic Function of P (M(P)))

$$M : (P : Process) \rightarrow Metadata(P)$$

Now we present how the metadata is calculated. For the Basic Processes, we calculate M as shown below; P and Q are processes, n is a process name, representing a recursive call, and g is a boolean value. Note that the following definition is by structural induction on the syntax of Basic Process.

Definition 12 (Metadata of the Basic Processes)

- $M(n) = \left\langle \left\{ \left(\left\langle \left\{ \langle 0 \rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$
- $M(SKIP) = \left\langle \left\{ \left(\left\langle \left\{ \langle SKIP \rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$
- $M(STOP) = \left\langle \left\{ \left(\left\langle \left\{ \langle STOP \rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$
- $M(a \rightarrow P) = \left\langle \left\{ prefixingBasic(a, getBasic(M(P))) \right\} \right\rangle$
- $M(g \ \& P) = M(P)$
- $M(if\, g\ then\ P\ else\ Q) = M(P)$
- $M(P\ ;\ Q) = \left\langle \left\{ seqComp(getBasic(M(P)), getBasic(M(Q))) \right\} \right\rangle$
- $M(P \ \Box \ Q) = \left\langle \left\{ extChoice(getBasic(M(P)), getBasic(M(Q))) \right\} \right\rangle$

Four auxiliary functions are used in the equations, which are presented below. The auxiliary functions themselves use some other functions, which we present first; these functions are standard sequence operators specialised to apply to the sequence component of an *eTrace*.

- $startSet(eTrace) = last(first(eTrace))$
- $frontSeq(eTrace) = front(first(eTrace))$
- $lengthSeq(eTrace) = length(first(eTrace))$
- $getSeq(eTrace) = first(eTrace)$

The function *prefixingBasic* receives an event and an Enhanced Trace, and appends the new event to the end of the valid structure of the *eTrace*. If the last set of the valid structure has only one sequence, the event is appended to this sequence, otherwise, a new set is created, containing a sequence with the new event and a pointer.

- $prefixingBasic(event, eTrace) =$
  if $\#(startSet(eTrace)) == 1$ then
  $(frontSeq(eTrace)^\frown\langle\{\langle event\rangle^\frown\langle startSet(eTrace)\rangle\}\rangle, \varnothing, \varnothing)$
  else
  $(getSeq(eTrace)^\frown\langle\{\langle event, lengthSeq(eTrace)\rangle\}\rangle, \varnothing, \varnothing)$

To access the Enhanced Trace of a Basic Process, we use the function *getBasic*. This function works in this context because a Basic Process has only one Enhanced Trace.

- $getBasic(\langle\{eTrace\}\rangle) = eTrace$

For sequential composition, we have the function *seqComp*. This function receives two Enhanced Traces, and returns a new Enhanced Trace, with the valid structures of the arguments appended and their pointers updated. To correctly model the composition, all occurrences of $SKIP$ in the first *eTrace* are replaced by a pointer; $\mathbb{N}^+$ stands for the positive naturals.

- $seqComp(eTrace1, eTrace2) =$
$$\left(getSeq(eTrace2)^\frown seqCompForSets \left(\begin{array}{l} getSeq(eTrace1), \\ getSeq(eTrace2) \end{array} \right), \varnothing, \varnothing \right)$$
- $seqCompForSets(\langle\rangle, seq2) = \langle\rangle$
- $seqCompForSets(\langle set\rangle^\frown seq1, seq2) =$
  $\langle seqCompInSet(set, seq2)\rangle^\frown seqCompForSets(seq1, seq2)$
- $seqCompInSet(\varnothing, seq2) = \varnothing$
- $seqCompInSet(\{seqInSet\} \cup set, seq2) =$
  if $last(seqInSet) == SKIP$ then
  $\{front(seqInSet)^\frown\langle length(seq2)\rangle\} \cup seqCompInSet(set \setminus \{seqInSet\}, seq2)$
  else if $last(seqInSet) \in \mathbb{N}^+$ then
  $\{front(seqInSet)^\frown\langle last(seqInSet) + length(seq2)\rangle\} \cup$
  $seqCompInSet(set \setminus \{seqInSet\}, seq2)$
  else
  $\{seqInSet\} \cup seqCompInSet(set \setminus \{seqInSet\}, seq2)$

The function *extChoice* receives two Enhanced Traces, and returns an Enhanced Trace modelling an external choice. The front of the valid sequences of the arguments is appended, and their pointers updated. The last set of the new valid sequence is the union of the last sets of the valid sequences of the arguments, with their pointers also updated.

- $extChoice(eTrace1, eTrace2) =$
$$\left(\begin{array}{l} frontSeq(eTrace1)^\frown \\ shift(frontSeq(eTrace2), length(frontSeq(eTrace1)))^\frown \\ joinInitials \left(\begin{array}{l} startSet(eTrace1), \\ startSet(eTrace2), \\ length(frontSeq(eTrace1)) \end{array} \right) \end{array} , \varnothing, \varnothing \right)$$
- $shift(\langle\rangle, val) = \langle\rangle$
- $shift(\langle set\rangle^\frown seq, val) = \langle shiftSet(set, val)\rangle^\frown shift(seq, val)$
- $shiftSet(\varnothing, val) = \varnothing$
- $shiftSet(\{seq\} \cup set, val) =$
  if $last(seq) \in \mathbb{N}^+$ then
    $\{front(seq)^\frown\langle last(seq) + val\rangle\} \cup shiftSet(set \setminus \{seq\}, val)$
  else
    $\{seq\} \cup shiftSet(set \setminus \{seq\}, val)$
- $joinInitials(set1, set2, val) = \langle set1 \cup shiftSet(set2, val)\rangle$

Example 8 The calculation of M($Ex4a$) is shown below.

$$M(Ex4a) = \left\langle \left\{ \left(\left\langle \left\{ \langle 0\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(b \to Ex4a) = \left\langle \left\{ \left(\left\langle \left\{ \langle b, 0\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(a \to b \to Ex4a) = \left\langle \left\{ \left(\left\langle \left\{ \langle a, b, 0\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

$$M(Ex4a) = M(a \to b \to Ex4a)$$

We differentiate between the process $Ex4a$ and its recursive call. For the sequential composition $Ex4b\,;\,Ex4a$, we have the following metadata.

$$\mathrm{M}(Ex4b\,;\,Ex4a) = \left\langle \left\{ \left(\left\langle \left\{ \langle a, b, 0\rangle \right\}, \left\{ \langle c, d, 1\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \right\rangle$$

□

We assume that the boolean values in guards and conditionals are always true; in those cases we simply keep the metadata of P. In the conditional, if the processes P and Q are not equivalent, the strategy returns the possibility of nondeterminism. With this approach, we record behaviours for the processes that may not be actually possible. The addition of behaviours, however, can only lead to nondeterminism, never remove it. So,

as already explained, it is possible that we indicate a potential nondeterminism that does not exist, but a process defined to be deterministic is guaranteed to be so.

We now present the equations to calculate the metadata of the Composite Processes; X is a set of events, and i is a fresh integer, different from zero.

Definition 13 (Metadata of the Composite Processes)

- $M(P\,;\,Q) = \; M(Q)^\frown addPointer(M(P), length(M(Q)))^\frown$
  $\langle\{(\langle\{\langle SKIP\rangle\}\rangle, \varnothing, \{length(M(P)) + length(M(Q))\})\}\rangle$
- $M(P \,\square\, Q) = M(P)^\frown shiftM(M(Q), length(M(P)))^\frown$
  $\langle\{(\langle\{\langle SKIP\rangle\}\rangle, \varnothing, \{length(M(P)), length(M(P)) + length(M(Q))\})\}\rangle$
- $M(P \sqcap Q) = M(P)$
- $M(P \;|||\; Q) = \; front(M(P))^\frown shiftM(front(M(Q)), length(front(M(P))))^\frown$
  $\langle last(M(P)) \cup shiftSet(last(M(Q)), length(front(M(P))))\rangle^\frown$
  $\langle\{(\langle\{\langle SKIP\rangle\}\rangle, \varnothing, \{length(M(P)) + length(M(Q)) - 1\})\}\rangle$
- $M(P[\![X]\!]Q) = applyAddSync(front(M(P)), X, i)^\frown$
  $applyAddSync(shiftM(front(M(Q)), length(front(M(P)))), X, -i)^\frown$
  $\langle addSyncToSet(last(M(P)), X, i) \cup$
  $addSyncToSet(shiftSet(last(M(Q)), length(front(M(P)))), X, -i)\rangle^\frown$
  $\langle\{(\langle\{\langle SKIP\rangle\}\rangle, \varnothing, \{length(M(P)) + length(M(Q)) - 1\})\}\rangle$
- $M(P \setminus X) = remove(M(P), X)$

The auxiliary functions used in the equations are presented below. For sequential composition, we need to add a new pointer to join the metadata of the two operands. To this end, we use the function *addPointer*. It adds a reference, in the leafs of the metadata of P, to the root of the metadata of Q; the existing pointers are also updated accordingly.

- $addPointer(\langle\rangle, val) = \langle\rangle$
- $addPointer(\langle set\rangle^\frown seq, val) = \langle addPointerSet(set, val)\rangle^\frown addPointer(seq, val)$
- $addPointerSet(\varnothing, val) = \varnothing$
- $addPointerSet(\{(fst, sec, trd)\} \cup set, val) =$
  if $trd == \varnothing$ then
  $\{(fst, sec, \{val\})\} \cup addPointerSet(set \setminus \{(fst, sec, trd)\}, val)$
  else
  $\{(fst, sec, shiftPointers(trd, val))\} \cup addPointerSet(set \setminus \{(fst, sec, trd)\}, val)$

To update the pointers of the metadata of a process, we use the function *shiftM*. This function receives a metadata and an integer value, and adds this value to all pointers in the metadata. The functions *shiftSet* and *shiftPointers* update the pointers of a given set of Enhanced Traces, and of an Enhanced Trace itself, and can be applied directly.

- $shiftM(\langle\rangle, val) = \langle\rangle$
- $shiftM(\langle set\rangle^\frown seq, val) = \langle shiftSet(set, val)\rangle^\frown shiftM(seq, val)$
- $shiftSet(\varnothing, val) = \varnothing$
- $shiftSet(\{(fst, sec, trd)\} \cup set, val) = \{(fst, sec, shiftPonters(trd, val))\} \cup shiftSet(set \setminus \{(fst, sec, trd)\}, val)$
- $shiftPonters(\varnothing, val) = \varnothing$
- $shiftPonters(\{p\} \cup set, val) = \{p + val\} \cup shiftPonters(set \setminus \{p\}, val)$

To add a new synchronisation to a metadata, we use the function *applyAddSync*. This function receives a metadata, a set of events, and an integer value, and updates all synchronisation sets in the metadata. As is the case with the update of pointers, we can use the functions *addSyncToSet* and *addSync* directly to a set of Enhanced Traces, and to an Enhanced Trace itself.

- $applyAddSync(\langle\rangle, X, id) = \langle\rangle$
- $applyAddSync(\langle set\rangle^\frown seq, X, id) = \langle addSyncToSet(set, X, id)\rangle^\frown applyAddSync(seq, X, id)$
- $addSyncToSet(\varnothing, X, id) = \varnothing$
- $addSyncToSet(\{eTrace\} \cup set, X, id) = \{addSync(eTrace, X, id)\} \cup addSyncToSet(set \setminus \{eTrace\}, X, id)$
- $addSync(eTrace, X, id) = (first(eTrace), second(eTrace) \cup \{(id, X)\}, third(eTrace))$

To remove a set of events from a metadata, we use the function *remove*. This function receives a metadata and a set of events, and it removes the events contained in the received set from the metadata; the pointers are not affected.

- $remove(\langle\rangle, X) = \langle\rangle$
- $remove(\langle set\rangle^\frown seq, X) = \langle removeFromSet(set, X)\rangle^\frown remove(seq, X)$
- $removeFromSet(\varnothing, X) = \varnothing$
- $removeFromSet(\{eTrace\} \cup set, X) = \{removeFromTuple(eTrace, X)\} \cup removeFromSet(set \setminus \{eTrace\}, X)$

- $removeFromTuple(eTrace, X) = \begin{pmatrix} removeSeq(first(eTrace), X), \\ removeSet(second(eTrace), X), \\ third(eTrace) \end{pmatrix}$
- $removeSeq(\langle\rangle, X) = \langle\rangle$
- $removeSeq(\langle set\rangle^\frown seq, X) = \langle removeSetOfSeq(set, X)\rangle^\frown removeSeq(seq, X)$
- $removeSetOfSeq(\varnothing, X) = \varnothing$
- $removeSetOfSeq(\{seq\} \cup set, X) = \{removeSeqOfEv(seq, X)\} \cup removeSetOfSeq(set \setminus \{seq\}, X)$
- $removeSeqOfEv(\langle\rangle, X) = \langle\rangle$
- $removeSeqOfEv(\langle a\rangle^\frown t, X) =$ if $a \in X$ then $removeSeqOfEv(t, X)$ else $\langle a\rangle^\frown removeSeqOfEv(t, X)$
- $removeSet(\varnothing, X) = \varnothing$
- $removeSet(\{(id, evSet)\} \cup s, X) = \{(id, evSet \setminus X)\} \cup removeSet(s \setminus \{(id, evSet)\}, X)$

For sequential composition we concatenate the metadata of the two operands, update their pointers, and add a new composition tuple to mark the composition. For external choice, we also concatenate the metadata of the operands, update their pointers, and add a new composition tuple, but in a different way. Instead of creating a sequence of references, from the composition tuple to the first operand, and then to the second, as is the case for sequential composition, the composition tuple becomes the parent of both operands in a tree structure. When creating the metadata of a process resulting from an internal choice, we simply keep the metadata of one of the operands, since the composition is only deterministic if both operands are equivalent.

For interleaving, we concatenate the front of the metadata of both operands, and join their last sets; the pointers are updated to reflect the concatenation. Multiple tuples in a same set represent the parallelism we want to capture. As a final step, we add a composition tuple to mark the composition. The calculation for a generalised parallel is similar to that of an interleaving, but we also add the new synchronisation to the metadata. For hiding, we simply remove the elements in X from the metadata.

Example 9 Considering the processes $Ex9a = a \rightarrow b \rightarrow c \rightarrow Ex9a$, and $Ex9b = Ex9a[\![\{b\}]\!]Ex9a$, we calculate M($Ex9b \setminus \{b\}$).

$$\mathrm{M}(Ex9b) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, b, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ \, b \, \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle\, a, b, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(-1, \left\{ \, b \, \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \right\rangle$$
$$\left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ \, 1 \, \right\} \right) \right\}$$

$$\mathrm{M}(Ex9b \setminus \{b\}) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(1, \{\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle\, a, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(-1, \{\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \right\rangle$$
$$\left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ \, 1 \, \right\} \right) \right\}$$

□

The composition tuples, introduced by many equations, are used to both record the compositions themselves, as is the case for external choice, and to keep their order. If we do not introduce a composition tuple after an interleaving, for example, we can have a set with Enhanced Traces of many compositions, after two or more interleavings in sequence.

If a synchronisation introduces deadlock, or if a synchronisation channel is hidden, there is the possibility that our strategy considers invalid behaviours of the process. This, however, can only introduce nondeterminism, never remove it.

Example 10 We consider the following processes.

$$Ex10a = b \rightarrow a \rightarrow c \rightarrow d \rightarrow Ex10a \qquad Ex10b = Ex9a[\![\{a, b\}]\!]Ex10a$$

The process $Ex10b$ is deterministic, because it is deadlocked from the start. Our strategy, however, predicts a nondeterministic behaviour because both $Ex9a$ and $Ex10a$ offer event c to the environment, which never happens in $Ex10b$. □

Tuples of a set in M are equivalent, symbolised by $\equiv$, if their valid structures are equal, they have the same meaningful synchronisations, with equivalent pairs, and their pointers reference equivalent sets. A synchronisation is meaningful if it involves at least one of the events in the valid structure of the pair.

Definition 14 (Meaningful Synchronisations) *Given an Enhanced Trace of the form (Struct, SetOfSyncs, Pointers), a pair sync* $\in$ *SetOfSyncs is a meaningful synchronisation if* $sync \cap ran(Struct) \neq \varnothing$.

We abuse the notation here, with sync standing for its set of events, and ran(Struct) representing the set of all events present in Struct.

Example 11 We consider the following metadata.

$$\mathrm{M}(Ex11a) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, b, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ \, r \, \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle\, x, y, SKIP \,\right\rangle, \right\} \right\rangle, \left\{ \left(-1, \left\{ \, r \, \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \left\langle \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ \, 1 \, \right\} \right) \right\} \right\rangle \right.$$

$$\mathrm{M}(Ex11b) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, b, c, 0 \,\right\rangle, \right\} \right\rangle, \left\{ \left(2, \left\{ \, r \, \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle\, x, y, SKIP \,\right\rangle, \right\} \right\rangle, \left\{ \left(-2, \left\{ \, r \, \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \left\langle \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ \, 1 \, \right\} \right) \right\} \right\rangle \right.$$

$$\mathrm{M}(Ex11c) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, b, c, 0 \,\right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle\, x, y, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \left\langle \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ \, 1 \, \right\} \right) \right\} \right\rangle \right.$$

The three metadata given are all equivalent, since the only difference between them is their synchronisation sets, which is irrelevant in this case. The synchronisations of $Ex11a$ and $Ex11b$ are not meaningful, as they do not affect the valid structures, and $Ex11c$ does not have synchronisations. □

3.3 Composition Rules

The algorithms that verify if the compositions are deterministic take the metadata of the operands as input, and sometimes also a set of events, and return true if the given composition is guaranteed to be deterministic, and false if we have an inconclusive result. We present here all the algorithms for the supported compositions; sequential composition does not have an algorithm, since it cannot introduce nondeterminism.

All algorithms need to compare events to know if they are equal or not. If there is no communication, this is a trivial matter. If data is being communicated, however, we need to consider all possible values that the types of channel can assume, which is very expensive. To avoid this problem, we deal with communications symbolically, by making an overestimation of the values that are being communicated at any given point, and always including the case that may lead to nondeterminism.

To make explicit comparisons, we use the function *EqEvents*, which takes two events, modelled as sequences, and a function that indicates the compositions being analysed. For example, for the event $inOut?val1!val2$, we have $\langle inOut, ?val1, !val2 \rangle$. If one of the events has an input, through "?", *EqEvents* assumes that the events are equal, since an input can assume any value of its type.

- EqEvents($\langle\rangle$, $\langle\rangle$, fun) = true

- EqEvents($\langle p1 \rangle^\frown$ ev1, $\langle p2 \rangle^\frown$ ev2, fun) =
  if head(p1) == "?" $\vee$ head(p2) == "?" then
    true
  else
    fun(p1,p2) $\wedge$ EqEvents(ev1,ev2,fun)

The function *EqEvents* checks whether the events are equal, since this is usually a condition for nondeterminism. At some points in our strategy, however, the metadata of two processes need to be equivalent to avoid nondeterminism. In this case, we use an alternative version of *EqEvents*, which checks whether the events are different.

We also deal with parameters symbolically, though indirectly. Since Basic Processes can only be tail recursive in our strategy, we know that the structure of the process does not change, only the value of the parameters might change. The parameters themselves are only used in communications, which are analysed symbolically.

3.3.1 External Choice

The external choice, with our restrictions, can only introduce nondeterminism if its two operands have at least one common initial event, since these are their only points of interaction. The limited interaction results from the nature of the Basic Processes, which are not able to reference processes other than themselves; the initial events of a Composite Process derive from the initials of its Basic Processes. In this scenario, the composition is deterministic only if the two processes have the same behaviour after every common initial event.

To check an external choice, we use Algorithm 1. This algorithm takes the metadata of two processes, P and Q, and compares the initial events of the tuples of each operand; the process names stand for their metadata. If there is a common initial event, then the environment has no control over how the choice is resolved when synchronising on this event, so the composition is deterministic only if the operands are equivalent.

Algorithm 1 External Choice (P,Q)

```
1: for each elemP ∈ last(P), elemQ ∈ last(Q) do
2:   for each seqP ∈ getStartSet(elemP), seqQ ∈ getStartSet(elemQ) do
3:     if EqEvents(head(seqP),head(seqQ),EqExtChoiceStart) ∧ ¬(P ≡ Q) then
4:       return false
5: return true
```

The function *getStartSet* returns a set with the initial valid sequences of an Enhanced Trace. This set can be the last set of the valid structure of the tuple, or a set containing valid sequences from other tuples, if the argument is a composition tuple. For a composition tuple, we simply follow the pointers until a non-composition tuple is found.

We use *EqExtChoiceStart* to compare events in an external choice; *Val* stands for an explicit value, as opposed to a variable. For this operator, unless we have two explicit values, we always assume that they are equal, which is a precondition for nondeterminism. When comparing the metadata of two processes, $P \equiv Q$, we check if they model the same behaviour, that is, the same events are offered in the same order.

- EqExtChoiceStart(p1,p2) = if $p1 \in Val \land p2 \in Val$ then
  if $p1 == p2$ then *true* else *false*
  else
  true

3.3.2 Internal Choice

An internal choice only results in a deterministic process if its operands have the same behaviour. For this operator, we simply check if the metadata of both operands are equivalent, as can be seen in Algorithm 2.

Algorithm 2 Internal Choice (P,Q)

```
1: if ¬(P ≡ Q) then
2:     return false
3: return true
```

3.3.3 Parallelism

We deal with parallelism in two forms: interleaving and generalised parallel. First, we discuss how interleaving can introduce nondeterminism. Afterwards, we present our considerations about generalised parallel. Finally, we show the algorithm for the verification of parallel compositions.

Differently from external and internal choice, with interleaving, as well as with the other parallel operators of CSP, both operands execute at the same time, so we must take into account all of their events, not only the initials. With interleaving, we need to consider that when one of its operands is offering a specific event to the environment, the other operand can be offering any of its events.

The condition for a composition using interleaving to be deterministic is that, after each event in common to both processes, the composition needs to offer the same events to the environment, no matter which process performs the event, so the environment does not observe any different behaviour.

Example 12 We consider the following processes.

$$Ex12a = a \rightarrow b \rightarrow Ex12a \qquad Ex12d = Ex12a \;|||\; Ex12b$$

$Ex12b = a \rightarrow Ex12b$ $\quad$ $Ex12e = Ex12b \;|||\; Ex12c$

$Ex12c = b \rightarrow Ex12c$ $\quad$ $Ex12f = Ex12a \;|||\; Ex12e$

The process $Ex12d$ is nondeterministic, because after performing event a, the environment can synchronise on either a again or on a or b, depending on whether a was performed by $Ex12a$ or $Ex12b$. The process $Ex12e$ is deterministic, because the alphabets of its components are disjoint, so there are no events in common. The composition in $Ex12f$ is deterministic, because, although there is an intersection of the alphabets, events a and b are always available to the environment. □

The generalised parallel operator allows us to have parallelism with synchronisations. The events that are not in the synchronisation set are analysed in a similar way to what is done with interleaving, and the events in the synchronisation set cannot introduce nondeterminism on their own, because each synchronised event happens only once and both operands engage in this event.

Example 13 The process $Ex13 = Ex12a[\![\{a\}]\!]Ex12b$, differently from $Ex12d$, is deterministic, because the event a is in the synchronisation set, so, after it occurs, the only possibility for the parallel composition is to offer event b. □

We use the same algorithm, Algorithm 3, for the two forms of parallelism discussed. It receives the processes being composed, P and Q, and the synchronisation set, X. For interleaving, the synchronisation set is empty. In Figure 11 we present a fluxogram that illustrates its execution; this fluxogram is used to incrementally explain the execution of the algorithm.

Algorithm 3 Parallelism (P,Q,X)

```
1: avEvents = availableEvents(P,Q,X)
2: localStatesP = getLocalStates(P)
3: localStatesQ = getLocalStates(Q)
4: for each stateP ∈ localStatesP, stateQ ∈ localStatesQ do
5:    if first(stateP) == first(stateQ) ∧ first(stateP) ∉ X then
6:       nextStateP = second(stateP) ∪ {first(stateQ)} ∪ avEvents
7:       nextStateQ = second(stateQ) ∪ {first(stateP)} ∪ avEvents
8:       if nextStateP ≠ nextStateQ then
9:          return false
10: if ¬extChoiceWithND(P,Q,X) ∨ ¬extChoiceWithND(Q,P,X) then
11:    return false
12: return true
```

Algorithm 3 iterates over all pairs of local states of P and Q, which are the states of each Basic Process that composes P and Q. If there is a local state in P and one in Q

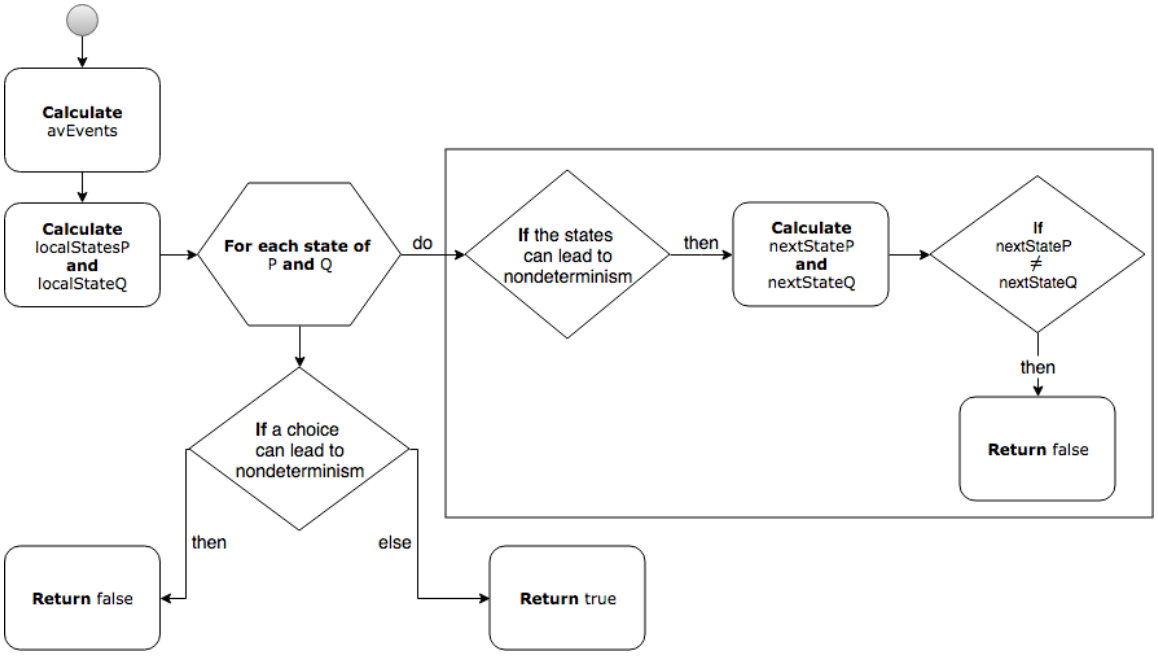

Figure 11 – Fluxogram of Parallelism(P,Q,X)

in which both offer the same event, line 5, if the processes offer different events after its occurrence, line 8, then we have a potential source of nondeterminism, line 9. We now describe in detail each part of the algorithm.

We initially define *avEvents*, line 1, which is the set that contains the events that are always available to the environment, in the given composition. This set is used in the calculation of which events are available to the environment after either P or Q evolves, when both offer the same event, lines 6 and 7.

To calculate *avEvents*, we use *availableEvents*, shown in Algorithm 4. This function initially calculates the events that are always available to the environment in each operand, lines 1 and 2. These events belong to Basic Processes of the form $P = ev \rightarrow P$, and do not change the state of the composition.

Algorithm 4 availableEvents(P,Q,X)

1: $localAvailableEventsP = localAvailableEvents(P,X)$
2: $localAvailableEventsQ = localAvailableEvents(Q,X)$
3: $finalSet = first(localAvailableEventsP) \cup first(localAvailableEventsQ)$
4: $needToCheckP = second(localAvailableEventsP)$
5: $needToCheckQ = second(localAvailableEventsQ)$
6: ***for*** $each\ evP \in needToCheckP,\ evQ \in needToCheckQ$ ***do***
7: ***if*** $evP == evQ$ ***then***
8: $finalSet = finalSet \cup \{evP\}$
9: ***return*** $finalSet$

The function *localAvailableEvents* returns a pair of sets of events. The first set contains events that are always available, and thus are directly added to *finalSet*, line 3. The second set contains events that, although always available in their individual process, can be restricted by the synchronisation set, lines 4 and 5. For these events to be added to *finalSet*, they must be always available in both sets, lines 6 to 8.

Example 14 We consider the following processes, and their metadata.

$$Ex14a = a \rightarrow b \rightarrow Ex14a \qquad Ex14e = Ex14a \;|||\; Ex14b$$
$$Ex14b = c \rightarrow Ex14b \qquad Ex14f = Ex14e \;|||\; Ex14c$$
$$Ex14c = d \rightarrow SKIP \qquad Ex14g = Ex14f \;|||\; Ex14d$$
$$Ex14d = e \rightarrow Ex14d$$

$$\mathrm{M}(Ex14g) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle a, b, 0 \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle c, 0 \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 1 \right\} \right), \\ \left(\left\langle \left\{ \left\langle d, SKIP \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 2 \right\} \right), \\ \left(\left\langle \left\{ \left\langle e, 0 \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 3 \right\} \right) \end{Bmatrix} \right\rangle$$

$$Ex14h = x \rightarrow Ex14h \qquad Ex14k = Ex14h [\![\{x\}]\!] Ex14i$$
$$Ex14i = x \rightarrow y \rightarrow Ex14i \qquad Ex14l = Ex14k \;|||\; Ex14j$$
$$Ex14j = e \rightarrow f \rightarrow Ex14j$$

$$\mathrm{M}(Ex14l) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle x, 0 \right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ x \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle x, y, 0 \right\rangle, \right\} \right\rangle, \left\{ \left(-1, \left\{ x \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 1 \right\} \right), \\ \left(\left\langle \left\{ \left\langle e, f, 0 \right\rangle, \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 2 \right\} \right) \end{Bmatrix} \right\rangle$$

If we execute Algorithm 3 for Parallelism($Ex14g$,$Ex14l$,{e}), we have $avEvents = \{c\}$. The Enhanced Traces with the sequences $\langle a, b, 0\rangle$, $\langle x, y, 0\rangle$, and $\langle e, f, 0\rangle$ are discarded for having more than one event. The tuple with $\langle d, SKIP\rangle$ is discarded for not being

recursive. The Enhanced Traces with the sequences $\langle x, 0\rangle$, and $\langle e, 0\rangle$ are discarded due to their synchronisations, the former with the valid sequence $\langle x, y, 0\rangle$, and the latter with the synchronisation being introduced in this composition, through the set {e}. □

To capture the events that are always available in a process that already had its metadata calculated, we use *localAvailableEvents*, shown in Algorithm 5; M is a metadata, and X is a set of events. This function returns two sets, *finalSet*, which contains the events that are guaranteed to always be available, and *needToCheck*, containing the events that may be restricted by the synchronisation set X.

Algorithm 5 localAvailableEvents(M,X)

```
1:  finalSet = needToCheck = ∅
2:  tuples = parallelTuples(M)
3:  for each elem ∈ tuples do
4:     validChoice = true
5:     finalSetTemp = needToCheckTemp = ∅
6:     for each seq ∈ getStartSet(elem) do
7:        if length(seq) = 2 ∧ last(seq) == 0 then
8:           if evAlwaysAvailable(elem,tuples) then
9:              if head(seq) ∉ X then
10:                finalSetTemp = finalSetTemp ∪ {head(seq)}
11:             else
12:                needToCheckTemp = needToCheckTemp ∪ {head(seq)}
13:       else
14:          validChoice = false
15:    if validChoice then
16:       finalSet = finalSet ∪ finalSetTemp
17:       needToCheck = needToCheck ∪ needToCheckTemp
18: return (finalSet,needToCheck)
```

First, we gather the Basic Processes that are in parallel in the *tuples* set, line 2, by removing the unnecessary composition tuples; for the process $Ex14g$, we would remove all composition tuples, gathering the four non-empty Enhanced Traces of the metadata. Then, for each gathered tuple, we check if it is a Basic Process of the form $P = ev \rightarrow P$, line 7. If the tuple is of the required form, and its event is not restricted by the internal synchronisations of the process, line 8, it may be added to either *finalSet* or *needToCheck*, lines 9 to 12.

A Basic Process in *tuples* can have an external choice of its own, with multiple valid sequences in the last set of its valid structure, which is captured by *getStartSet* in line 6. If this is the case, unless all valid sequences offer exactly one event and then recurse, the valid sequences that follow our pattern are being restricted, and are not always available. We check for this possibility with the boolean *validChoice*, lines 4, 14 and 15.

Example 15 We consider the following processes.

$$Ex15a = (a \rightarrow Ex15a) \,\Box\, (b \rightarrow c \rightarrow SKIP) \qquad Ex15d = Ex15a \,;\, Ex15b$$
$$Ex15b = d \rightarrow SKIP \qquad Ex15e = Ex15d \,|||\, Ex15c$$
$$Ex15c = (e \rightarrow Ex15c) \,\Box\, (f \rightarrow Ex15c)$$

$$\mathrm{M}(Ex15e) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, d, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, 0 \,\right\rangle, \left\langle\, b, c, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \left\{ 1 \right\} \right) \\ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ 2 \right\} \right), \\ \left(\left\langle \left\{ \left\langle\, e, 0 \,\right\rangle, \left\langle\, f, 0 \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{ 3 \right\} \right) \end{Bmatrix} \right\rangle$$

If we execute $localAvailableEvents(Ex15e, \varnothing)$, the result is $(\{e, f\}, \varnothing)$. The process $Ex15c$ in the composition presents a more general form of $P = ev \rightarrow P$, offering more than one event, while complying with our requirement that the state must not change. The event a, in $a \rightarrow Ex15a$, is not always available because of the external choice in the process $Ex15a$, which leads to *validChoice* being false. □

To remove the unnecessary composition tuples in *localAvailableEvents*, line 2, we use the function *parallelTuples*, shown in Algorithm 6. This function starts at the root of the given metadata, line 1, and gathers all non-composition tuples that can be reached from this node that do not introduce restrictions, in the set *returnTuples*; the root itself can be a non-composite node.

The function *getFromSet*, line 4, returns a random tuple from *tempTuples*, which is then checked to see if it is a composition tuple, lines 6 to 14. Note that, in line 8, we check if the composite node has more than one pointer, which indicates an external choice. We do this because the presence of an external choice can restrict an event that otherwise would be always available. The function *getChildren* returns a set with all tuples in the sets in M referenced by the given pointers.

Example 16 If we apply *parallelTuples* to $M(Ex15e)$, the result is the following:

$$returnTuples = \begin{Bmatrix} \left(\left\langle \left\{ \left\langle\, a, 0 \,\right\rangle, \left\langle\, b, c, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \left\{ 1 \right\} \right), \\ \left(\left\langle \left\{ \left\langle\, e, 0 \,\right\rangle, \left\langle\, f, 0 \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \end{Bmatrix}$$

The root of the metadata is removed, since it is a composition tuple. In its child, we have one composition tuple, which we also remove, and one non-composition tuple, added to *returnTuples*. The child of the second composition tuple has one non-composition tuple,

Algorithm 6 parallelTuples(M)

```
1:  tempTuples = last(M)
2:  returnTuples = ∅
3:  while tempTuples ≠ ∅ do
4:      elem = getFromSet(tempTuples)
5:      tempTuples = tempTuples \ {elem}
6:      if size(startSet(elem)) == 1 then
7:          seq = getFromSet(getStartSet(elem))
8:          if seq == ⟨SKIP⟩ ∧ size(third(elem)) == 1 then
9:              pointer = third(elem)
10:             tempTuples = tempTuples ∪ getChildren(pointer,M)
11:         else
12:             returnTuples = returnTuples ∪ {elem}
13:     else
14:         returnTuples = returnTuples ∪ {elem}
15: return returnTuples
```

also added to *returnTuples*. If, instead of the interleaving in $Ex15e$, we had an external choice, as in $Ex16 = Ex15d \;\Box\; Ex15c$, the function *parallelTuples* would simply return the root of the metadata, which leads *localAvailableEvents* to return $(\varnothing, \varnothing)$; M(Ex16) is shown below.

$$\mathrm{M}(Ex16) = \left\langle \begin{array}{l} \left\{ \left(\left\langle \left\{ \left\langle\, d, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle\, a, 0 \,\right\rangle, \left\langle\, b, c, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \left\{\, 1 \,\right\} \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{\, 2 \,\right\} \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle\, e, 0 \,\right\rangle, \left\langle\, f, 0 \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{\, 3, 4 \,\right\} \right) \right\} \end{array} \right\rangle$$

□

Finally, to check, in Algorithm 5, if an event is restricted by an internal synchronisation, we use *evAlwaysAvailable*, shown in Algorithm 7. This function checks, for each meaningful synchronisation of an event that has the potential to be always available, line 3, if there is another event, that is also always available, and that enables the synchronisation to happen unrestricted, lines 4 to 12.

Example 17 We consider the following processes.

$$Ex17a = a \rightarrow Ex17a \qquad Ex17c = Ex17a \;\Box\; Ex17b$$

$$Ex17b = b \rightarrow Ex17b \qquad Ex17d = Ex17c[\![\{a\}]\!]Ex17a$$

Algorithm 7 evAlwaysAvailable(elem,set)

```
1:  seq = getFromSet(getStartSet(elem))
2:  for each syncTuple ∈ second(elem) do
3:    if head(seq) ∈ second(syncTuple) then
4:      for each otherElem ∈ set do
5:        for each otherSyncTuple ∈ second(otherElem) do
6:          if first(syncTuple) == first(otherSyncTuple)*-1 then
7:            if size(startSet(otherElem)) == 1 then
8:              otherSeq = getFromSet(getStartSet(otherElem))
9:              if seq ≠ otherSeq then
10:               return false
11:           else
12:             return false
13: return true
```

If we execute *localAvailableEvents* with M($Ex17d$), shown below, as an argument, *evAlwaysAvailable* is called once, with the first argument being the tuple representing $Ex17a$, and the second argument containing this tuple and the composition tuple representing the external choice in $Ex17c$. The function *evAlwaysAvailable* returns false in this case, because, although there can be a synchronisation between $Ex17a$ and $Ex17c$ in event a, this synchronisation depends on the outcome of the external choice.

$$\mathrm{M}(Ex17d) = \left\langle \left\{ \begin{array}{l} \left(\left\langle \left\{ \left\langle a, 0 \right\rangle \right\} \right\rangle, \left\{ \left(1, \left\{ a \right\} \right) \right\}, \varnothing \right) \\ \left(\left\langle \left\{ \left\langle b, 0 \right\rangle \right\} \right\rangle, \left\{ \left(1, \left\{ a \right\} \right) \right\}, \varnothing \right) \\ \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ a \right\} \right) \right\}, \left\{ 1, 2 \right\} \right), \\ \left(\left\langle \left\{ \left\langle a, 0 \right\rangle \right\} \right\rangle, \left\{ \left(-1, \left\{ a \right\} \right) \right\}, \varnothing \right) \\ \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 3 \right\} \right) \end{array} \right\} \right\rangle$$

□

With *avEvents* calculated, in Algorithm 3, we now proceed to capture the possible local states of both operands, lines 2 and 3; the current stage of Parallelism(P,Q,X) is shown, in green, in Figure 12. To this end, we use *getLocalStates*, shown in Algorithm 8. This function returns a set of pairs, with the their first element being an event, representing the event being offered to the environment at a given state, and the second element being a set of sets of events, representing what will be offered by the process to the environment, if the first event happens; we use a set of sets in order to capture different states in which the same event is offered.

The function *getLocalStates* checks for each tuple in M, lines 2, 3, 4 and 24, what events they offer, and what will be offered after these events, lines 8 to 23. For each valid

Algorithm 8 getLocalStates(M)

```
1:  localStates = ∅
2:  tuples = last(M)
3:  while tuples ≠ ∅ do
4:     elem = getFromSet(tuples)
5:     localSeq = first(elem)
6:     syncSet = second(elem)
7:     pointers = third(elem)
8:     while localSeq ≠ ⟨⟩ do
9:        for each seq ∈ last(localSeq) do
10:          if size(seq) > 1 then
11:             for each ev ∈ front(front(seq)) do
12:                if newEvent(ev,localStates) then
13:                   localStates = localStates ∪ {(ev,∅)}
14:                localStates = addState(ev,nextEvents(ev,seq,syncSet,M),localStates)
15:             ev = last(front(seq))
16:             end = last(seq)
17:             if newEvent(ev,localStates) then
18:                localStates = localStates ∪ {(ev,∅)}
19:             if end ∈ ℕ then
20:                localStates = addState(ev,nextEvets(ev,end,localSeq,syncSet,M),localStates)
21:             else if end == SKIP then
22:                localStates = addState(ev,nextEvents(ev,pointers,syncSet,M),localStates)
23:       localSeq = front(localSeq)
24:    tuples = tuples \ {elem} ∪ getChildren(pointers,M)
25: return localStates
```

sequence, lines 9 and 23, we initially check the events in its front, lines 11 to 14. If it is the first time the event has been captured, we add it to *localStates*, lines 12 and 13. We add what will be offered to the environment by the process in line 14; the function *addState* adds a set of events, its second argument, to the set of sets of events referenced by its first argument. The function *nextEvents* returns the events that a process, or part of a process, modelled through an Enhance Trace, offers to the environment after a given event happens; this function is detailed after Example 18.

We deal with the last event of the valid sequence in lines 15 to 22. If the valid sequence references another valid sequence, we capture its first event, line 20; if it ends in *SKIP* we follow the pointers of the tuple, line 22, and if it ends in $STOP$ nothing is added; the function *nextEvent* is overloaded.

Example 18 We consider the following processes.

$$Ex18a = (a \rightarrow Ex18a) \,\Box\, (b \rightarrow c \rightarrow SKIP) \qquad Ex18d = Ex18a \,;\, Ex18b$$

$$Ex18b = a \rightarrow d \rightarrow SKIP \qquad Ex18e = Ex18d \,\Box\, Ex18c$$

$$Ex18c = (c \rightarrow d \rightarrow e \rightarrow Ex18c) \,\Box\, (f \rightarrow Ex18c)$$

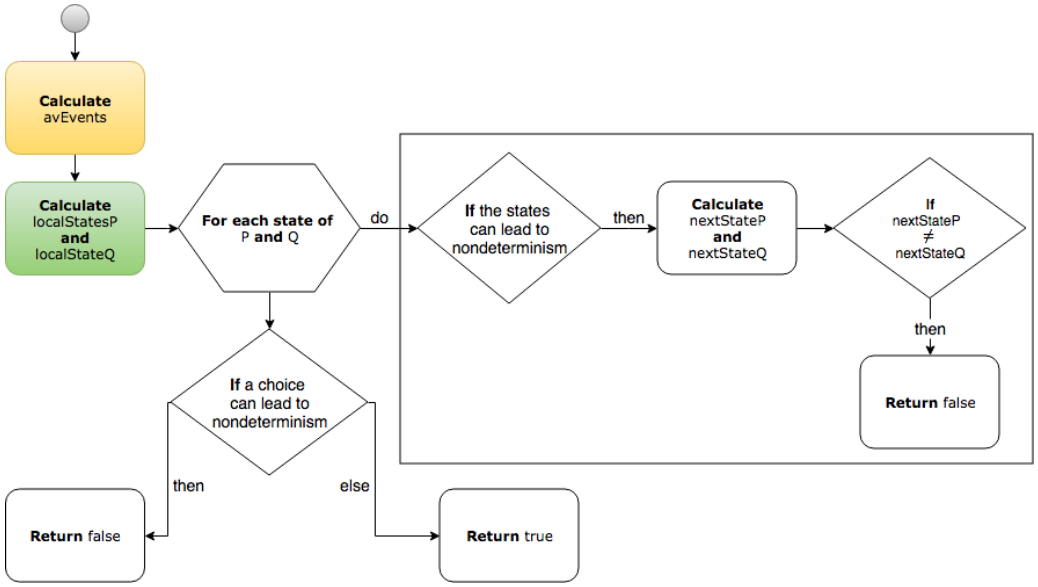

Figure 12 – Calculating the local states in Parallelism(P,Q,X)

$$\mathrm{M}(Ex18e) = \left\langle \begin{array}{l} \left\{ \left(\left\langle \left\{ \left\langle\, a, d, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle\, a, 0 \,\right\rangle, \left\langle\, b, c, SKIP \,\right\rangle \right\} \right\rangle, \varnothing, \left\{\, 1 \,\right\} \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{\, 2 \,\right\} \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle\, c, d, e, 0 \,\right\rangle, \left\langle\, f, 0 \,\right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle\, SKIP \,\right\rangle, \right\} \right\rangle, \varnothing, \left\{\, 3, 4 \,\right\} \right) \right\} \end{array} \right\rangle$$

If we execute *getLocalStates* with $\mathrm{M}(Ex18e)$ as the argument, the result is:

$$\left\{ \begin{array}{l} (a, \{\{a, b\}, \{d\}\}), (b, \{\{c\}\}), (c, \{\{a\}, \{d\}\}), (d, \{\{e\}\}), (e, \{\{c, f\}\}), \\ (f, \{\{c, f\}\}) \end{array} \right\}$$

There are two occurrences of event a, one in $Ex18a$, which leads to a recursion, and one in $Ex18b$. We capture the two states, with the former leading to the offer of the initial events of $Ex18a$, and the latter leading to the offer of event d. Event b simply leads to event c. Event c also has two occurrences, the first, in $Ex18a$, pointing to the initials of $Ex18b$, and the second, in $Ex18c$, to event d. Of the two occurrences of event d, one leads to $SKIP$ and is not a trigger to a sequential composition, so no event is offered after it happens, and the other leads to event e. Events e and f both lead to the initials of $Ex18c$. □

We have three functions *nextEvents*, shown below. The first one is used in line 14 of *getLocalStates* and simply gets the next event in the given valid sequence, using

getNextFromSeq; *getLocalStates* can differentiate between multiple occurrences of an event in a sequence, such as in $a \rightarrow b \rightarrow a \rightarrow SKIP$. The second one is used in line 20 of *getLocalStates* and gets the initial events of the valid sequences in the set in localSeq, a valid structure, referenced by index; *getFromSeq* returns the needed set, and *getHeads* extracts the events. The third one is used in line 22 of *getLocalStates* and get the initials of the Enhanced Traces of the sets in M referenced in pointers; *getFromSeq* is overloaded to work with the metadata as well as valid structures.

1. nextEvents(ev,seq,sycnSet,M) =
   {getNextFromSeq(ev,seq)} ∪ syncEvents(ev,syncSet,M)

2. nextEvents(ev,index,localSeq,sycnSet,M) =
   getHeads(getFromSeq(index,localSeq)) ∪ syncEvents(ev,syncSet,M)

3.1. nextEvents(ev,∅,sycnSet,M) = syncEvents(ev,syncSet,M)

3.2. nextEvents(ev,{p} ∪ pointers,sycnSet,M) =
   getHeads(getFromSeq(index,M)) ∪ nextEvents(ev,pointers \ {p},sycnSet,M)

In all versions of *nextEvents*, we use the function *syncEvents*, shown in Algorithm 9. This function is used to get the events in the metadata that are guaranteed to be available due to internal synchronisations. These events are then joined by those gathered by the particular method of each *nextEvents* function.

For each meaningful synchronisation in the given synchronisation set, we check all Enhanced Traces that lead to the handshake, lines 2 to 13; *otherNextEvents* stores the events that may be added to *syncEvents*. If a possible synchronisation is found, line 13, we check the valid structure of the other tuple for events that have the possibility of being always offered to the environment after the synchronisation, lines 15 to 42.

The first time a synchronisation is found, line 18, we update *otherNextEvents* accordingly, lines 19 to 28. If another synchronisation is found later on, line 29, we gather the events that will be available after its occurrence, lines 30 to 39, and check if they are the same to what was previously gathered, lines 40 and 41. If they are not, *validEvent* is marked false, meaning that no guarantee can be given regarding the events offered by the synchronised tuple. Since a synchronisation can happen with any of the tuples that match the given synchronisation pair, if more than one is present in the metadata, they need to offer the same events to be added to *syncEvents*, lines 44 and 45.

Example 19 We consider the following processes.

$$Ex19a = a \rightarrow b \rightarrow Ex19a \qquad Ex19d = Ex19a \;|||\; Ex19b$$
$$Ex19b = b \rightarrow c \rightarrow Ex19b \qquad Ex19e = Ex19d[\![\{a, b\}]\!]Ex19c$$
$$Ex19c = (a \rightarrow b \rightarrow c \rightarrow Ex19c) \,\Box\, (d \rightarrow SKIP)$$

Algorithm 9 syncEvents(ev,sycnSet,M)

```
1: syncEvents = ∅
2: for each syncTuple ∈ syncSet do
3:    if ev ∈ second(syncTuple) then
4:       for each tuple ∈ last(M) do
5:          localTuples = {tuple}
6:          otherNextEvents = ∅
7:          validEvent = true
8:          while localTuples ≠ ∅ do
9:             otherElem = getFromSet(localTuples)
10:            localSeq = first(otherElem)
11:            pointers = third(otherElem)
12:            for each otherSyncTuple ∈ second(otherElem) do
13:               if first(syncTuple) == first(otherSyncTuple)*-1 then
14:                  nextAdded = false
15:                  while localSeq ≠ ⟨⟩ do
16:                     for each otherSeq ∈ last(localSeq) do
17:                        for each otherEv ∈ front(otherSeq) do
18:                           if ev == otherEv ∧ ¬nextAdded then
19:                              otherEnd = last(otherSeq)
20:                              if getNextFromSeq(otherEv,otherSeq) == otherEnd then
21:                                 if otherEnd ∈ ℕ then
22:                                    otherNextEvents =
                                      otherNextEvents ∪ getHeads(getFromSeq(otherEnd,localSeq))
23:                                 else if otherEnd == SKIP then
24:                                    for each pointer ∈ pointers do
25:                                       otherNextEvents =
                                         otherNextEvents ∪ getHeads(getFromSeq(pointer,M))
26:                              else
27:                                 otherNextEvents =
                                    otherNextEvents ∪ {getNextFromSeq(otherEv,otherSeq)}
28:                              nextAdded = true
29:                           else if ev == otherEv ∧ validEvent then
30:                              otherEnd = last(otherSeq)
31:                              otherNextEventsTemp = ∅
32:                              if getNextFromSeq(otherEv,otherSeq) == otherEnd then
33:                                 if otherEnd ∈ ℕ then
34:                                    otherNextEventsTemp =
                                      otherNextEventsTemp ∪ getHeads(getFromSeq(otherEnd,localSeq))
35:                                 else if otherEnd == SKIP then
36:                                    for each pointer ∈ pointers do
37:                                       otherNextEventsTemp =
                                         otherNextEventsTemp ∪ getHeads(getFromSeq(pointer,M))
38:                              else
39:                                 otherNextEventsTemp =
                                    otherNextEventsTemp ∪ {getFromSeq(otherEv,otherSeq)}
40:                              if otherNextEvents ≠ otherNextEventsTemp then
41:                                 validEvent = false
42:                     localSeq = front(localSeq)
43:               localTuples = localTuples \ {otherElem} ∪ getChildren(pointers,M)
44:            if otherNextEvents ≠ ∅ ∧ validEvent then
45:               syncEvents = syncEvents ∪ otherNextEvents
46: return syncEvents
```

$$\mathrm{M}(Ex19e) = \left\langle \begin{Bmatrix} \left(\left\langle \left\{ \left\langle a, b, 0 \right\rangle \right\} \right\rangle, \left\{ \left(1, \left\{ a, b \right\} \right) \right\}, \varnothing \right), \\ \left(\left\langle \left\{ \left\langle a, b, 0 \right\rangle \right\} \right\rangle, \left\{ \left(1, \left\{ a, b \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \begin{Bmatrix} \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \left\{ \left(1, \left\{ a, b \right\} \right) \right\}, \left\{ 1 \right\} \right), \\ \left(\left\langle \begin{Bmatrix} \left\langle a, b, c, 0 \right\rangle \\ \left\{ \left\langle d, SKIP \right\rangle \right\} \end{Bmatrix} \right\rangle, \left\{ \left(-1, \left\{ a, b \right\} \right) \right\}, \varnothing \right) \end{Bmatrix}, \\ \left\{ \left(\left\langle \left\{ \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 2 \right\} \right) \right\} \end{Bmatrix} \right\rangle$$

If we execute $syncEvents(a, \{(1, \{a, b\})\}, Ex19e)$, the result is {b}, since there is only one tuple with event a that synchronises on -1. If, on the other hand, we execute $syncEvents(b, \{(-1, \{a, b\})\}, Ex19e)$, the result is $\varnothing$, since, of the two tuples that have event b and synchronise on 1, one offers event a, and the other c. For the two possible executions of $syncEvents(b, \{(1, \{a, b\})\}, Ex19e)$, the function can differentiate between the valid sequences of $Ex19a$ and $Ex19b$, which come from the call to *nextEvents* in *getLocalStates*. □

With the local states calculated, we are now in a position to check for the possible presence of nondeterminism; the current stage of the verification can be seen in Figure 13. There are two scenarios where nondeterminism can arise from a parallel composition, with both having an event in common as the trigger.

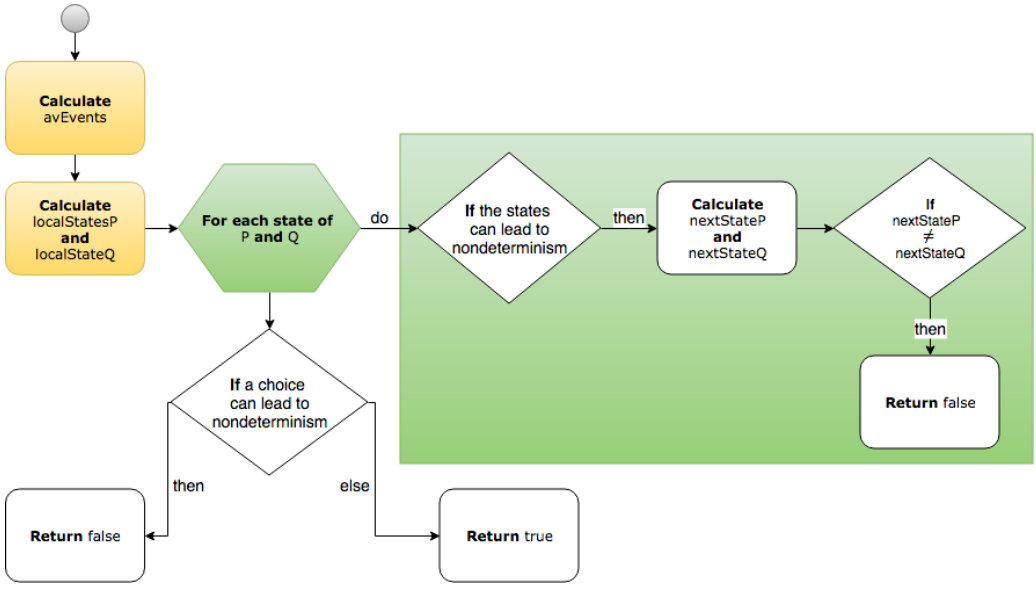

Figure 13 – Checking the local states in Parallelism(P,Q,X)

First, the algorithm checks for each possible pair of local states of P and Q, if they offer the same event to the environment, and if this event is not in the synchronisation set, lines 4 and 5. If this is the case, then the events offered to the environment if either

process evolves must be the same, for the composition to be deterministic, lines 6 to 9. We calculate the events that will be offered by joining the events gathered by *getLocalStates*, the second element of the local state, the event of the other process, that continues to be offered, and the events that are always available, *avEvents*.

At a given moment, a process can be offering a number of events to the environment, and not only the event that we capture in the first element of *stateP* and *stateQ*. These events, however, can be offered in some situations, and not in others, depending on the overall state of the CSP process. To avoid the state explosion problem, we only consider the events that are guaranteed to happen. One effect of this design choice is that there are some deterministic specifications that the algorithm cannot guarantee to be deterministic, since it does have all the information available.

Example 20 We consider the following processes.

$$Ex20a = a \rightarrow b \rightarrow a \rightarrow Ex20a \qquad Ex20b = Ex20a \;|||\; Ex20a$$

For the process $Ex20b$, we have $avEvents = \varnothing$. The conditional in line 5 of Algorithm 3 returns true for the first events of both operands of $Ex20b$, with *nextStateP* and *nextStateQ* being both the result of $\{b\} \cup \{a\} \cup \varnothing$, so $nextStateP == nextStateQ$, line 8. For the first event of the first operand, a, and the second event of the second operand, b, the conditional in line 5 returns false. For the first event of the first operand, a, and the third event of the second operand, a, line 5 returns true, but $nextStateP = \{b\} \cup \{a\} \cup \varnothing$, and $nextStateQ = \{a\} \cup \{a\} \cup \varnothing$, so Algorithm 3 returns false in line 9. □

Example 21 We consider the following processes.

$$Ex21a = a \rightarrow b \rightarrow c \rightarrow Ex21a \qquad Ex21d = Ex21a \;|||\; Ex21b$$
$$Ex21b = d \rightarrow e \rightarrow f \rightarrow Ex21b \qquad Ex21e = Ex21d [\![\{d\}]\!] Ex21c$$
$$Ex21c = d \rightarrow e \rightarrow g \rightarrow Ex21c$$

If we check $Ex21e$, we have $avEvents = \varnothing$, since there is no sequence with a freely occurring event. Algorithm 3 then calculates the local states and check for equal events. For the states offering events a, b, c, and f, in $Ex21d$, and the state offering event g, in $Ex21c$, the conditional in line 5 returns false. When *stateP* and *stateQ* both offer d, the conditional in line 5 also returns false, since event d is in the synchronisation set. For event e, however, we have that $nextStateP = \{g\} \cup \{e\} \cup \varnothing$ and $nextStateQ = \{f\} \cup \{e\} \cup \varnothing$, so Algorithm 3 returns false, in line 9. □

The second possibility of nondeterminism arises if one of the operands has an external choice, and one of the initial events of this choice is also present in the other operand; this is the final verification stage, as shown in Figure 14. In this case it is possible that the environment looses control over how the choice is resolved after the composition. An example of this scenario is shown in Example 22.

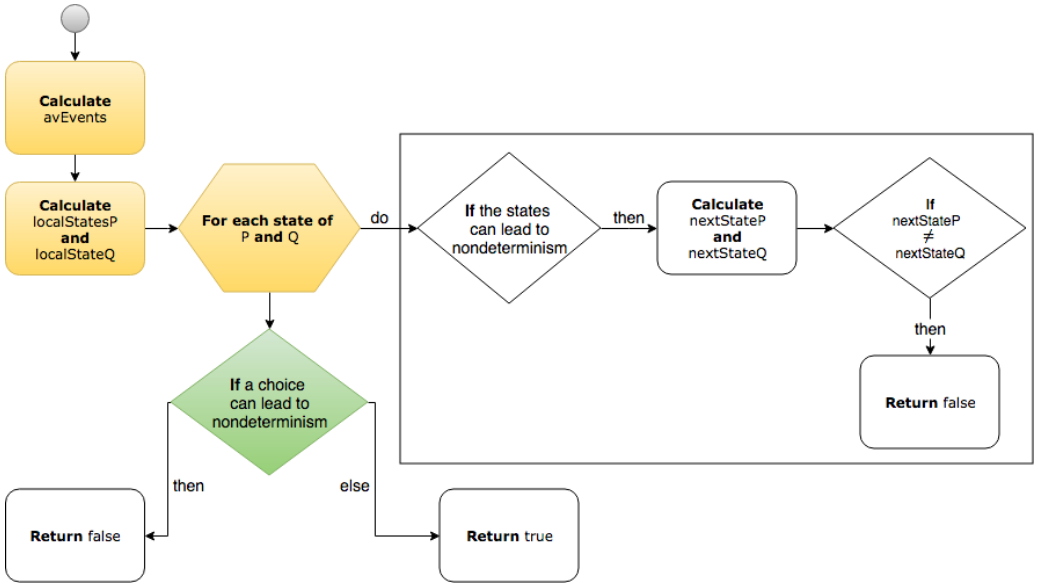

Figure 14 – Checking the external choices in Parallelism(P,Q,X)

Example 22 We consider the following processes.

$$Ex22a = Ex21a \,\Box\, Ex21b \qquad\qquad Ex22b = Ex22a \,|||\, Ex21a$$

The process $Ex22a$ is deterministic, because the initials of each operand are disjoint. The composition in $Ex22b$, however, have a nondeterminism arising from event a. Even though every pair of states from P and Q do not cause a problem in themselves, the choice in $Ex22a$ causes nondeterminism. If the first event that the environment synchronises with in $Ex21b$ is a, we cannot know if events a and b will be offered afterwards, if $Ex21a$ in the left operand evolves, or if events a, b, and d will be available, if $Ex21a$ in the right operand evolves. □

To check if a choice can lead to nondeterminism in a parallel composition, we use *extChoiceWithND*, shown in Algorithm 10. This function takes the metadata of two processes and checks if a choice in the first one can lead to nondeterminism. For the analysis, *extChoiceWithND* checks in the tree of P, lines 2 to 11, if an external choice in Basic Process of P, lines 6 to 8, or Composite Process of P, lines 9 and 10, lead to nondeterminism; the functions *basicExtChoiceWithND* and *compExtChoiceWithND* checks those cases, respectively.

We apply *extChoiceWithND* to both P and Q in Algorithm 3, line 10, and if a choice present in either operand has the possibility of introducing nondeterminism, the algorithm returns false in line 11.

The function *basicExtChoiceWithND*, shown in Algorithm 11, checks, for every event in Q, lines 1 to 11, if this event occurs in the initials of the external choice modelled in *setP*, if it is not in the synchronisation set, and if the branches of the external choice

Algorithm 10 extChoiceWithND(P,Q,X)

```
1: tuplesP = last(P)
2: while tuplesP ≠ ∅ do
3:    elemP = getFromSet(tuplesP)
4:    localSeqP = first(elemP)
5:    pointersP = third(elemP)
6:    for each setP ∈ localSeqP do
7:       if size(setP) > 1 ∧ ¬basicExtChoiceWithND(setP,elemP,Q,X) then
8:          return false
9:    if size(pointersP) > 1 ∧ ¬compExtChoiceWithND(pointersP,P,Q,X) then
10:      return false
11:   tuplesP = tuplesP \ {elemP} ∪ getChildren(pointersP,P)
12: return true
```

are not equivalent. The function *pathEq* checks if the events starting in each sequence in $setP$, in $elemP$, lead to the equivalent states; we only need to check the valid structure here, and not the the other elements of P, because, if equal sequences of events end in $SKIP$, they will follow the same pointers.

Algorithm 11 basicExtChoiceWithND(setP,elemP,Q,X)

```
1: tuplesQ = last(Q)
2: while tuplesQ ≠ ∅ do
3:    elemQ = getFromSet(tuplesQ)
4:    localSeqQ = first(elemQ)
5:    pointersQ = third(elemQ)
6:    for each setQ ∈ localSeqQ do
7:       for each seqQ ∈ setQ do
8:          for each evQ ∈ front(seqQ) do
9:             if evQ ∈ getHeads(setP) ∧ evQ ∉ X ∧ ¬pathEq(setP,elemP) then
10:               return false
11:   tuplesQ = tuplesQ \ {elemQ} ∪ getChildren(pointersQ,Q)
12: return true
```

For external choices that arise from compositions, we use *algCompExtChoiceWithND*, shown in Algorithm 12. This function works similarly to its counterpart for external choice in Basic Processes, checking, for all events in Q, if they interfere with an initial event of the external choice captured by the set of pointers. The function *pathEq* is overloaded to also analyse the metadata itself; in this case it needs to go down the tree to ensure that both branches of the choice are equivalent.

In all points where two events are compared in the verification of parallelism, be it directly or by checking if it is part of a set of events, we use the function *EqParallelism*. The body of this function is equal to that of *EqExtChoiceStart*, since, in both cases, the

Algorithm 12 compExtChoiceWithND(pointersP,P,Q,X)

```
1: tuplesQ = last(Q)
2: while tuplesQ ≠ ∅ do
3:    elemQ = getFromSet(tuplesQ)
4:    localSeqQ = first(elemQ)
5:    pointersQ = third(elemQ)
6:    for each setQ ∈ localSeqQ do
7:       for each seqQ ∈ setQ do
8:          for each evQ ∈ front(seqQ) do
9:             if evQ ∈ getHeads(getFromSeq(pointersP,P)) ∧ evQ ∉ X ∧
               ¬pathEq(pointersP,P) then
10:               return false
11:   tuplesQ = tuplesQ \ {elemQ} ∪ getChildren(pointersQ,Q)
12: return true
```

condition that can lead to nondeterminism is that the events are equal.

- EqParallelism(p1,p2) = if $p1 \in Val \wedge p2 \in Val$ then
  if $p1 == p2$ then *true* else *false*
  else
  true

With the functional aspect of our verification of parallel compositions explained, let us consider the efficiency of our approach. The analysis of determinism is naturally exponential, since we need to check all possible pairs of states of both operands. By doing so we achieve a sound and complete analysis, but a very inefficient one. Our strategy takes into account the subset that we are dealing with and all the potential sources of nondeterminism that can arise from it to limit the scope of the analysis.

As can be seen in line 4 of Algorithm 3, we do check all pairs of events from both operands. This, however, is a subset of all the possible states of the processes, since, when checking a valid sequence, we do not take into account the state of the other Enhanced Traces in the metadata. For process $Ex21e$, for example, we do not consider all possible states of its left operand, $Ex21d$, which would be 9 states, because, when checking an event in $Ex21a$, the event that is being offered in $Ex21b$ is not taken into account, so we only consider 6 states of the 9 available. This trivial example does not show it, but this approach is essential to the scalability of our strategy.

If we define a process composed of 10 copies of $Ex21a$ in interleaving, we will have 3^{10}, more than fifty thousand, possible states. Following our constructive approach, we check each composition individually. In the first one, the strategy analyses all the nine states available. On the following compositions, however, we analyse only part of the available states. Each copy of $Ex21a$ has three local states, so for $Comp1 \;|||\; Ex21$, with

$Comp1 = Ex21a \;|||\; Ex21a$, we have (3+3)*3 states. By adding the number of states checked during each composition, we reach a total of 405.

All Basic Processes that offer more than one event can, at a given state, offer a specific event, but, more importantly, can refuse to offer it; this is the idea that underpins our whole verification of parallelism. To ensure that we do not discard relevant information, we use *avEvents* to capture the events that are available to the environment in all states.

3.3.4 Hiding

In the failures model, hiding of the initials of processes in external choice can introduce nondeterminism. The reason is that hiding does not remove the events from the processes, but makes them invisible, so, if initial events are hidden, the environment may loose the way of controlling a choice, which becomes internal.

Example 23 We consider the following processes.

$$Ex23a = a \rightarrow b \rightarrow c \rightarrow Ex23a \qquad Ex23c = Ex23a \;\Box\; Ex23b$$
$$Ex23b = b \rightarrow c \rightarrow d \rightarrow Ex23b$$

Hiding event c in $Ex23c$, which is deterministic, does not introduce nondeterminism, because this does not affect its initials. Hiding event a, however, allows the choice to be resolved in favour of $Ex23a$, due to the possibility of engaging in the hidden event, without the command of the environment. □

Algorithm 13 Hiding *(P,X)*

```
1:  tuplesP = last(P)
2:  while tuplesP ≠ ∅ do
3:     elemP = getFromSet(tuplesP)
4:     localSeqP = first(elemP)
5:     pointersP = third(elemP)
6:     for each setP ∈ localSeqP do
7:        if size(setP) > 1 ∧ getHeads(setP) ∩ X ≠ ∅ then
8:           newElem = hideEv(elemP,X)
9:           newSet = hideEv(setP,X)
10:          if ¬pathEq(newSet,newElem) then
11:             return false
12:    if size(pointersP) > 1 ∧ getHeads(pointersP,P) ∩ X ≠ ∅ then
13:       newP = hideEv(P,X)
14:       if ¬pathEq(pointersP,newP) then
15:          return false
16:    tuplesP = tuplesP \ {elemP} ∪ getChildren(pointersP,P)
17: return true
```

When hiding is applied to a process P we check if an initial event of an external choice is being hidden. If it is, then, to be deterministic, all branches of the choice, with the required events hidden, need to equivalent. We use Algorithm 13; P is a metadata, and X the set of events to be hidden. This function checks the entire metadata, lines 1 to 16, for the presence of external choices, both in Basic Processes, lines 6 to 11, and Composite Processes, lines 12 to 15, similarly to *extChoiceWithND*.

If an external choice is found, we check, using *hideEv*, if, after removing the events in X, the branches of the metadata are equivalent, lines 10 and 14; the function *hideEv* is overloaded, and simply returns the given structure without the events in its second argument, the set X.

Hiding events from a deterministic internal choice or the parallel composition does not introduce nondeterminism. In the case of internal choice, the reason is that its operands need to be equivalent for it to be deterministic, and, for parallelism, all events in the intersection of the alphabets lead to the same set of events being offered to the environment.

We have implemented all the algorithms presented in this section to construct a prototype determinism checker. In the next chapter, we show the results of experiments carried out using this prototype.

4 EXPERIMENTAL RESULTS

To automate the application of our strategy, and to check its efficiency, we developed a prototype and performed a number of case studies. We discuss the features of our prototype in Section 4.1. The case studies performed are presented in Section 4.2, with their results being detailed in Section 4.3, and the threats to validity being discussed in Section 4.4. All the files used in the experiments, together with the prototype itself, are available online[1].

4.1 Prototype

The prototype is implemented in Java (GOSLING et al., 2015), and is built upon the CSP parser developed by Jesus Júnior (2009). It receives a CSP specification complying with its requirements and outputs a message with its conclusion. The message can be "All processes are deterministic", or it can point out a potential nondeterministic composition in the specification.

The current version of the prototype implements a large subset of our strategy, containing all its algorithms. The elements that were not yet implemented are some details related to the access of some structure of the metadata. One example of a process that the prototype is not yet able to analyse is shown below.

$$P = a \rightarrow P \qquad PQ = P \,\Box\, Q$$
$$Q = b \rightarrow Q \qquad PQR = PQ \,\Box\, R$$
$$R = c \rightarrow R \qquad PQRS = PQR \,\Box\, S$$
$$S = a \rightarrow b \rightarrow S$$

$$\mathrm{M}(PQRS) = \left\langle \begin{array}{l} \left\{ \left(\left\langle \left\{ \left\langle \left\langle a, 0 \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle b, 0 \right\rangle \right\} \right\rangle, \varnothing, \varnothing, \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 1, 2 \right\} \right) \right\} \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle c, 0 \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 3, 4 \right\} \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle a, b, 0 \right\rangle \right\} \right\rangle, \varnothing, \varnothing \right) \right\}, \\ \left\{ \left(\left\langle \left\{ \left\langle \left\langle SKIP \right\rangle, \right\} \right\rangle, \varnothing, \left\{ 5, 6 \right\} \right) \right\} \end{array} \right\rangle$$

[1] https://www.cin.ufpe.br/~rbo2/DissertationFiles.zip

In M($PQRS$) we have a tree with two composite nodes from the root to the tuples that model the processes P and Q. Our prototype, currently, can only check up to one composite node between two tuples, so the process $PQRS$ cannot be checked. This is only an implementation issue, since the strategy does not have this limitation.

Besides restrictions on the analysis, we also have some limitations regarding the writing of the specifications. Synchronisation channels, for example, need to be written explicitly, due to how they are treated by the implementation. For a channel $in : \{0, 1\}$, we need to explicitly write the set of events $\{in.0, in.1\}$, instead of the usual shorthand $\{|\ in\ |\}$.

One final comment must be made regarding the compositions. In all examples so far we always performed one composition at a time. The reason behind this is that our prototype can only check compositions written in such a manner. The strategy itself does not forbid us from writing $PQRS = ((P \ \Box\ Q) \ \Box\ R) \ \Box\ S$, but since the current implementation of the prototype does not calculate the metadata of the intermediary components $(P \ \Box\ Q)$ and $(P \ \Box\ Q) \ \Box\ R)$, it cannot handle this style of writing. This restriction does not diminishes the expressiveness of the specifications that the prototype can analyse, and it can be addressed in future implementations.

The limitations of our prototype require only implementation effort to be resolved. They do not reflect any theoretic limitations of our strategy, beyond those discussed in Chapter 3. They do not impact our experiments, since the unimplemented aspects of the strategy would not be executed even if they were present in the prototype.

4.2 Case Studies

We performed seven case studies: two with systems extracted from the literature, and five with toy examples. The toy examples are used as a benchmark to analyse each algorithm individually, and the more complex systems are used to check their combined performance.

4.2.1 Systems from the Literature

One of our case studies is a Ring Buffer described by Woodcock and Cavalcanti (2001). This system is a fixed sized buffer with a controller and a number of cells, each one capable of storing one piece of information, organized in a circular fashion. The controller stores the index of the first and last cells with information, in a FIFO-style, and contains itself a cache, so a buffer with four cells is capable of storing up to five pieces of information.

After each input or output operation, the controller updates its indexes, if needed, and the value of its cache. If the buffer is empty its output operation is disabled, as is its input option, if it is full. An example of a ring buffer with eight cells can be seen in Figure 15. In this example the buffer contains two values, 2 in the cache, and 5 referenced by the *bot* index; the *top* index points to the next cell to be written. The values in the first three cells were already read, so their state is irrelevant.

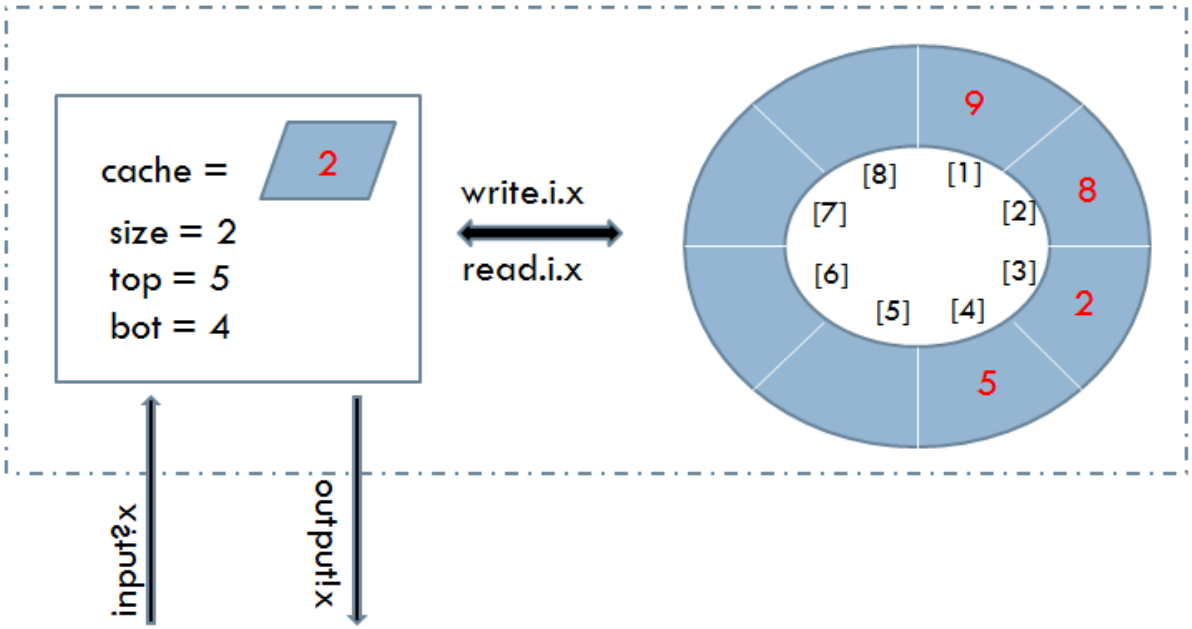

Figure 15 – Graphical representation of a ring buffer.

The CSP specification of a cell is shown below. The current state of the cell is modelled by *IRingCellState*, with internal operations to read, *i_rd*, and write, *i_wrt*. These operations are controlled by *IRingCellBody*, that initially behaves as *IRingCellInit*, since output is not enabled, and then it behaves as *IRingCellLoop*. The cell itself, *IRingCell*, is the result of the parallel composition of its state and body, *IRingCellComp*, with the internal communications on *i_rd* and *i_wrt* hidden. So the visible channels of a cell are *read* and *write*, as shown in Figure 15.

$$IRingCellState(id, val) = \begin{array}{l} i\_rd.id!val \rightarrow IRingCellState(id, val) \\ \Box \\ i\_wrt.id?newVal \rightarrow IRingCellState(id, newVal) \end{array}$$

$$IRingCellInit(id) = write.id?newVal \rightarrow i\_wrt.id!newVal \rightarrow SKIP$$

$$IRingCellLoop(id) = i\_rd.id?val \rightarrow \left(\begin{array}{l} read.id!val \rightarrow IRingCellLoop(id) \\ \Box \\ write.id?newVal \rightarrow i\_wrt.id!newVal \rightarrow IRingCellLoop(id) \end{array} \right)$$

$$IRingCellBody(id) = IRingCellInit(id) \,;\, IRingCellLoop(id)$$

$$IRingCellComp(id) = IRingCellBody(id) [\![\{\!|\, i\_rd, i\_wrt \,|\!\}]\!] IRingCellState(id, 0)$$

$$IRingCell(id) = IRingCellComp(id) \setminus \{\!|\, i\_rd, i\_wrt \,|\!\}$$

The specification of the controller is similar in structure, and is shown below. The process *Controller* is the result of the parallel composition of *ControllerLoop*, which manages the communication with the environment, through the channels *input* and *output*, and *ControllerState*, which manages the cache and the cells, with the internal communications on *readState*, *readVal*, and *writeVal* hidden.

$$
\begin{aligned}
&ControllerState(top, botton, cache, size) = \\
&\quad readState!size!cache \rightarrow ControllerState(top, botton, cache, size) \\
&\quad \Box \\
&\quad readVal \rightarrow \left(\begin{array}{l} size > 1 \ \ \& \ read.botton?val \rightarrow \\ ControllerState(top, (botton + 1)\%MaxRing, val, size - 1) \\ \Box \\ size == 1 \ \ \& \ ControllerState(top, botton, cache, 0) \end{array} \right) \\
&\quad \Box \\
&\quad writeVal?val \rightarrow \left(\begin{array}{l} size == 0 \ \ \& \ ControllerState(top, botton, val, 1) \\ \Box \\ size > 0 \ and \ size < MaxBuff \ \ \& \ write.top!val \rightarrow \\ ControllerState((top + 1)\%MaxRing, botton, cache, size + 1) \end{array} \right) \\
&ControllerLoop = readState?size?cache \rightarrow \\
&\left(\begin{array}{l} size < MaxBuff \ \ \& \ input?val \rightarrow writeVal!val \rightarrow ControllerLoop \\ \Box \\ size > 0 \ \ \& \ output!cache \rightarrow readVal \rightarrow ControllerLoop \end{array} \right) \\
&ControllerComp = \ ControllerLoop \\
&\qquad\qquad\qquad [\![\{\!| \ readState, readVal, writeVal \ |\!\}]\!] \\
&\qquad\qquad\qquad ControllerState(0, 0, 0, 0) \\
&Controller = ControllerComp \setminus \{\!| \ readState, readVal, writeVal \ |\!\}
\end{aligned}
$$

The buffer itself is simply the parallel composition of the controller with a given number of cells. We can see below the specification of ring buffer that contains four cells.

$$
\begin{aligned}
&CellComp1 = IRingCell(0) \ ||| \ IRingCell(1) \\
&CellComp2 = CellComp1 \ ||| \ IRingCell(2) \\
&CellComp3 = CellComp2 \ ||| \ IRingCell(3) \\
&RingComp = Controller [\![\{\!| \ read, write \ |\!\}]\!] CellComp3 \\
&RingBuffer = RingComp \setminus \{\!| \ read, write \ |\!\}
\end{aligned}
$$

For our experiments, we created a nondeterministic version of the ring buffer presented, in which one of the cells can have an error, after which it reads or writes nondeterministically. The *IRingCellLoop* process of the modified cell is shown below; the other processes of the specification remain unchanged.

$$IRingCellLoopND(id) = i\_rd.id?val \rightarrow \left(\begin{array}{l} read.id!val \rightarrow IRingCellLoopND(id) \\ \Box \\ write.id?newVal \rightarrow i\_wrt.id!newVal \rightarrow IRingCellLoopND(id) \\ \Box \\ error \rightarrow read.id!val \rightarrow IRingCellLoopND(id) \\ \Box \\ error \rightarrow write.id?newVal \rightarrow i\_wrt.id!newVal \rightarrow IRingCellLoopND(id) \end{array} \right)$$

Besides the ring buffer, we also used the railway network described in Section 2.1.1 as a case study. In addition to the original, deterministic, version, we created nondeterministic instances for the experiments. These instances have a problem in their last two pairs of tracks, which leads to nondeterminism. The processes $Pair18$ and $Pair19$ of a nondeterministic network with twenty pairs are shown below.

$$Pair18 = \begin{array}{l} signal.18 \rightarrow signal.19 \rightarrow signal.0 \rightarrow interference \rightarrow FixingProblem \rightarrow \\ delay.5 \rightarrow ProblemFixed \rightarrow Pair18 \end{array}$$

$$Pair19 = \begin{array}{l} signal.1 \rightarrow signal.19 \rightarrow signal.0 \rightarrow interference \rightarrow FixingProblem \rightarrow \\ delay.8 \rightarrow ProblemFixed \rightarrow Pair19 \end{array}$$

4.2.2 Toy Examples

To check the efficiency of each algorithm individually, we created five toy examples. These examples consist of a number of simple Basic Processes composed with a particular operator. Below we have the instance of size five for the external choice example.

$$Basic0 = a.0 \rightarrow b.0 \rightarrow c.0 \rightarrow Basic0 \qquad Comp0 = Basic0 \,\Box\, Basic1$$
$$Basic1 = a.1 \rightarrow b.1 \rightarrow c.1 \rightarrow Basic1 \qquad Comp1 = Comp0 \,\Box\, Basic2$$
$$Basic2 = a.2 \rightarrow b.2 \rightarrow c.2 \rightarrow Basic2 \qquad Comp2 = Comp1 \,\Box\, Basic3$$
$$Basic3 = a.3 \rightarrow b.3 \rightarrow c.3 \rightarrow Basic3 \qquad Comp3 = Comp2 \,\Box\, Basic4$$
$$Basic4 = a.4 \rightarrow b.4 \rightarrow c.4 \rightarrow Basic4$$

For internal choice and interleaving, we use a similar structure, only changing the operator in the *Comp* processes. One of our toy examples is a mix of operators, with the

odd compositions using interleaving, and the even ones using external choice. The aim of this example is to check parallelism with operands that have an external choice.

To analyse the hiding algorithm, we use a series of external choices, which then have one of their events hidden. The instance of size five of the hiding example is shown below. The processes $Basic0$ to $Basic4$ are as in the previous example.

$$Comp0 = Basic0 \,\Box\, Basic1 \qquad Hiding0 = Comp0 \setminus \{b.1\}$$
$$Comp1 = Hiding0 \,\Box\, Basic2 \qquad Hiding1 = Comp1 \setminus \{b.2\}$$
$$Comp2 = Hiding1 \,\Box\, Basic3 \qquad Hiding2 = Comp2 \setminus \{b.3\}$$
$$Comp3 = Hiding2 \,\Box\, Basic4 \qquad Hiding3 = Comp3 \setminus \{b.4\}$$

For the nondeterministic versions of the examples, we made small modifications to certain compositions. The modifications made depend on the operator being changed. The nondeterministic version of the hiding instance shown above, for example, has the process $Hiding3 = Comp3 \setminus \{a.4\}$, while the external choice instance shown before it has the body of the process $Basic4$ being changed to $a.3 \rightarrow b.4 \rightarrow c.4 \rightarrow Basic4$. In all cases, nondeterminism was introduced in the last compositions of the specification, which is the most unfavourable position for the prototype, since it requires the verification of all previous processes.

4.3 Results

The experiments were run in a server with an Intel Core i7-2600k, 16GB of RAM, 160GB of SSD, and Ubunto 17.04 64-bit. We used FDR 4.2.3. All the experiments were run three times, and we present here the best results for FDR and the worst results for our prototype; the * indicates an out-of-memory error.

In all experiments our prototype correctly identified the deterministic specifications as such, and returned an inconclusive result for the nondeterministic ones. It is important to remember that FDR implements a complete strategy, and it provides counter-examples when nondeterminism is found, something that our prototype is unable to do.

The results for the ring buffer experiment can be seen in tables 1 and 2; the instance size indicates the number of memory cells. As we can see, FDR can only analyse the most trivial instances, while our prototype can analyse almost all of them. For the nondeterministic instances, in particular, the prototype can quickly give a result, although an incomplete one. The experiments show that the ring buffer is a especially hard system to analyse through a global approach.

For the railway network, we considered three scenarios: one train in the network, three trains in the network, and five trains in the network. The increase in the number of trains leads to considerable larger state spaces, so we can analyse how each tool scales. The

Table 1 – Deterministic instances of the ring buffer experiment.

Instance	FDR4	Prototype
3	0.48s	0.75s
6	*	0.50s
9	*	0.55s
10	*	0.55s
30	*	2.00s
60	*	3.29s
90	*	6.14s
100	*	7.99s
300	*	1m 11s
500	*	5m 21s
700	*	11m 29s
900	*	31m 47s
1000	*	*

Table 2 – Nondeterministic instances of the ring buffer experiment.

Instance	FDR4	Prototype
3	0.21s	0.68s
6	*	0.46s
9	*	0.47s
10	*	0.53s
30	*	0.98s
60	*	2.44s
90	*	3.24s
100	*	4.51s
300	*	19.15s
500	*	46.02s
700	*	1m 30s
900	*	2m 23s
1000	*	2m 58s

results for the network with one train can be seen in tables 3 and 4; the instance number indicates the size of the network, that is, the amount of pairs of tracks.

Table 3 – Deterministic instances with one train in the railway.

Instance	FDR4	Prototype
20	0.12s	0.60s
30	0.14s	0.87s
60	0.26s	5.14s
90	0.40s	20.46s
100	0.42s	31.97s
300	1.44s	47m 33s
500	2.82s	*
700	4.49s	*
900	6.82s	*
1000	8.38s	*

Table 4 – Nondeterministic instances with one train in the railway.

Instance	FDR4	Prototype
20	0.13s	0.65s
30	0.15s	1.04s
60	0.27s	5.02s
90	0.40s	20.81s
100	0.43s	31.12s
300	1.70s	43m 59s
500	3.38s	*
700	6.14s	*
900	9.84s	*
1000	11.84s	*

In this example FDR gives the best results, since it is not only capable of analysing all instances, but it is also more efficient in doing so. Our prototype suffers with the need to calculate and maintain its metadata, which hinders its efficiency. The scalability of our strategy, however, shows up in the other two scenarios, as can be seen in tables 5 to 8.

With a larger state space to analyse, FDR loses its edge. In the network with three trains it is still a better alternative than our prototype, but it starts to struggle with

Table 5 – Deterministic instances with three trains in the railway.

Instance	FDR4	Prototype
20	0.17s	0.86s
30	0.26s	0.89s
60	0.87s	4.95s
90	2.36s	20.24s
100	3.29s	32.60s
300	4m 33s	45m 33s
500	32m 57s	*
700	*	*
900	*	*
1000	*	*

Table 6 – Nondeterministic instances with three trains in the railway.

Instance	FDR4	Prototype
20	0.12s	0.59s
30	0.16s	1.01s
60	0.35s	5.29s
90	0.72s	20.79s
100	0.91s	31.78s
300	44.33s	46m 55s
500	5m 46s	*
700	25m 12s	*
900	1h 9m	*
1000	*	*

Table 7 – Deterministic instances with five trains in the railway.

Instance	FDR4	Prototype
20	0.21s	0.59s
30	0.77s	0.87s
60	1m 28s	5.58s
90	20m 39s	20.91s
100	*	31.72s
300	*	47m 56s
500	*	*
700	*	*
900	*	*
1000	*	*

Table 8 – Nondeterministic instances with five trains in the railway.

Instance	FDR4	Prototype
20	0.13s	0.83s
30	0.19s	0.88s
60	1.54s	4.90s
90	17.38s	21.05s
100	35.52s	30.80s
300	*	45m 46s
500	*	*
700	*	*
900	*	*
1000	*	*

the larger instances. In the network with five trains it becomes preferable to apply our strategy, since it is more efficient and is capable of analysing instances that FDR cannot.

In all three scenarios our prototype shows a somewhat similar result, this happens because an increase in the size of the state space does not automatically translates into a greater verification effort. Our strategy does not need to analyse the new states that arise from the positions of the additional trains to achieve its result.

It is important to point out that the bad result that our prototype achieved when analysing the network with only one train comes from the overhead created by the metadata. An earlier version of our strategy (OTONI; CAVALCANTI; SAMPAIO, 2017), in which we considered a more restricted subset of CSP, and thus had a simpler metadata structure, had more positive results in this scenario.

With the toy examples we can get a clear picture of the efficiency of each individual algorithm. When analysing external choice compositions, FDR was more efficient, checking all instances in less than one second. Our prototype, although less efficient, was also able to analyse all instances in a reasonable amount of time. The results for the experiments with external choice can be seen in tables 9 and 10.

Table 9 – Deterministic instances of the external choice experiment.

Instance	FDR4	Prototype
3	0.06s	0.53s
6	0.06s	0.26s
9	0.06s	0.28s
10	0.06s	0.29s
30	0.07s	0.36s
60	0.08s	0.51s
90	0.09s	0.57s
100	0.09s	0.56s
300	0.19s	1.23s
500	0.33s	2.81s
700	0.52s	4.89s
900	0.67s	6.27s
1000	0.84s	11.35s

Table 10 – Nondeterministic instances of the external choice experiment.

Instance	FDR4	Prototype
3	0.04s	0.26s
6	0.04s	0.28s
9	0.05s	0.28s
10	0.05s	0.32s
30	0.05s	0.38s
60	0.06s	0.47s
90	0.07s	0.54s
100	0.08s	0.53s
300	0.12s	1.22s
500	0.32s	2.95s
700	0.47s	5.12s
900	0.68s	7.21s
1000	0.76s	9.71s

When it comes to internal choice, both tools again can analyse all given instances. The results can be seen in tables 11 and 12. For the deterministic case, the scalability of the prototype starts to show in the instance of size 100, due to increase of the state space. In the nondeterministic case, however, FDR seems to be able to quickly capture the nondeterminism, despite the size of the state space.

For interleaving the limitations of global analysis show up clearly, as can be seen in tables 13 and 14. FDR is only capable of analysing the trivial instances, while our prototype can check all instances in less than one minute.

In the experiment that combines interleaving and external choice we can see that the results, shown in tables 15 and 16, are somewhat similar to the experiment with only interleaving. FDR still struggles greatly, but the fact that half of the compositions are not made with interleaving alleviates the verification effort a bit, specially in the nondeterministic case. Our prototype is also impacted positively by the reduced number of interleaves, analysing all instances in less than thirty seconds.

Hiding is another operator that FDR has difficulties in dealing with, as can be seen in tables 17 and 18, only being able to check the trivial instances. Our prototype, on the other hand, is able to efficiently analyse all instances.

Table 11 – Deterministic instances of the internal choice experiment.

Instance	FDR4	Prototype
3	0.06s	0.25s
6	0.07s	0.25s
9	0.07s	0.27s
10	0.08s	0.28s
30	0.14s	0.36s
60	0.24s	0.41s
90	0.33s	0.49s
100	0.40s	0.49s
300	1.46s	0.77s
500	3.09s	1.07s
700	5.27s	1.25s
900	14.39s	1.38s
1000	25.56s	1.47s

Table 12 – Nondeterministic instances of the internal choice experiment.

Instance	FDR4	Prototype
3	0.05s	0.24s
6	0.05s	0.27s
9	0.05s	0.27s
10	0.05s	0.27s
30	0.05s	0.32s
60	0.06s	0.43s
90	0.07s	0.45s
100	0.07s	0.52s
300	0.14s	0.76s
500	0.21s	1.03s
700	0.28s	1.23s
900	0.38s	1.46s
1000	0.43s	1.43s

Table 13 – Deterministic instances of the interleaving experiment.

Instance	FDR4	Prototype
3	0.06s	0.25s
6	0.08s	0.31s
9	0.24s	0.32s
10	0.54s	0.34s
30	*	0.51s
60	*	0.84s
90	*	1.14s
100	*	1.31s
300	*	4.57s
500	*	12.01s
700	*	21.10s
900	*	33.93s
1000	*	40.76s

Table 14 – Nondeterministic instances of the interleaving experiment.

Instance	FDR4	Prototype
3	0.07s	0.27s
6	0.08s	0.30s
9	0.13s	0.34s
10	0.18s	0.34s
30	*	0.52s
60	*	0.79s
90	*	1.14s
100	*	1.23s
300	*	4.94s
500	*	11.63s
700	*	21.75s
900	*	33.93s
1000	*	41.63s

For most experiments our prototype took a similar time to analyse both the deterministic and nondeterministic instances of a given size. The reason behind this is that the nondeterministic instances were created in way to pose the hardest scenario to the prototype. By adding a nondeterminism in the final composition, we force the prototype to analyse, and then create the metadata for, all the previous processes, while FDR does not necessarily suffers this impact.

Table 15 – Deterministic instances of the interleaving with external choice experiment.

Instance	FDR4	Prototype
3	0.08s	0.26s
6	0.15s	0.29s
9	0.36s	0.33s
10	0.38s	0.32s
30	*	0.47s
60	*	0.69s
90	*	0.93s
100	*	1.00s
300	*	3.15s
500	*	7.97s
700	*	14.12s
900	*	21.42s
1000	*	25.10s

Table 16 – Nondeterministic instances of the interleaving with external choice experiment.

Instance	FDR4	Prototype
3	0.07s	0.28s
6	0.13s	0.31s
9	0.34s	0.30s
10	0.35s	0.32s
30	4.86s	0.49s
60	32.57s	0.65s
90	1m 41s	0.94s
100	2m 22s	1.03s
300	*	3.34s
500	*	7.62s
700	*	13.83s
900	*	21.82s
1000	*	25.30s

Table 17 – Deterministic instances of the hiding experiment.

Instance	FDR4	Prototype
3	0.08s	0.25s
6	1.69s	0.28s
9	*	0.32s
10	*	0.31s
30	*	0.41s
60	*	0.56s
90	*	0.65s
100	*	0.76s
300	*	2.78s
500	*	4.87s
700	*	8.21s
900	*	15.92s
1000	*	16.96s

Table 18 – Nondeterministic instances of the hiding experiment.

Instance	FDR4	Prototype
3	0.06s	0.26s
6	1.69s	0.27s
9	*	0.31s
10	*	0.30s
30	*	0.44s
60	*	0.60s
90	*	0.67s
100	*	0.79s
300	*	3.00s
500	*	4.91s
700	*	8.48s
900	*	16.01s
1000	*	17.46s

The results show that FDR struggles to analyse large parallel systems, with our prototype being a scalable alternative. Another composition operator that our experiments indicate that FDR cannot efficiently handle is hiding, which is also efficiently analysed by our prototype. These are very promising results, since ours is an academic prototype, while FDR is a very optimized commercial tool.

4.4 Threats to Validity

There are four threats to the validity of our experiments, discussed below. For each threat, we present the reasons for its occurrence, and the steps taken to minimise its effects.

- **Instance generation** The instances generated have the potential to favour one tool over the other, which can create misleading results. To address this, part of the specifications were selected from the literature, and the rest were created in such a way to provide the hardest scenario for our prototype. The nondeterministic instances, in particular, had the nondeterminism introduced in the least favourable position for the prototype.

- **Instance topology** All our experiments used regular topologies, to allow for the increase of the instance sizes. Due to this, we have no data regarding the efficiency of our prototype in other scenarios. Regular topologies can potentially be tackled by induction (CREESE; REED, 1999), so it is interesting to ensure that our strategy is scalable in the general case. Our expectation is that the efficiency of our strategy will the same in other scenarios, but this remains as a weakness to be addressed in future work.

- **Tool comparison** Our prototype was compared only with FDR4. To allow for a richer result, we need to compare it with other tools. These can include other model checkers, such as ProB (LEUSCHEL; BUTLER, 2003) and PAT (SUN et al., 2009), or theorem provers, like CSP Prover (BARTELS; KLEINE, 2011). The additional comparisons are not a trivial action, since the other tools use different semantic models of CSP, together with variations in syntax, and this remains as a weakness. It is important to note, however, that FDR4 is the main model checker for CSP, and the comparisons with it do provide relevant information.

- **Algorithm correctness** The algorithms used have no formal connection with our definition of determinism. This remains as the main threat to validity, and requires formal proofs in order to be addressed. The use of formal tools, such as Dafny (LEINO; MOSKAL, 2013), to provide a partial guarantee is a possibility.

5 CONCLUSION

In this dissertation we propose a local analysis strategy for the verification of determinism in specifications written in a subset of CSP. Our strategy is sound and it encompasses most of the main operators of CSP. With these features we are able to provide a viable alternative to existing techniques.

Our strategy is constructive and compositional. To check if a process is deterministic, we first check its components. The components themselves have their operands checked first, until we reach processes that are assumed to be deterministic, which we call Basic Processes. For each process that is known to be deterministic, we gather metadata about it. When checking a composition we use only the medatada of its operands, thus removing the need to re-evaluate processes that were already verified.

By considering a controlled subset of CSP, we know what the potential sources of nondeterminism are. With this knowledge we developed algorithms to check, for each composition operator, these particular sources. By knowing in advance what should be analysed, we do not need to check the entire state space, which allows the verification to scale. We performed some experiments and the results show that, in most cases, our approach scales better than that of FDR4, the main tool for verification of CSP models, specially when dealing with large parallel systems.

Despite its advantages, our strategy has some limitations. Its main restrictions are the incompleteness of the analysis, the use of a controlled subset of CSP, instead of the full language, and the lack of formal proofs.

- **Incompleteness.** Our analysis is not able to ensure that a negative result indicates nondeterminism. This limitation on the accuracy permeates the entire strategy. The algorithms not only can consider unreachable, nondeterministic, behaviours, in some circumstances, but can also fail to capture restrictions in the specification that remove nondeterminism. The metadata used is also a factor, since, for example, it always considers guards and conditionals to be true. These inaccuracies exist because we do not analyse the complete state space. As already mentioned, this is a common trade-off to improve efficiency. Our examples and case studies indicate good accuracy in the verification, but a more in depth study must be carried out.

- **Subset of CSP.** Our strategy is able to check specifications written in a restricted subset of CSP. By restricting the language, we limit the potential sources of nondeterminism, so that we can conduct our analysis. This restriction is needed for efficiency. It is important to note that, besides the inclusion of only part of the CSP operators, there are also restrictions on their use, such as the categorisation of Basic and Composite processes.

- **Lack of formal proofs.** We closely analysed the potential sources of nondeterminism in our selected subset of CSP, and carried out a number of examples and case studies, all of which provided correct results. This, however, although an indicator, is not enough to guarantee the soundness of our strategy. To achieve this goal formal proofs need to be developed, regarding both the adequacy of the metadata and the application of all the algorithms.

5.1 Related Work

As the focus of this dissertation is to provide a local analysis strategy to check for the absence of nondeterminism, our aim is to compare our strategy with others from the literature. Although there are global analysis techniques to check for nondeterminism, like the ones provided by tools such as FDR (GIBSON-ROBINSON et al., 2014), ProB (LEUSCHEL; BUTLER, 2003), and PAT (SUN et al., 2009), to the best of our knowledge, ours is the first approach to local analysis of determinism, not only in the context of CSP, but also of other formal modelling notations. Given the lack of direct comparative works, we discuss here local analysis that check other properties.

The local analysis of deadlock is a topic that has been studied in depth. Antonino, Sampaio and Woodcock (2014) propose two patterns for deadlock avoidance, together with a series of assertions to automatically and efficiently check adherence to these patterns. This approach is extended by Antonino et al. (2014), whose work presents and additional pattern for deadlock avoidance, together with assertions to locally verify adherence to it.

The transformation of a deadlock verification in a satisfiability problem is proposed by Antonino, Gibson-Robinson and Roscoe (2016a). This work presents a transformation framework to allow the use of SAT solvers. This approach is extended by improving its accuracy (ANTONINO; GIBSON-ROBINSON; ROSCOE, 2016b), and by generalizing it, allowing the verification of other properties, such as mutual exclusion (ANTONINO; GIBSON-ROBINSON; ROSCOE, 2017).

A constructive approach to check both deadlock and livelock in component-based systems written in CSP is proposed by Ramos, Sampaio and Mota (2009). This work follows a grey-box component driven architecture, using a number of composition rules that ensure the preservation of deadlock- and livelock-freedom. The livelock verification aspect of this work, however, is limited, with the analysis being trivial. To address this, Conserva Filho et al. (2016) propose a black-box constructive approach, by allowing the hiding of internal communications, to check for livelock in this component-based scenario. A version of this local analysis of livelock, for general CSP specifications, is proposed by Conserva Filho et al. (2018).

CSP is widely used for the development of verification techniques, but other notations can also be adopted. Francesca et al. (2011) propose a deadlock verification strategy

using the process algebra CCS. In this work the authors explore the use of AI techniques, namely Ant Colony Optimization, in this context. Bensalem et al. (2011) proposes a deadlock verification strategy using a C like language. This work is underpinned by the use of invariants, and the authors provide a tool capable of generating C code of the verified systems.

5.2 Future Work

In this section we present research avenues that can improve the verification strategy proposed in this dissertation. We consider both enhancements to the strategy itself and ways to make it more applicable in practice.

- **Development of formal proofs.** The more pressing issue with our strategy is the lack of formal proofs. As already mentioned, we need to establish the adequacy of the metadata used and the correctness of the algorithms.
- **Complexity of the algorithms.** A relevant follow-up is an study of the complexity of the algorithms used. This study is important to correctly analyse their efficiency, and detect and correct any bottlenecks. With this information we will be able to correctly present their efficiency, which is now done via a proxy, through our experimental results.
- **Extension of the subset used.** The widening of the subset of CSP considered can make the strategy more appealing and improve its applicability. This includes not only the inclusion of new CSP operators, such as renaming, but also the removal of some of the restrictions, allowing, for example, the use of process references other than tail recursion.
- **Refactoring of specifications.** A refactoring function that attempts to adapt a general CSP specification to our controlled subset can be an interesting addition to our strategy. It can work as a pre-processing step and has the possibility of lifting some of the restrictions without the need to change the strategy.
- **Revision of the metadata.** To improve the efficiency of our strategy, one possibility is the reshaping of the metadata, so that less information is stored, and what is stored is more easily accessible. This can have direct impact on the verification effort and on the amount of memory needed by the strategy.
- **Extension to other formalisms.** As a more ambitious future work, one possibility is to try to transfer the verification strategy to other formalisms. In particular, to other process algebras, since this would be the easier first step, and would allow for a greater reach for the strategy.

- **Improvement of the tool support.** Improvements to our prototype can range from the development of a user interface to internal optimizations, both of which would be a welcome addition. Two important features to be considered are the possibility to save the metadata of a process for future use, and the option for the user to allow the verification to continue even if a potential nondeterminism has been found. As a long term goal, the combination of techniques that verify deadlock, livelock, and determinism locally is a possibility. By identifying a subset of CSP shared by all of them, an integrated approach, and tool, to analyse all the three classical properties is viable.

- **Perform more case studies.** To further demonstrate the usefulness of our strategy, more case studies can be performed. In particular, case studies of systems used in industry would be ideal. This would allow us to draw more conclusions in regards to the efficiency and accuracy of the strategy.

REFERENCES

ANTONINO, P. R. G.; GIBSON-ROBINSON, T.; ROSCOE, A. W. Efficient deadlock-freedom checking using local analysis and sat solving. In: *Integrated Formal Methods*. [S.l.]: Springer International Publishing, 2016. p. 345–360. ISBN 978-3-319-33693-0.

ANTONINO, P. R. G.; GIBSON-ROBINSON, T.; ROSCOE, A. W. Tighter Reachability Criteria for Deadlock-Freedom Analysis. In: *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. [S.l.: s.n.], 2016. p. 43–59.

ANTONINO, P. R. G.; GIBSON-ROBINSON, T.; ROSCOE, A. W. Checking static properties using conservative SAT approximations for reachability. In: *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*. [S.l.: s.n.], 2017. p. 233–250.

ANTONINO, P. R. G.; OLIVEIRA, M. V. M.; SAMPAIO, A.; KRISTENSEN, K. E.; BRYANS, J. W. Leadership Election: An Industrial SoS Application of Compositional Deadlock Verification. In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. [S.l.: s.n.], 2014. p. 31–45.

ANTONINO, P. R. G.; SAMPAIO, A.; WOODCOCK, J. A Refinement Based Strategy for Local Deadlock Analysis of Networks of CSP Processes. In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. [S.l.: s.n.], 2014. p. 62–77.

BARTELS, B.; KLEINE, M. A CSP-based Framework for the Specification, Verification, and Implementation of Adaptive Systems. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. [S.l.: s.n.], 2011. p. 158–167.

BENSALEM, S.; GRIESMAYER, A.; LEGAY, A.; NGUYEN, T.; PELED, D. A. Efficient Deadlock Detection for Concurrent Systems. In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. [S.l.: s.n.], 2011. p. 119–129.

BOVE, A.; DYBJER, P. Dependent types at work. In: _____. *Language Engineering and Rigorous Software Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 57–99. ISBN 978-3-642-03153-3.

CONSERVA FILHO, M. S.; OLIVEIRA, M. V. M.; SAMPAIO, A.; CAVALCANTI, A. Local Livelock Analysis of Component-Based Models. In: *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*. [S.l.: s.n.], 2016. p. 279–295.

CONSERVA FILHO, M. S.; OLIVEIRA, M. V. M.; SAMPAIO, A.; CAVALCANTI, A. Compositional and Local Livelock Analysis for CSP. *Information Processing Letters*, v. 133, p. 21 – 25, 2018. ISSN 0020-0190.

CREESE, S. J.; REED, J. Verifying end-to-end protocols using induction with CSP/FDR. In: *Parallel and Distributed Processing*. [S.l.]: Springer Berlin Heidelberg, 1999. p. 1243–1257.

FAKHROUTDINOV, K. *UML 2.5*. 2015. Available at: <https://www.uml-diagrams.org/>.

FRANCESCA, G.; SANTONE, A.; VAGLINI, G.; VILLANI, M. L. Ant Colony Optimization for Deadlock Detection in Concurrent Systems. In: *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-22 July 2011*. [S.l.: s.n.], 2011. p. 108–117.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. FDR3 - A Modern Refinement Checker for CSP. In: *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference*. [S.l.: s.n.], 2014. p. 187–201.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A. *Java SE 8 Edition*. 2015. Available at: <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>.

HOARE, C. A. R. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5.

JESUS JúNIOR, J. B. de. *Design e Validação Formal de Sistemas de Controle de Voo Fly-By-Wire*. Master's Thesis (Dissertation) — CIn UFPE, 2009.

LEINO, R.; MOSKAL, M. *Co-Induction Simply: Automatic Co-Inductive Proofs in a Program Verifier*. [S.l.], 2013.

LEUSCHEL, M.; BUTLER, M. J. ProB: A Model Checker for B. In: *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*. [S.l.: s.n.], 2003. p. 855–874.

OTONI, R.; CAVALCANTI, A.; SAMPAIO, A. Local Analysis of Determinism for CSP. In: *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 — December 1, 2017, Proceedings*. [S.l.: s.n.], 2017. p. 107–124. ISBN 978-3-319-70848-5.

RAMOS, R.; SAMPAIO, A.; MOTA, A. Systematic Development of Trustworthy Component Systems. In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. [S.l.: s.n.], 2009. p. 140–156.

ROSCOE, A. *Understanding Concurrent Systems*. 1st. ed. New York, NY, USA: Springer-Verlag New York, Inc., 2010. ISBN 184882257X, 9781848822573.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733.

SUN, J.; LIU, Y.; DONG, J. S.; PANG, J. PAT: Towards Flexible Verification under Fairness. In: *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. [S.l.: s.n.], 2009. p. 709–714. ISBN 978-3-642-02658-4.

WATT, D. A. *Programming Language Design Concepts*. [S.l.]: John Wiley & Sons, 2004. ISBN 0470853204.

WOODCOCK, J.; CAVALCANTI, A. A Concurrent Language for Refinement. In: *5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16-17 July 2001*. [S.l.: s.n.], 2001.