Verification of Stochastic Process Calculi

Skrypnyuk, Nataliya

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

Verification of Stochastic Process Calculi

Nataliya Skrypnyuk

Kongens Lyngby 2011
IMM-PHD-2011-252

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Stochastic process calculi represent widely accepted formalisms within Computer Science for modelling nondeterministic stochastic systems in a compositional way. Similar to process calculi in general, they are suited for modelling systems in a hierarchical manner, by explicitly specifying subsystems as well as their interdependences and communication channels. Stochastic process calculi incorporate both the quantified uncertainty on probabilities or durations of events and nondeterministic choices between several possible continuations of the system behaviour.

Modelling of a system is often performed with the purpose to verify the system. In this dissertation it is argued that the verification techniques that have their origin in the analysis of programming code with the purpose to deduce the properties of the code's execution, i.e. Static Analysis techniques, are transferable to stochastic process calculi. The description of a system in the syntax of a particular stochastic process calculus can be analysed in a compositional way, without expanding the state space by explicitly resolving all the interdependencies between the subsystems which may lead to the *state space explosion* problem.

In support of this claim we have developed analysis methods that belong to a particular type of Static Analysis – Data Flow / Pathway Analysis. These methods have previously been applied to a number of non-stochastic process calculi. In this thesis we are lifting them to the stochastic calculus of Interactive Markov Chains (IMC). We have devised the Pathway Analysis of IMC that is not only correct in the sense of over-approximating all possible behaviour scenarios, as is usual for Static Analysis methods, but is also precise. This gives

us the possibility to explicitly decide on the trade-off between precision and complexity while post-processing the analysis results.

Another novelty of our methods consists in the kind of properties that we can verify using the results of the Pathway Analysis. We can check both qualitative and quantitative properties of IMC systems. In particular, we have developed algorithms for constructing bisimulation relations, computing (over-approximations of) sets of reachable states and computing the expected time reachability, the last for a linear fragment of IMC. In all the cases we have the complexities of algorithms which are low polynomial in the size of the syntactic description of a system. The presented methods have a clear application in the areas of embedded systems, (randomised) protocols run between a fixed number of parties etc.

Resumé

I datalogien bruges stokastiske procesalgebraer til at modellere nondeterministiske stokastiske systemer på en kompositionel måde. Som det normalt gælder for procesalgebraer, gør de det bekvemt at modellere den hierakiske struktur af systemer ved blot at identificere deres delsystemer og deres indbyrdes afhængigheder og kommunikationskanaler. I de stokastiske procesalgebraer håndteres derudover kvantitativ information om de usikkerheder og ventetider der indgår i systemet, hvilket giver en særlig udfordring i forbindelse med de ikkedeterministiske handlinger, som omgivelserne måtte påtrykke.

Det væsentligste formål ved at modellere IT systemer er normalt at kunne verificere deres opførsel. Hovedtesen for denne afhandling er, at de verifikationsteknikker der kendes fra programmeringssprog, fx programanalyse, kan overføres til analysen af stokastiske procesalgebraer. Det giver mulighed for at analysere systemerne med udgangspunkt i den kompositionelle måde i hvilken de er beskrevet, hvilket kan mindske risikoen for at skulle udfolde hele tilstandsrummet til at kunne blive eksponentielt stort og dermed vanskeligt håndterbart.

Som støtte for denne tese udvikler afhandlingen en række analysemetoder fra en bestemt type programanalyse, kendt som data flow analyse. De er tidligere blevet brugt på ikke-stokastiske procesalgebraer men først i denne afhandling på stokastiske procesalgebraer - konkret en meget udtryksfuld procesalgebra der tillader at udtrykke interaktive Markov kæder. I afhandlingen udvikles en avanceret data flow analyse, som giver en både præcis og korrekt beskrivelse af alle mulige opførsler af systemet. Det giver mulighed for kun at foretage tilnærmede analyser når det giver en tids- eller pladsmæssig gevinst.

En anden nyskabelse i de præsenterede resultater er den måde som kvalitative

og kvantiative egenskaber ved interaktive Markov kæder kan beskrives på. Der udvikles således algoritmer til at konstruere bisimulationsrelationer mellem IT systemer, algoritmer til at beregne de tilstande der kan nås, og for et lineært fragment af interaktive Markov kæder også tidsforbruget langs en given udførelse. I alle tilfælde opnås algorithmer af lav polynomiel kompleksitet. Afhandlingens resultater har brede anvendelsesmuligheder indenfor indlejrede systemer og de randomiserede kommunikationsprotokoller der indgår i service-orienterede IT systemer.

Preface

This dissertation was prepared at the department of Informatics and Mathematical Modelling, the Technical University of Denmark, in partial fulfilment of the requirements for acquiring the Ph.D. degree in Computer Science. The Ph.D. study has been carried out under the supervision of Professor Flemming Nielson (Technical University of Denmark), Professor Hanne Riis Nielson (Technical University of Denmark) and Professor Helmut Seidl (Technical University of Munich) in the period from September 2007 to February 2011. The study was financed by the Danish Agency for Science, Technology and Innovation in the project "Software Engineering for the 21'st Century" (DTU Informatics reference 95-15561).

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. Many ideas have been elaborated together with my excellent research collaborators. I have also received many valuable comments and suggestions from them. Besides the collaboration with my supervisors, the development of Pathway Analysis for IMC (Chapter 3) was done in cooperation with Henrik Pilegaard.

Lyngby, February 2011

Nataliya Skrypnyuk

Acknowledgements

I would like to thank my supervisor, Flemming Nielson, and my co-supervisors, Hanne Riis Nielson and Helmut Seidl, for providing me the opportunity to work on this interesting research project, for their excellent guidance, useful suggestions and comments, sharing ideas with me and helping in developing my own ideas during the entire time of my Ph.D. study.

I would like to thank the rest of the LBT group: Christian W. Probst, Ender Yuksel, Matthieu Queva, Alejandro Hernandez, Fuyuan Zhang, Piotr Filipiuk, Michael J.A. Smith, Sebastian Mödersheim, Lijun Zhang, Jose Quaresma, Michal Tomasz Terepeta, and Carroline D. P. K. Ramli, for creating a motivating and friendly working environment which allowed both for scientific discussions and for social activities. I would also like to thank the former members of the group – Sebastian Nanz, Christoffer Rosenkilde Nielsen, Fan Yang, and Ye Zhang – for their help, especially at the initial stage of my Ph.D. project.

I would like to thank Henrik Pilegaard for his collaboration on parts of my thesis and for motivating comments. Thanks to Michael J.A. Smith for many fruitful discussions and constructive comments, especially for putting my research into a broader perspective. I would like to thank Alan Mycroft for many useful comments concerning the way of presenting my research results.

I have appreciated my productive stays at the Technical University of Munich and would like to thank the members of the research group under the leadership of Helmut Seidl for their comments, ideas and insightful questions. I have benefited a lot from those fruitful exchanges of ideas and from broadening my knowledge of the field.

I would also like to thank in advance the evaluation committee: Holger Hermanns, Chris Hankin, and Christian W. Probst, for their helpful comments on this dissertation.

Thanks to Eva and Marian for their help with practical matters throughout the whole Ph.D. study.

Last but not the least, I would like to thank my family: my father, my mother, my brother, for their love and support. Thanks to Andrey for his support and understanding.

Contents

Chapter 1

Introduction

The study of concurrent systems has been a dynamically developing field of research in Computer Science in the last decades. Such systems can be recursively constructed out of several subsystems that progress in parallel, can exhibit nondeterministic behaviour and interact at some time points during their execution, or globally influence each other's execution in some way. Many real-world systems can be modelled as concurrent systems – embedded, distributed, networked, etc. – therefore their understanding and verification is of increasing importance.

Widely accepted formalisms for describing concurrent systems are *process calculi*, also called *process algebras*. Starting from the pioneering process calculi CCS [Mil80] and CSP [Hoa85], we have now a broad class of process calculi, some of which are more general and some are more adapted to specific application areas. We can name among others π-calculus [MPW92] (added communication between processes through channels) and its enhancement for modelling cryptographic protocols Spi-calculus [AG97], Ambients [CG98] (added movement of processes), and its version for biological systems BioAmbients [RPS$^+$04], a higher-order process calculus KLAIM [dNFP98] (a language for distributed systems), etc. Process calculi provide operators that make it possible to specify systems in a compositional way, by building more complex systems out of simple ones, and to explicitly declare nondeterministic choices. A syntactic description of a system in a particular process calculus is always finite, while it can also

describe an infinite system behaviour.

The behaviour of each subcomponent of a concurrent system can be represented by a labelled transition system, which is called its *semantic model*. Labelled transition systems consist of states and transitions between the states. The number of states and transitions can be finite or infinite. Each transition is annotated by a label that describes the meaning of the transition and influences the way in which a labelled transition system can interact with other labelled transition systems external to it – the latter are often called *external environments*. A finite expression in a syntax of a particular process calculus corresponds to a labelled transition system where all the interdependencies between subcomponents have been resolved and the behaviours of separate subsystems have been "merged together" in one integrated representation. In this way all possible behaviour of a model becomes explicitly declared. The rules according to which this transformation is performed, i.e. from a compositional system description in a particular process calculi into a labelled transition system, are often defined in the spirit of Structural Operational Semantics [Plo81].

Historically the first process calculi could only describe functional aspects of systems behaviour: they did not incorporate the notions of time and quantified uncertainty (for example, of probabilities). A number of *stochastic* process calculi have been defined later on that add quantitative information to the qualitative in the description of systems – among others, we can name TIPP [GHR92], PEPA [Hil96], EMPA [BG98], stochastic π-calculus [Pri95], stochastic CCS [KS08], and IMC [BH00]. With these calculi it is possible to model the same systems as with purely functional process calculi with added information on their performance and also new systems where quantitative information is decisive – for example, biochemical or randomised. A usual way to represent the time information is by using Continuous-Time Markov Chains as formal models of delays between qualitative progress steps in the system execution.

Modelling of a system is usually accompanied by the verification of its properties – for example, safety or liveness properties, – or by comparison with a reference implementation. For stochastic process calculi a new class of properties – performance properties – can be verified as well. Moreover, it becomes possible to express quantitatively the difference between two systems. The usual method of doing system verification is to use Structural Operational Semantics rules of the corresponding process calculus to build a labelled transition system out of the system's syntactic description. Consequently model-checking algorithms can be run on the constructed labelled transition system [CES86].

One potential problem with this approach is the state space explosion: a small description of a process in a corresponding process calculus's syntax can give rise to a large or even infinite labelled transition system, whose verification is

undecidable in the general case. Even if the state space of a particular system is finite, and therefore its verification is decidable, the existing model-checking algorithms are usually of high time and space complexity. For this reason there is a need for simplified, more efficient verification methods. The Static Analysis methods can provide us with the necessary methodology. They have been originally applied to the analysis of programs, but later on have shown to be useful in other areas as well, for example, in the analysis of biological systems or cryptographic protocols. In all cases they have been able to provide efficient algorithms, often at the price of precision.

Main Thesis. We can now formulate the *main thesis* of this dissertation:

> Static Analysis techniques can be used for verification in a syntax-directed and compositional way of systems modelled by stochastic process calculi.

Our main thesis is thus the statement that it is possible to "lift" the Static Analysis methods from a discrete / functional / qualitative world to a stochastic / non-functional / quantitative world, i.e. to apply them to stochastic process calculi. We will obtain a "concise description" (the result of the Static Analysis) of the semantics of a concurrent stochastic system, and can verify a number of properties with a chosen level of precision by post-processing it. Static Analysis can in particular be useful in order to compare the behaviour of such systems, to assess possible system behaviour at some point in the future, and to estimate the expected time interval until reaching that point.

1.1 Background

Static Analysis has initially been developed in the field of imperative programming languages. Its purpose was to check for errors in a program without actually executing it, purely by analysing the code (see [NNH99] for a detailed description of different Static Analysis techniques). Some of the Static Analysis techniques (Control Flow Analysis, Data Flow Analysis, etc.) have been later adapted to a variety of process calculi with a similar purpose – to deduce some properties of system's behaviour directly from the syntax, without actually building a complete labelled transition system induced by the semantics. In this thesis we have been inspired by the adaptation of Data Flow Analysis to CCS [NN07] and to Pathway Analysis for BioAmbients [NNPR04], [Pil07].

After the application of Data Flow Analysis methods to a syntactic description of a system, a deterministic finite automaton can be constructed based on the analysis results which *safely over-approximates* the execution scenarios of the labelled transition system induced by the semantic rules of a particular process calculus. By *safely* we mean that all existing execution scenarios of the labelled transition system are taken into account. By *over-approximation* we mean that some impossible scenarios can also be embedded into the deterministic finite automaton. This kind of approximation is also called *may-approximation* as the deterministic finite automaton represents the behaviour that *may* occur.

Also *must-approximations* are feasible with Data Flow Analysis methods where the deterministic finite automata represent only the existing execution scenarios of semantic models but not necessary all of them [NNN08]. While over-approximations are useful for deciding safety properties, under-approximations are useful for deciding liveness properties; besides, their comparison can help us to estimate the "error gap" of the analysis. In both cases it is a challenge to build the smallest possible automaton that is still good enough for checking interesting properties, i.e. for which the "error gap" is the smallest possible. Unlike with model-checking methods, infinite labelled transition systems can be analysed by Data Flow Analysis methods, and for finite systems we usually have lower time and space complexity while using Data Flow Analysis methods compared to model-checking methods.

In this PhD project we are mainly concentrating on the application of Data Flow Analysis or a broader class of Static Analysis methods to concurrent nondeterministic stochastic systems. The systems are modelled on a high level of abstraction as processes which synchronise and communicate with each other, and are described using stochastic process calculi. We have chosen one such calculus – the calculus of Interactive Markov Chains [BH00] (IMC) – for our analysis. The IMC is an orthogonal union of labelled transition systems and Continuous-Time Markov Chains. This calculus is useful for our purposes, as we can model purely stochastic systems, purely concurrent nondeterministic systems, and a combination of both in this calculus. Also syntactically the calculus of IMC is beneficial for us, as actions and delays are treated in a very similar way. Therefore the "lifting" of Data Flow Analysis methods from dealing only with actions to dealing with delays as well is quite straightforward. As IMC allows the presence of nondeterministic choices in a complete model, the power of Static Analysis methods can be exploited in this aspect.

As already mentioned, an alternative way to verify concurrent nondeterministic stochastic systems is to model check them. Model checking of stochastic systems, especially with inherent nondeterminism, is a problem with a high computational complexity: it can be solved by using numerical algorithms that may show themselves unstable on some instances. Therefore uniformisation

[HKMKS00] or discretisation [NZ10] techniques are often applied to such systems prior to model checking them, and the result is computed up to some precision level.

We give a schematic view of the relation between (stochastic) model-checking and Static Analysis methods applied to (stochastic) process calculi in Figure 1.1. A syntactic expression in a particular process calculus can be either analysed by Static Analysis methods (in our case by the Pathway Analysis) or a labelled transition system induced by the semantics of the process calculus can be built, eventually followed by merging states through abstraction of some of the state properties – see, for example, [KKLW07], [KKN09], [Smi10].

The verification of properties can take place by either post-processing the Pathway Analysis results in the first case or by model-checking methods carried out on the (abstract) labelled transition system. As it is described in Figure 1.1, post-processing the Pathway Analysis results can be understood as a model-checking method. In case the Pathway Analysis is not only correct but also precise, we could, for example, build the deterministic finite automaton based on the results of the Pathway Analysis which would be equal or bisimilar to the labelled transitions system induced by the semantic derivation rules and apply model-checking methods to it. If the Pathway Analysis is not precise then we get an over- or under-approximation which we can model check using, for example, three-valued logic – see, for example, [NN07] or [NNN08].

In general, verification based on Static Analysis may lead to insufficient precision but we might improve on it by some additional post-processing. Model-checking methods are based on exhaustive exploration of the state space, therefore they lead in principle to precise results, but the precision might be not sufficient due to a preceding abstraction, so several rounds of model refinement might be necessary in order to reach a sufficient precision level. Pathway Analysis results can also be post-processed in several rounds for reaching the necessary precision – for this to be possible we need that the Pathway Analysis is precise, i.e. that no information is lost while doing the analysis. As it is indicated in Figure 1.1, the relations between Model Checking and Static Analysis methods are quite complex.

1.2 Contributions

We will shortly describe the main contributions of our work and put them into relation to the previous work.

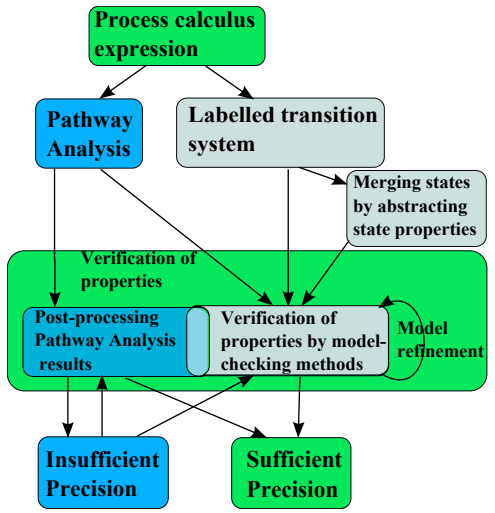

Figure 1.1 – Schematic illustration of the relations between Pathway Analysis and Model Checking of (stochastic) process calculi.

Pathway Analysis of a stochastic process calculus

We have adapted Pathway Analysis for the application to Interactive Markov Chains (IMC) systems [BH00]. A significant difference of IMC from other process calculi to which Data Flow / Pathway Analysis has been applied before (CCS, Bioambients etc.) is that IMC incorporates time information into the modelling of systems, while the other calculi are purely functional. Moreover, IMC makes use of a different synchronisation model: any number of actions can synchronise in IMC, not just two as in CCS or Bioambients. This is a a so-called CSP [Hoa85] or *multi-way* synchronisation. Which actions can/have to synchronise depends on the global structure of an expression in IMC calculus. We have defined an additional operator on the syntax of IMC in order to analyse the global "synchronisation structure" of an expression.

The second difference of our version of Pathway Analysis to the previous work is that our analysis is not only correct (it over-approximates the behaviour scenarios) but also precise (it also underestimates the behaviour scenarios), i.e. it predicts *exactly* the behaviour scenarios that are possible in the analysed system. We could achieve this result in particular due to the fact that IMC – unlike other process calculi to which Data Flow / Pathway Analysis has been applied before – has only finite semantic models [Her02]. The number of processes running in parallel and the way they are interacting in an IMC system do not change

throughout their lifetime though some of the processes may become inactive.

The obvious advantage of making a precise analysis is that we can decide which level of precision is acceptable while doing the verification of a system using the analysis results. The results of the analysis in this case succinctly describe all the possible behaviour of an initial IMC process and any process into which it can evolve with the time.

Syntax-directed construction of bisimulation and reachability relations

Conducting Pathway Analysis on the syntax of IMC systems has allowed us to devise algorithms for computing in a compositional way relations on the states of labelled transition systems that are induced by the semantics. This has allowed us to construct relations in time and space complexities which are low polynomial in the size of the syntactic description of an IMC system and not in the number of states in the labelled transition system, which can be exponentially larger than the size of the syntax.

Identifying bisimilar states can be very beneficial: bisimilar states behave "in the same way" and therefore can be merged, which leads to smaller systems and helps to alleviate the problem of state space explosion [Par81], [Mil89]. We have devised conditions based on the results of the previously conducted Pathway Analysis under which states are guaranteed to be bisimilar. This is to the best of our knowledge the first case of applying Pathway Analysis to calculating bisimulation relations.

We have defined two bisimulation relations, of which the first is contained in the second one. Even though the devised bisimulation relations are in general not the coarsest ones for IMC systems, they can lead to a considerable state space reduction and are the coarsest bisimulation relations on a subclass of IMC systems. Our methods can be extended in order to compute coarser bisimulation relations than currently.

We have also devised several algorithms for computing (in general an over-approximation of) reachable states using the results of the previously conducted Pathway Analysis. We have considered the reachability problem in different aspects – which states are possible in principle, which states are reachable from a particular state, which states can be reached repeatedly – and have devised suitable algorithms for these problems which are time polynomial in the length of the syntax of an IMC system. We show how the estimations can be recomputed after new information becomes available (i.e. after one semantic step taken by

a system) with a lower complexity than if computed from scratch; in this way the precision of the estimation can be improved.

Timed reachability on linear fragment

We have devised an algorithm for computing the minimum expected time needed to reach a state on a linear fragment of IMC. The algorithm requires the application of an additional operator to the syntactic description of an IMC system and the post-processing of the results which can be implemented in time and space that is polynomial in the size of the process. The minimum time is computed under the class of simple schedulers. This is an important step in the direction of the verification of time properties of stochastic nondeterministic systems in a syntax-directed way.

The contributions presented above support our thesis, i.e. we can make a preliminary conclusion that Static Analysis can in fact be used in the verification of systems that can be modelled in stochastic process calculi and that give rise to finite labelled transition systems – in particular, to IMC systems. We can perform the correct and precise Pathway Analysis of such systems, and using the analysis results construct bisimulation relations, compute (an over-approximations of) reachable states, and for linear systems also tackle the minimum expected time reachability problem.

1.3 Dissertation outline

In Chapter 2 we discuss necessary preliminaries for the following chapters. In particular, we discuss the syntax and the semantics of the IMC calculus in Section 2.1. We also present Static Analysis methods in general and its subclass Data Flow / Pathway Analysis methods in Section 2.2. We give a short overview of model-checking methods applied for verification of stochastic systems with or without nondeterminism in Section 2.3.

Our main contributions are presented in Chapters 3, 4, and 5. In Chapter 3 we first present the syntactic/semantic specifics of the version of IMC which we have named *guarded* IMC (IMC^{G}) (Sections 3.2-3.3). We present then the Pathway Analysis of IMC^{G} in Section 3.4 and prove that it is correct and precise in Section 3.5.

In Chapter 4 we first transfer the usual notion of a strong bisimulation relation on IMC systems to the Pathway Analysis setting in Section 4.2. After that we introduce two bisimulation relations on $\mathrm{IMC}^{\mathrm{G}}$ systems that can be computed using the Pathway Analysis results and prove their relation to each other in Sections 4.3 and 4.4.

In Chapter 5 we use the results of the Pathway Analysis to tackle a number of reachability problems. In Section 5.1 we discuss how we can exclude some states as impossible based on the Pathway Analysis results. In Section 5.2 we discuss how the usual model-checking methods can be applied in the Pathway Analysis setting to verify the existence of states with predetermined properties. In Section 5.3 we give the algorithms for assessing which states are reachable from some initial state and how to update the assessment after one semantic step. In Section 5.4 we discuss the timed reachability for a linear fragment of IMC.

We summarise, make conclusions, give possible directions for future work, and evaluate their implementability in Chapter 6.

Chapter 2

Setting the scene

The technical developments of the thesis consist in the application of Data Flow / Pathway Analysis methods [NN07] to a version of the process calculus of *Interactive Markov Chains* (IMCs) [BH00]. The novelty of the approach is that the calculus of IMC is considerably different from the ones that Pathway/Data Flow Analysis and Static Analysis in general have been applied until now – these are, for example, CCS [Mil80], BioAmbients [RPS$^+$04], π-calculus [MPW92], that do not have the notions of time and probability distributions that determine the probabilities of taking some of the transitions (as it is the case in IMC) and admit infinite semantic models (semantic models in IMC are finite).

Also properties checked with the help of Pathway Analysis results are different – in previous work, for example, in [NN07] and [Pil07], the main purpose was building a deterministic finite automaton that in a sense over-approximates the possibly infinite semantic model by simulating it, with the granularity function regulating the size of the automaton and therefore the degree of precision in the over-approximation. We had our sight on a wider range of problems. First, we can build with the results of the Pathway Analysis a deterministic finite automaton that bisimulates the system that has been analysed. Moreover, we solve the problems of reachability and time reachability in an efficient way without building a semantic model, just based on the results of the Pathway Analysis. On the other hand, we have gained a few insights concerning the limitations of applying Pathway Analysis (and Static Analysis in general) to process calculi.

The Pathway Analysis that we have developed strongly depends on the properties of IMC. Therefore in Section 2.1 in this chapter we will first of all shortly present the stochastic process calculus of IMC, its syntax and semantics, and also the main differences of IMC from other widely used stochastic process calculi. We will then present the principles of Pathway/Data Flow Analysis for programs and for process calculi in Section 2.2. Our own Pathway Analysis is the extension thereof. We have namely the same operators on IMC syntax as the original Data Flow Analysis for CCS, but we have extended it with several new operators.

In the last part of the chapter, i.e. in Section 2.3, we will give an overview of the current state of verifying IMC systems and, closely connected to them, *Continuous-Time Markov Decision Processes* (CTMDPs). We will present both properties (i.e. logics) that are currently being checked and methods that are often used in doing this. It is namely the case that direct model checking of large IMC systems is often not possible due to the presence of both non-determinism and stochastic features; therefore abstraction or/and approximation techniques need to be used. Our own methods can be seen as an alternative to currently used abstraction methods, hence it will be advantageous for the reader to get an overview of the state-of-the-art.

2.1 IMC calculus

2.1.1 Main features of IMC

Interactive Markov Chains, shortly IMC, is a stochastic process calculus that combines features of non-stochastic process calculi and of *Continuous-Time Markov Chains* (CTMCs). It was introduced by Brinksma and Hermanns in the 90s [BH00] and was extended in [Her02]. IMC features most wide-spread compositional operators of process calculi, in particular the *choice* operator (+) between two possible system transitions, CSP-style [Hoa85] or also often called TCSP-style [SDBR84] *parallel composition*, and in particular *synchronisation* of actions, when two or more actions can be executed only "together", otherwise they cannot be executed at all, *prefixing*, action *internalisation* (hide), and *recursion* operators. See Table 2.1 below for the formal definition of the IMC syntax. We will come back to these operators in some more detail in the next chapter in Section 3.2 while presenting a version of IMC that we will work on – we have called it IMC^{G} or "guarded" IMC.

There are two types of prefixes in IMC – "action" and "delay" prefixes. An

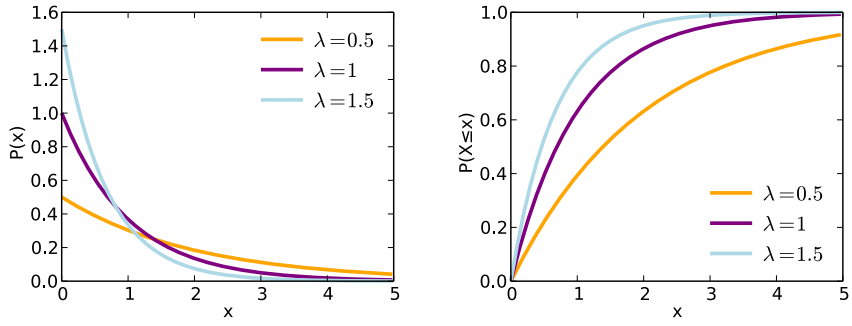

Figure 2.1 – Probability density functions (figure on the left) and cumulative distribution functions (figure on the right) for exponential distribution with three different rate parameters. Source: www.wikipedia.org.

intuitive understanding would be that "actions" denote moments in time (with 0 duration) when some events happen. The name of the action points to the nature of the event – for example, some action a might mean that a request from a client application has just arrived at the server or that a user has pushed a button on the coffee machine, depending on the previously agreed notation. "Delay" prefixes on the other hand denote non-zero time durations when no events happen – this can be, for example, time duration between the user's pushing of the button on a coffee machine and the moment the coffee is ready.

All delays in IMC are exponential, i.e. their duration is drawn from a special class of probability distributions, namely, exponential distributions. In order to define a particular exponential distribution from a class of exponential distributions it is enough to specify only one parameter – its rate, which is a positive real number. The density of exponential distribution with a rate λ is given by the function $\lambda e^{-\lambda t}$ for $t \geq 0$ and the cumulative distribution function is then $\int_0^t \lambda e^{-\lambda t} dt = 1 - e^{-\lambda t}$ [Tri02]. Intuitively the value of the cumulative distribution function at $t \geq 0$ determines the probability that an exponentially distributed delay with the rate λ will be over before the time point t. See the graphics in Figure 2.1 for exponential density and exponential distribution functions with rate parameters from $\{0.5, 1, 1.5\}$.

The expected duration of exponentially distributed delays with the rate λ is $\frac{1}{\lambda}$ and the standard deviation is $\frac{1}{\lambda}$ as well. If two or more exponentially distributed delays are possible, then the system will "execute" (i.e. move along with) a delay transition whose respective delay expires first. This situation is called a *race condition*. The *sojourn time* (i.e. the time until the first delay expires) is then distributed with the rate $\sum_{i=1..n} \lambda_i$ if $\lambda_1..\lambda_n$ are rates of all possible (also called

"enabled") delay transitions – this fact can be easily derived from the properties of exponential distributions.

Exponentially distributed delays are often used in stochastic process calculi, as they are easy to describe (only one parameter is needed, i.e. the rate) and memoryless, i.e. do not take system execution history into account. Furthermore many delays in real systems (for example, time until an error occurs or a request for a service arrives at the server) can be best described by an exponential distribution. At the same time each so-called phase-type distribution [O'C99] can be represented by a finite CTMC with exponentially distributed delays, more exactly by a distribution of time until reaching an absorbing state in the CTMC. Phase-type distributions are interesting as they allow to approximate any other kind of probability distribution with an arbitrarily high precision by using more and more states in a CTMC [Tri02]. We might mention here that in many real systems the delay rates change in fact over the time – for example, the rate of server requests might be different in the day- and night-time etc. This discussion is however outside the scope of this thesis.

2.1.2 Syntax and Semantics of IMC

Syntax of IMC

We will shortly present the syntax and semantics of IMC systems [BH00]. Let **Act** denote a set of names of external actions, τ be a distinguished name for an internal action such that $\tau \notin \mathbf{Act}$, **Var** be a set of names of variables (often called "process variables") different from action names. Then each IMC system can be described by an $\mathcal{E}$-type expression with a syntax inductively defined in Table 2.1.

The rules (1)-(6) in Table 2.1 define the so-called *linear* fragment of IMC. The rules (2)-(3) define a prefix operator, the rule (4) defines a choice operator and the rule (6) defines a halted process. The rule (5) is often called a recursive definition of a variable. See the definition of a weakly guarded and closed expressions below.

Definition 2.1. *A variable X occurs* bound *in an expression E if it occurs contained in a subexpression $\underline{X := E'}$ of E for some E'. A variable X occurs* free *in an expression E if there is no E' such that $\underline{X := E'}$ is a subexpression of E. A variable can occur both free and bound in the same expression. An expression is called* closed *if there is no variable that is occurring free in it.*

$$
\begin{array}{lcll}
P & ::= & X & | \quad (1) \\
 & & \mathsf{a}.P & | \quad (2) \\
 & & (\lambda).P & | \quad (3) \\
 & & P + P & | \quad (4) \\
 & & \underline{X := P} & | \quad (5) \\
 & & \mathbf{0} & \quad (6) \\
\mathcal{E} & ::= & P' & | \quad (7) \\
 & & \mathsf{hide}\ A\ \mathsf{in}\ P' & | \quad (8) \\
 & & P' \ {}_{|\!|A|\!|}\ P' & | \quad (9)
\end{array}
$$

Table 2.1 – Syntax of IMC: IMC expressions are $\mathcal{E}$-type expressions, with $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $\lambda \in \mathbb{R}^+$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$, P' is a closed and weakly guarded P-type expression.

A variable X is called weakly guarded *in an expression E if its every occurrence in E is contained in a* guarded expression*, i.e. expression of the form $\mathsf{a}.E'$ or $(\lambda).E'$ for some E'. An expression is called* weakly guarded *if all variables that are occurring in it are occurring weakly guarded.*

Intuitively, in a weakly guarded expression there is always at least one action or delay in the definition of each variable. The expression $\underline{X := X}$ is thus ruled out as X is not uniquely defined: in fact, any expression composed according to the rules (1)-(6) would be a solution for X. On the other hand, $\underline{X := \mathsf{a}.X}$ is weakly guarded and also uniquely defines the variable X. We talk about "weakly" guarded expression because it is allowed that X is guarded by the internal action τ. A "strong" guardedness is sometimes required in [Her02], namely, when the axioms for the weak bisimulation are presented for the linear fragment of IMC.

Note that the *parallel composition* or *synchronisation* (rule (9)) and the *internalisation* (rule (8)) operators are only applied to closed expressions. As we will see from the definition of the IMC semantics, this ensures the finiteness of the semantic model for every IMC expression. Parallel composition puts two IMC process to run "in parallel", synchronising on a set of actions (for only one action in the set we may leave out the set notation, just writing an action name) and internalisation makes internalised actions behave as if they were internal actions, see the semantics of IMC in Table 2.2 below. We will have a more permissive syntax for our variation of IMC concerning application of parallel composition and internalisation operators (see Table 3.1 in Section 3.2), but we will be a bit more restrictive with the guardedness condition.

As we will see from the definition of the IMC semantics, $\underline{X := \mathsf{a}.X}$ is at the same time a definition of the variable X and a definition of a process which continuously repeats the action a. These two meanings can be made more clear if we take as an example the IMC expression $\underline{X := \mathsf{a}.X} + \underline{Y := \mathsf{b}.Y}$. This IMC expression defines the variables X and Y and it defines at the same time a process which either continuously repeats the action a or continuously repeats the action b.

We have presented the syntax of IMC according to [Her02]. In [BH00] the authors have used a different syntax, where the two aspects mentioned above – definitions of variables and definitions of behaviour – are separated. The example from above $\underline{X := \mathsf{a}.X} + \underline{Y := \mathsf{b}.Y}$ will be written in the syntax from [BH00] as a set of definitions $[X = \mathsf{a}.X, Y = \mathsf{b}.Y]$ and the IMC expression determining the behaviour $X + Y$. Both variants of syntax are semantically equivalent. We have chosen (and subsequently extended) the syntax from [Her02] for technical reasons.

Semantics of IMC

The interleaving semantics of IMC is defined in Table 2.2 using Structured Operational Semantics (SOS) in the style of Gordon Plotkin [Plo81]. There are two kinds of transitions – "interactive" or "action" transitions denoted by solid arrows and "exponential" ("Markovian", "delay") transitions denoted by dashed arrows. It can be proved by induction of a transition derivation that from $E \xrightarrow{\alpha} E'$ for $\alpha \in \mathbf{Act} \cup \{\tau\}$ or $E \overset{\lambda}{\dashrightarrow} E'$ for $\lambda \in \mathbb{R}^+$ follows that E' as an IMC expression as well. We will sometimes skip decorating a transition by a corresponding action name or delay rate and denote the fact that there exists a transition from E to E' simply by $E \longrightarrow E'$.

We will say that an IMC expression E_n can be *reached* from another IMC expression E_1 in case there exists a sequence of transitions $E_i \longrightarrow E_{i+1}$ for $1 \leq i < n$, $n > 1$. We can denote the fact of the existence of such sequence of transitions as $E_1 \xrightarrow{*} E_n$. We can also say that E_n is *reachable* from E_1, is a *derivative expression* or a *derivation* of E_1. We will also call in the following an IMC expression E and its derivative expressions *states*, as it is usual for *state transition systems*. In fact the semantics of IMC is defined in terms of *labelled transition systems* (LTSs) [Plo81] of a special kind.

The semantics of an IMC expression is defined as a set of all interactive transitions and a multiset of all Markovian transitions derivable for it and its derivative expressions from the SOS rules in Table 2.2. This means that we need

to take into account *all* derivable Markovian transitions, but only one of each kind of derivable interactive transitions. For example, $\underline{X := \mathsf{a}.X + \mathsf{a}.X}$ has exactly the same semantic model as $\underline{X := \mathsf{a}.X}$ but the semantics of the expression $\underline{X := (\lambda).X + (\lambda).X}$ is not the same as of the expression $\underline{X := (\lambda).X}$. We will explain the difference between interactive and Markovian transitions in the next section.

Another unusual feature of IMC semantics is the condition $F \stackrel{\tau}{\not\longrightarrow}$ which means that there is no IMC expression F' such that the transition $F \stackrel{\tau}{\longrightarrow} F'$ is derivable from the rules in Table 2.2. This condition is needed in order to ensure the *maximal progress* assumption (see explanation below). It is always decidable whether $F \stackrel{\tau}{\not\longrightarrow}$ for an arbitrary IMC expression F due to the guardedness of variable occurrences by either action or delay and the requirement that every variable is defined – see the requirement of the weak guardedness and closeness in rules (7)-(9) in Table 2.1. This refers of course also to transitions decorated by any other action name or delay, i.e. it is always decidable whether $F \stackrel{\alpha}{\not\longrightarrow}$ for all $\alpha \in \mathbf{Act}$ or $\alpha \in \mathbb{R}^+$. The state E' such that $E \stackrel{\alpha}{\longrightarrow} E'$ for $\alpha \in \mathbf{Act} \cup \{\tau\}$ or $E \stackrel{\lambda}{\dashrightarrow} E'$ for $\alpha \in \mathbb{R}^+$ is uniquely defined for fixed E and α or E and λ.

Note that a *structural congruence* is not defined for IMC expressions, but is rather "incorporated" into the SOS rules of IMC. Thus, the rules (3) and (4) for action-transitions and the rules (11) and (12) for delay-transitions in Table 2.2 express that for two summands their order of appearance does not matter, which is a typical equivalence in the structural congruence [Mil99]. The same refers to the order of parallel processes (rules (4) and (5), (13) and (14)). For recursion unfolding (rules (9) and (16)) the structural congruence rules usually state that the behaviour of a process with and without recursion unfolding is the same, and this is also the case for the rules (9) and (16). However, due to the specifics of IMC syntax we need to do recursion unfolding before executing some prefix from the process definition. From the semantic rules (9) and (16) we can see that variable definitions are replicated after their unfolding.

It has been proved in [Her02] that IMC expressions have finite semantic models. The intuitive reason for this is that the parallel operator is always applied "on top" of variable definitions. As a consequence no new processes can be "spawned": for example, the process $\underline{X := \mathsf{a}.X \,||\emptyset||\, \mathsf{b}.X}$ is not allowed. An IMC process can therefore only repeat the same behaviour patterns and its semantics is finite. The restriction on application of the internalisation operator is also necessary in order to ensure finiteness of semantic models: for example, the process $\underline{X := \mathsf{hide}\ \emptyset\ \mathsf{in}\ \mathsf{a}.X}$ has the infinite semantic model due to the repetition of the hiding construct after each transition.

$$\mathsf{a}.E \xrightarrow{\mathsf{a}} E \quad (1) \qquad (\lambda).E \xrightarrow{\lambda} E \quad (10)$$

$$\frac{E \xrightarrow{\mathsf{a}} E'}{E + F \xrightarrow{\mathsf{a}} E'} \quad (2) \qquad \frac{E \xrightarrow{\lambda} E' \qquad F \not\xrightarrow{\tau}}{E + F \xrightarrow{\lambda} E'} \quad (11)$$

$$\frac{F \xrightarrow{\mathsf{a}} F'}{E + F \xrightarrow{\mathsf{a}} F'} \quad (3) \qquad \frac{F \xrightarrow{\lambda} F' \qquad E \not\xrightarrow{\tau}}{E + F \xrightarrow{\lambda} F'} \quad (12)$$

$$\frac{E \xrightarrow{\mathsf{a}} E' \qquad \mathsf{a} \notin A}{E \,|\!|A|\!|\, F \xrightarrow{\mathsf{a}} E' \,|\!|A|\!|\, F} \quad (4) \qquad \frac{E \xrightarrow{\lambda} E' \qquad F \not\xrightarrow{\tau}}{E \,|\!|A|\!|\, F \xrightarrow{\lambda} E' \,|\!|A|\!|\, F} \quad (13)$$

$$\frac{F \xrightarrow{\mathsf{a}} F' \qquad \mathsf{a} \notin A}{E \,|\!|A|\!|\, F \xrightarrow{\mathsf{a}} E \,|\!|A|\!|\, F'} \quad (5) \qquad \frac{F \xrightarrow{\lambda} F' \qquad E \not\xrightarrow{\tau}}{E \,|\!|A|\!|\, F \xrightarrow{\lambda} E \,|\!|A|\!|\, F'} \quad (14)$$

$$\frac{E \xrightarrow{\mathsf{a}} E' \qquad F \xrightarrow{\mathsf{a}} F' \qquad \mathsf{a} \in A}{E \,|\!|A|\!|\, F \xrightarrow{\mathsf{a}} E' \,|\!|A|\!|\, F'} \quad (6)$$

$$\frac{E \xrightarrow{\mathsf{a}} E' \qquad \mathsf{a} \notin A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow{\mathsf{a}} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (7) \qquad \frac{E \xrightarrow{\lambda} E'}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow{\lambda} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (15)$$

$$\frac{E \xrightarrow{\mathsf{a}} E' \qquad \mathsf{a} \in A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow{\tau} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (8)$$

$$\frac{E\{\underline{X := E}/X\} \xrightarrow{\mathsf{a}} E'}{\underline{X := E} \xrightarrow{\mathsf{a}} E'} \quad (9) \qquad \frac{E\{\underline{X := E}/X\} \xrightarrow{\lambda} E'}{\underline{X := E} \xrightarrow{\lambda} E'} \quad (16)$$

Table 2.2 – Structural Operational Semantics of IMC: $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $\lambda \in \mathbb{R}^+$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$.

An example of an IMC system together with it's semantic model is shown in Figure 2.2 (with a simplified syntax for better readability). We have two systems S and C (for example, a server and a client) which are synchronising. They execute together the action a and then need some time to process information internally before the next interaction becomes possible. There are two different behaviours for a client – if $\lambda_1 > \lambda_2$, then we will have one faster information processing and one slower processing for a client. Remember that a higher rate for an exponential delay means a lower expected delay's duration. The server cannot influence which kind of internal processing the client will have.

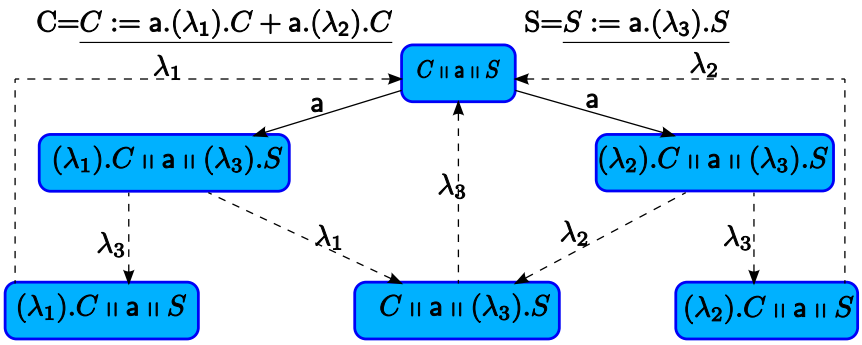

Figure 2.2 – Syntax and semantics of IMC process $\underline{C := \mathsf{a}.(\lambda_1).C + \mathsf{a}.(\lambda_2).C} \parallel \mathsf{a} \parallel \underline{S := \mathsf{a}.(\lambda_3).S}$.

Interactive and Markovian transitions

Let us shortly discuss the main differences between so-called "action"- and "delay"-transitions in IMC. The exact number of action-transitions with the same action name from one state in an IMC system to another state does not have any importance for the semantics. This is totally different for delay-transitions: all delay transitions are important for determining the complete semantics of IMC. The reason is that we can compute the probability of taking a transition to another state in a correct way only if we know all delay transitions possible for the initial state.

Take as an example the IMC expression $(\lambda_1).\mathsf{a}.\mathbf{0} + (\lambda_1).\mathsf{a}.\mathbf{0} + (\lambda_2).\mathsf{b}.\mathbf{0}$. The probability to take a transition into the state $\mathsf{a}.\mathbf{0}$ is then equal to $(\lambda_1+\lambda_1)/(\lambda_1+\lambda_1+\lambda_2)$ and the probability to take a transition into the state $\mathsf{b}.\mathbf{0}$ is equal to $\lambda_2/(\lambda_1+\lambda_1+\lambda_2)$. The reason is that the time until the state $(\lambda_1).\mathsf{a}.\mathbf{0}+(\lambda_1).\mathsf{a}.\mathbf{0}+(\lambda_2).\mathsf{b}.\mathbf{0}$ decides to "move on" is given by the minimum of all transition delays, which is as we have seen above exponentially distributed itself with the rate equal to the sum of rates of all outgoing transitions: $\lambda_1 + \lambda_1 + \lambda_2$. In a similar manner the minimum of two λ_1-transitions is exponentially distributed with the rate $\lambda_1 + \lambda_1$. The probability that one exponential delay completes before the other one can be determined as the ratio of the rate of the first exponential delay and the cumulative rate, therefore we get $(\lambda_1 + \lambda_1)/(\lambda_1 + \lambda_1 + \lambda_2)$ as the probability to move to the state $\mathsf{a}.\mathbf{0}$.

From the discussion above we can conclude that in case there are several delay-transitions between two states, they can be substituted by one transition with the rate equal to the sum of rates. In the example above, the process $(\lambda_1 *$

2).$\mathsf{a}.\mathbf{0} + (\lambda_2).\mathsf{b}.\mathbf{0}$ is equivalent to the process $(\lambda_1).\mathsf{a}.\mathbf{0} + (\lambda_1).\mathsf{a}.\mathbf{0} + (\lambda_2).\mathsf{b}.\mathbf{0}$.

Probabilistic choices in IMC are thus characteristic for choosing among several delay transitions. In most other cases (between two interactive transitions, between an interactive transition and a Markovian transition) the choice in IMC is done nondeterministically, but there is one important exception – the principle of *maximal progress*. The maximal progress is the only case in IMC in which actions and delays influence each other.

The maximal progress assumption requires that if an internal action and a delay transition are enabled at the same time then the internal transition is chosen for execution. An intuitive justification is that internal actions are always executed "on their own", i.e. never synchronise, and are immediate; so there is no reason for a system to delay something that can be done immediately. On the other hand, a Markovian transition never executes after exactly 0 time units, it always takes some time before it "fires", therefore the internal transition always "wins". An external action can be on the other hand blocked by the environment, so its immediate execution cannot be guaranteed.

We have already mentioned that IMC has an interleaving semantics. This means that non-synchronising actions of two parallel processes can be executed in any order. For example, we may have both $\mathsf{a}.\mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathsf{b}.\mathbf{0} \xrightarrow{\mathsf{a}} \mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathsf{b}.\mathbf{0} \xrightarrow{\mathsf{b}} \mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathbf{0}$ and $\mathsf{a}.\mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathsf{b}.\mathbf{0} \xrightarrow{\mathsf{b}} \mathsf{a}.\mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathbf{0} \xrightarrow{\mathsf{a}} \mathbf{0} \,{}_{|\!|\emptyset|\!|}\, \mathbf{0}$. From the SOS rules in Table 2.2 follows that the same holds for delays instead of actions.

In fact, interleaving semantics for delays is in concordance with the properties of exponential distribution, in particular, the memoryless property. Take as an example the process $(\lambda_1).\mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, (\lambda_2).\mathsf{a}.\mathbf{0}$. According to the interleaving semantics, the only possible executions for this process are $(\lambda_1).\mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, (\lambda_2).\mathsf{a}.\mathbf{0} \overset{\lambda_1}{\dashrightarrow} \mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, (\lambda_2).\mathsf{a}.\mathbf{0} \overset{\lambda_2}{\dashrightarrow} \mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, \mathsf{a}.\mathbf{0} \xrightarrow{\mathsf{a}} \mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, \mathbf{0}$ and $(\lambda_1).\mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, (\lambda_2).\mathsf{a}.\mathbf{0} \overset{\lambda_2}{\dashrightarrow} (\lambda_1).\mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, \mathsf{a}.\mathbf{0} \overset{\lambda_1}{\dashrightarrow} \mathsf{a}.\mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, \mathsf{a}.\mathbf{0} \xrightarrow{\mathsf{a}} \mathbf{0} \,{}_{|\!|\mathsf{a}|\!|}\, \mathbf{0}$. This would mean, that the fact that the delay-transition with the rate λ_1 has completed its execution does not influence the duration of the second delay transition – it is still exponentially distributed with the rate λ_2. The situation is similar if the transition with the rate λ_2 has completed first – then the process $(\lambda_1).\mathsf{a}.\mathbf{0}$ can just proceed with the λ_1-transition. Such simplicity in interleaving of delay transition is due to the memoryless property of exponential distribution: the already passed time does not influence the expected transition duration and the form of time distribution until the event occurs.

2.1.3 Nondeterminism and probabilistic choices in IMC

Design choices in IMC

We will discuss in this section the differences between IMC and other stochastic process calculi that have a similar expressive power. This will also give us some insights into the SOS rules of IMC systems that have been defined in Table 2.2. An overview of design choices made for widely used stochastic process calculi can be found in [BH00] and [KS08].

One of the design decisions in constructing IMC was a choice of synchronisation model. There are two main synchronisation models that have been used until now: so-called CCS synchronisation [Mil80] and CSP synchronisation [Hoa85]. In the first case only two explicitly denoted actions can synchronise with each other (for example, a and $\bar{\mathsf{a}}$), while in the second case any number of actions can synchronise – this is clearly the case for IMC. The second case allows for more flexibility in building bigger systems from smaller ones, but it also harder to analyse, as we will see below in the definition of Pathway Analysis for IMC.

Some difficulties in understanding IMC semantics may arise from the fact that there is no explicit distinguishing between *external* and *internal* choice in IMC. An external choice is the one that is made by an external environment. This is modelled in IMC as a possibility of blocking some actions that have to be synchronised. An internal choice is made by a scheduler independently of the environment. Internal choices can be modelled in IMC as choices between actions with the same action name. The rest of the situations (i.e. choices between several external actions with different names none of which is blocked by the environment) may be interpreted as external or internal choices, dependent on possible environments that we might not know yet.

Probably the most important design choice in IMC concerns the nature of interactions between actions and delays. At the time of the creation of IMC there had already been proposed a number of stochastic calculi. The most well-known example is probably Performance Evaluation Process Algebra (PEPA) [Hil96]. In PEPA each action has a rate. There are "active" actions, whose duration is exponentially distributed with their rate, and "passive" actions whose rate is infinite. Active actions can be executed on their own with their own rate. Passive actions on the other hand cannot be executed in isolation, but only in synchronisation with active actions; the joint execution has the execution rate of the active action. If there is more then one passive action which is executed simultaneously with an active action, the joint execution rate is still defined by the active action in the same way.

A question arises what happens if two or more active actions are executed simultaneously – and that can happen according to the semantics of PEPA. In this case the joint rate of two or more active actions is assigned to be the minimum of all their rates [Hil96]. The intuitive explanation that the authors of PEPA have proposed is that simultaneous execution of two actions is not the same as two totally independent executions, where we just wait for both of them to complete. In the last case we would have a random variable which is not exponentially distributed anymore, because the class of exponential distributions is not closed under maximum. We would rather get a random variable that is governed by a phase-type distribution [O'C99]. In PEPA it is assumed instead that two processes develop together, and they do it with the tempo of the slowest process. This is similar to a situation where two runners are "chained" together, so they can only run with the speed of the slowest runner.

Other possibilities have been proposed in other stochastic process calculi, for example, multiplications of two rates (sometimes called "mass action law") in MTIPP [HR94] and in the variant of PEPA for modelling biomolecular systems [CGH05]. This solution can be easily understood if one of two actions participating in a transition is considered as "passive" and its rate as a "scaling factor" for the rate of the active action. Multiplying two rates if both actions are "active" can be understood in the context of biochemical reactions as multiplying current concentrations of reactants with each other. In PEPA the result of the multiplication is additionally multiplied with a rate denoting the "speed" of the reaction [CGH05].

It is also possible just to assign combined rates to all sets of synchronising actions with rates (for example, saving them in a separate data structure), but this is also not the best solution, as it may violate the principle of compositionality – the main advantage of describing systems by process calculi instead of directly by labelled transition systems. We can conclude this short overview of the formulas for combining rates with a remark that it is assumed in general that there is no "best" combining function but the function used for combining rates should depend on the application area.

In IMC it is assumed instead that two synchronising actions are not executed jointly "the whole way", but rather "check" with each other. Synchronisation itself does not take any time and can be regarded as a moment in time when several subsystems have reached each a certain state which allows them all to continue their executions after the check that all processes are "ready". The synchronisation is therefore only possible for actions which take no time to execute, but not for delays which are completely independent from actions. This design choice solves in some sense a problem of combining synchronisation rates (there is no need to combine), but it also means that we have an interleaving semantics and nondeterminism as a natural feature of IMC semantics (there is

no nondeterminism in PEPA).

Nondeterminism is not only a "curse" for a system (it does lead in general to a state space explosion and to a much higher complexity of model checking), but it also creates a possibility of generalisation. In one model we are "synthesising" many models. Nondeterminism allows to account for, for example, implementation freedom (each concrete implementation can choose to implement only a subset of nondeterministic choice alternatives), scheduling freedom (a concrete scheduler can decide which alternative is chosen in any moment of time) and an external environment (some alternatives might be blocked by an environment) [Her02]. We can therefore reason about many systems by reasoning about one IMC system, while systems without nondeterminism basically represent only one system, though necessarily abstracted from some details.

Another design choice that has been made in IMC is only considering IMC systems with finite semantic models. This is an easily explainable design decision as it is obviously much easier to work with semantic models which are known to be finite. The same design choice has been made in PEPA. In [BH00] and [Her02] the ways to achieve the finiteness of semantic models are slightly different. In [BH00] it is required that the parallelisation operator is applied only to closed IMC expressions, while in [Her02] it is only allowed to be applied on the highest syntactic level (see Table 2.1) – this automatically means that it can be applied only to closed expressions but also that no choice operator can be applied "on top" of the parallelisation operator. The first variant of restricting the parallelisation operator is clearly more permissive than the second one. We have decided in favour of the first variant of restriction in the syntax of $\mathrm{IMC}^{\mathrm{G}}$ in Section 3.2 in order to limit ourselves to finite semantic models as well.

Examples of process calculi with infinite semantic models are the well-known calculi CCS [Mil80] and CSP [Hoa85], the new developments include the BioAmbients calculus [RPS+04]. In all mentioned examples there are no restrictions on "spawning" new processes in parallel to already existing ones, so there can be any number of (similar) processes running in parallel. This leads to in general an infinite state space of the labelled transition systems that are semantic models of, for example, CCS processes. In many cases, however, some of the states can be merged without information loss, and there can even exist a finite labelled transition system with the same behaviour as an "original" infinite labelled transition system (so-called *bisimulation equivalent* to it). In this work we will only analyse finite systems, but our analysis methods can be potentially extended to some of the systems with infinite semantic models, especially if there exist bisimulation equivalent to them finite systems.

Recently new developments have been taken place in the area of stochastic process calculi with a purpose of creating higher-level formalisms in which widely-

used stochastic process calculi can be understood as particular cases. This way of representation can help to compare process calculi with each other, to understand their similarities and differences, and to create new process calculi with predictable properties. For example, in [KS08] a so-called SGSOS framework has been developed for defining Markovian stochastic transition systems called *Rated Transition Systems*. This work is based on the developments from [TP97]. It was shown that the operational semantics of, for example, PEPA [Hil96] and stochastic π-calculus [Pri95] can be formulated as Rated Transition Systems. A number of further stochastic process calculi can be also defined in the SGSOS framework. An important result is that stochastic bisimilarity is guaranteed to be a congruence for Rated Transition Systems, i.e. we get a method of defining process calculi with guaranteed congruence properties.

In [NLLM09b] so-called *Rate Transition Systems* have been introduced as basic models for stochastic processes. In Rate Transition Systems there are rates associated with all state, all executable actions, and sets of possible next states. Stochastic extensions of both CCS and CSP can be formulated as Rate Transition Systems that preserve the associativity of the parallel composition. Rate Transition Systems have been subsequently generalised to *Function Transition Systems*.

IMC and CTMDP

Many stochastic process calculi (for example, PEPA) have CTMCs as their semantic models. As we have already mentioned, the situation is more complicated for IMC. As it is often pointed out in the literature, the semantics of IMC is similar to the semantics of *Continuous-Time Markov Decision Processes* (CTMDPs), but there are also some differences. CTMDPs have only one kind of transition. Each transition has an associated action name and a set of transition rates to other states. The idea is that firstly a scheduler chooses an action name among all enabled transitions and then a transition to that state takes place whose associated delay completes first.

See an example of a CTMDP in Figure 2.3. Note that we do not use any state names – they are not necessary, as a state is fully characterised by its transitions as in IMC. Basically CTMDPs can be understood as IMCs where interactive (i.e. with only interactive transitions) and Markovian states (i.e. with only delay transitions) strictly alternate, initial states are interactive, and absorbing states have only Markovian incoming transitions.

In [Joh07] the question of transforming IMC systems into CTMDPs has been thoroughly considered. Such conversion can be useful as CTMDP is a better

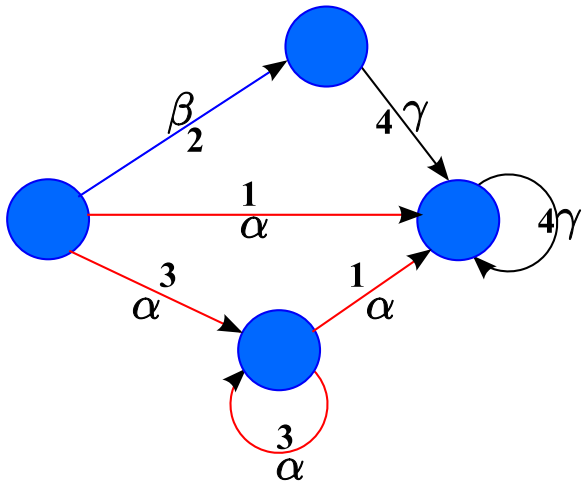

Figure 2.3 – Example of CTMDP used in [BHKH05].

studied performance model than IMC. The premise is that an IMC system is free from interactive cycles, i.e. it is not possible for a system to have an infinite path consisting of only interactive transitions. Such behaviour is considered unrealistic [Joh07] and therefore it can and should be avoided.

The transformation is done in several stages: an IMC system is converted firstly into an *Alternating* IMC (IMC with no hybrid states, i.e. states that both have at least one Markovian and at least one interactive transition). The latter is then converted into a *Markov Alternating* IMC (alternation of interactive and Markovian transitions is achieved by inserting τ-transitions), which is in turn converted into an *Interactive Alternating* IMC by "compressing" sequences of interactive transitions into single transitions. The result of the conversion is a CTMDP, which has a corresponding state for any state in the original IMC system except for those interactive and non-initial states in the original IMC system that do not have any Markovian predecessor state.

The transformation preserves the timed probabilistic behaviour [Joh07], but it is non-compositional. Transformation results for "smaller" IMCs cannot be reused for "bigger" IMCs [Joh07], because the maximal progress assumption is also applied to external actions and not only to the internal action during the transformation. This is justified by the assumption that we are transforming only "ready" or "closed" systems where the environment has already been integrated [Joh07].

2.2 Pathway Analysis

The main contribution of the thesis is applying Static Analysis methods to systems representable in a version of IMC. Static Analysis methods have been first developed for analysing programs without executing them (e.g. [NNH99]): hence the name – *static* (program code) in contrast to *dynamic* (program execution) analysis. It has been established that many interesting properties could be extracted from a program text alone.

In many cases precise analysis results are unobtainable or too expensive to compute in the sense of complexity, but acceptable "over-approximations" or "under-approximations" (also called "safe" results) can be computed instead. That means that results that we can compute from a program text "contain" in some sense the correct results, but in general also include incorrect results (over-approximation) or "contain" an in general strict subset of correct results (under-approximation). We may get therefore sometimes "false positives" but no "false negatives", or the other way round – mistakes in only one direction make Static Analysis results useful. Static Analysis methods have shown themselves so advantageous in the area of program analysis that they have been transferred to other areas as well – in particular to the analysis of systems described by process calculi.

In the following we will shortly describe one Static Analysis method – Data Flow Analysis [NNH99]. Consequently we will give an account of its application to process calculi, in particular, to CCS [Mil80] and BioAmbients [RPS$^+$04]. A version of Data Flow Analysis applied to BioAmbients has been named "Pathway Analysis" by association with "pathways", which are development sequences in biology [NNPR04]. Our own method is an adaptation of a Data Flow/Pathway Analysis to IMC systems, even though we subsequently post-process our analysis results in order to answer more complex questions than has been done before.

2.2.1 Data Flow Analysis as a program analysis method

Data Flow Analysis is a form of Static Analysis which is a well-known method of program analysis, being one of the so-called Formal Methods for analysing programs [BIM03]. The idea is to deduce some information about program executions based on the program text alone without actually executing it. The advantage is that each program text is finite while its execution can run forever. Therefore we can always check the program code for mistakes but we cannot always check the program at run-time. Moreover, all interesting, i.e. non-

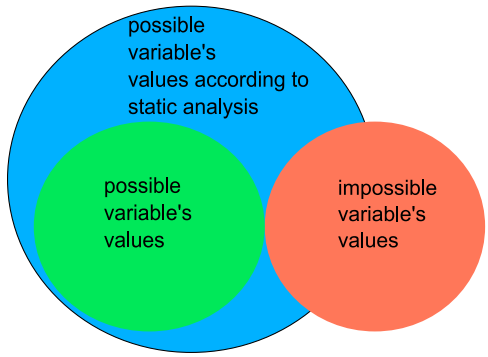

Figure 2.4 – Static Analysis and variable's values.

trivial, program properties are undecidable [HMU06], therefore methods similar to Static Analysis are needed if we would like to obtain at least "safe" if not precise answers. Static Analysis works on reachable configurations (i.e. states of memory and registers) which are finite even for infinite program executions and in general either over- or under-approximates them.

Traditionally program analysis has been used for optimising compilers. One of the purposes was to determine whether two syntactically different programs always return the same outputs on the same inputs [NNH99]. To this end Static Analysis methods have been used, for example, in order to determine variables' values or relations between values of several variables: for example, whether $x < y$ always holds for two variables x and y. See a typical relation between Static Analysis results and actual values of some variable (that can be different for different program inputs) in Figure 2.4 – in this case Static Analysis over-approximates a variable's possible values.

In the last decade Static Analysis has become a popular method for verification of security properties of programs [CM04]. Namely, Static Analysis methods are constructed in such a way that if they return "yes" then the program is secure and if they return "no" then the program might be insecure. Static Analysis thus under-approximates the security properties of the program. See the schematic presentation of this situation in Figure 2.5. If a Static Analysis method returns "no" then the programmer should debug the program with another methods or rewrite potentially unsafe parts of the program and apply Static Analysis methods again.

A broader understanding of Static Analysis methods has led to their acceptance in other domains than conventional program analysis – in particular, in the

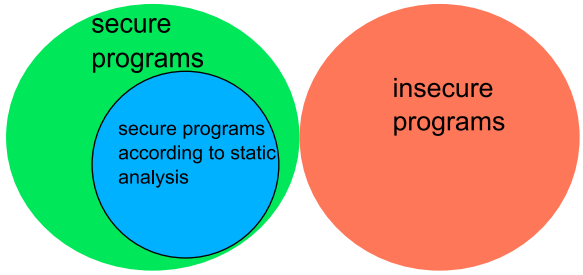

Figure 2.5 – Static Analysis and program security properties.

analysis of process calculi which we are doing in this thesis. In this section we will shortly present the Data Flow Analysis method. For other program analysis methods, e.g. Control Flow Analysis, see, for example, [NNH99].

Data Flow Analysis tracks how the data (mostly variables) changes from one program point to another one. The initial stage in Data Flow Analysis is assigning to each program block (usually each program command) a unique label. The purpose of performing Data Flow Analysis is to associate with each label an information describing executional status (e.g. the state of the memory and registers) of the program when the program counter has moved past the corresponding program block. This means that the information should correctly describe the state of the memory and the registers also in case the program block will be executed many times, infinitely often or never in a concrete program run.

Clearly it is not always possible to obtain exact information characterising a program block: for example, the block in question might be executed many times and the variables used in it might have different values during each execution. Some Static Analysis methods unfold a loop 1-2 times in order to achieve higher precision, but they do not unfold the loop until the looping condition changes for performance reasons. Another source of imprecision is that values of some variables might not be known in advance, for example, are obtained from the user input. Moreover, there can be some complex conditions in a program for execution of some program blocks which cannot be checked with Static Analysis methods, therefore it cannot be determined whether a particular program block will be executed or not.

As a consequence, a variable's possible values returned by Static Analysis are usually represented by elements of a complete lattice, with the lattice's elements representing more and more imprecise information. If two different bits of information (e.g. two different values of the same variable during two different program executions) need to be merged then their least upper (or greatest lower,

depending on the analysis purpose) bound has to be taken. It is usual to work with *Moore families* which represent the chosen steps of losing the precisions while merging several informations. See a formal definition of these concepts below. For a more detailed explanation see [NNH99].

Definition 2.2. *A partial order $(S, \leq)$ is a set S with a reflexive, transitive and antisymmetric binary relation $\leq$ defined on it.*

For a subset $Y \subseteq S$ its least upper bound $\bigsqcup Y$ *is the smallest upper bound of Y, i.e. the smallest element which is equal or bigger than any $s \in Y$. Similarly the* greatest lower bound $\bigsqcap Y$ *of Y is the greatest element which is equal or smaller than any $s \in Y$.*

A complete lattice is a non-empty partial order where each set of elements has the least upper and the greatest lower bound.

A Moore family is a subset S' of a complete lattice which is closed under the greatest lower bounds: for all $S'' \subseteq S'$ holds $\bigsqcap S'' \in S'$.

Let us have a look at the following example. If the information we are tracking is the value of the variable x, then we might have the following elements in the complete lattice, with their ordering determined by interval inclusion relation: $[0,0]$, $[1,1]$, $[-1,-1]$, $[0,1]$, $[-1,0]$, $[-1,1]$, $[0,+\infty)$, $(-\infty,0]$, $(-\infty,+\infty)$. If we only know that, for example, $3 \leq \mathsf{x} \leq 5$ then we need to write $\mathsf{x} \in [0,+\infty)$. Due to the complexity of computation we might even not be able to know anything about the value of x; then we will need to write $\mathsf{x} \in (-\infty,+\infty)$.

For the starting/finishing program points the values are usually assigned and for the rest of program points they are computed with the help of so-called *transfer functions*. Transfer functions indicate how the values in the successive/preceding program blocks depend on the values in the current block.

Let us take as an example the following program fragment:

$[\mathsf{x} := some\_function();]^{\ell_1}$

$[\mathsf{x} := \mathsf{x} + 1;]^{\ell_2}$

We have the following transfer function $\ell_1(\mathsf{x}) \to \ell_2(\mathsf{x})$:

$\ell_1(\mathsf{x}) = [0,0] \to \ell_2(\mathsf{x}) = [1,1]$

$\ell_1(\mathsf{x}) = [1,1] \to \ell_2(\mathsf{x}) = [0,+\infty]$

$\ell_1(\mathsf{x}) = [-\infty, 0] \rightarrow \ell_2(\mathsf{x}) = [-\infty, +\infty]$ etc.

For monotone transfer functions, a complete lattice of values that are assigned to program blocks and a program with all program blocks being reachable there exists the best solution, i.e. the best assignment of values from the lattice to the labels denoting program blocks [NNH99]. It is a least fixed point of the transfer functions. The least fixed point can be computed as a Kleene fixed point if the transfer functions are continuous – it is equal to the least upper bound of the transfer functions' applications to the least element of the complete lattice [NNH99].

2.2.2 Data Flow/Pathway Analysis of process calculi

Monotone Framework and its application to process calculi

Data Flow analysis has been applied to process calculi, in particular to CCS calculus (see [Hoa85]), for the first time in [NN07]. Until then Data Flow Analysis has mainly been applied to imperative languages and the usual Static Analysis methods for analysing process calculi had been type systems [NN99] and Control Flow Analysis [BDNN98], [NRNPdR07]. There is an obvious difference between imperative programming languages and process calculi, namely, there is no clear distinction between data and commands in the latter. Some adjustments therefore were needed in order to transfer Data Flow Analysis techniques to process calculi.

The idea is to generalise Data Flow Analysis of programs to the *Monotone Framework*, which can often be simplified to a *bit vector* framework [NN07]. In this framework the information can be described in a binary way, i.e. some elements are present or absent in the current state, and all transfer functions remove or add some elements to the information about the current state, independently of other present or absent elements. It is easy to see that such transfer functions are monotone, and if the state information is described by a complete lattice, then the least fix-point solution exists [NN07].

All transfer functions at the program point ℓ can thus be expressed as

$$transfer_\ell(E_\ell) = (E \backslash kill_\ell) \cup gen_\ell,$$

with E denoting analysis information at the program point ℓ, $kill_\ell$ denoting the "kill" function that removes/invalidates some elements and gen_ℓ denoting the "generate" function which adds/generates some elements. There is however an

important difference to the analysis of imperative languages: in the latter we usually know which are the possible next program points. If, for example, the only next program point is ℓ' then we would have an equation

$E_{\ell'} = \mathit{transfer}_\ell(E_\ell)$.

We do not have a notion of the "next program points" in process calculi, where data and control structures are not separated. Therefore we have to identify both the next program point(s) and the analysis information that characterises them from the current information, i.e. E and ℓ from the previous formula become the same, as well as $\mathit{transfer}_\ell(E_\ell)$ and ℓ'. This is only possible if analysis results for ℓ contain enough information to compute all possible next program points, also called next states. This is the case for the Data Flow Analysis of CCS: the analysis information contains (an over-approximation of) actions that can be "executed" in the next step and this is enough to predict all the possible next states. The number of the next states is given thus implicitly by a number of different transfer functions for the analysis information E.

In the Data Flow Analysis of CCS the generate and kill functions are computed that are the same for all states [NN07]. In short, the applicability of Data Flow Analysis to process calculi is due to the fact that we can "predict" all possible transfer functions from the current analysis information of a state and from the common generate and kill functions. In this way an (abstract) control flow graph of a system described by a particular process calculus expression can be obtained just from the initial configuration and the generate and kill functions computed on it. Generate and kill functions can be computed "once and for all" states are characteristic for all process calculi that Data Flow Analysis has been applied to until now. The method might be extendable also to generate and kill functions that change after some of the transitions in a certain way, but this has not been tried yet.

Data Flow Analysis of CCS

In this section we will shortly present the original idea of applying Data Flow Analysis to CCS process calculus (see [NN07] or [NN09] for the detailed description). We will use as a running example the CCS process

$\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C) | \bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C.$

This process first "forks" into two parallel processes, the first of which can execute either the action a or τ and then restarts with the same behaviour, and the second process executes the action $\bar{\mathsf{a}}$ and then restarts with the same

behaviour. The actions a and $\bar{\mathsf{a}}$ can only be executed together according to the semantics of CCS. See the syntax and semantics of CCS in [Mil80].

Note that in contrast to the original CCS all action names in our example got labels assigned to them. The labels make the task of defining the Data Flow Analysis on CCS expressions easier. Usually we start with a *uniquely labelled* expression in the syntax of a process calculus, i.e. all labels are different, but after several semantic steps it is not necessarily the case anymore that all the derived expressions are uniquely labelled. Our example CCS expression above is uniquely labelled, but its derivative expression

$$\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C)|\bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C|\bar{\mathsf{a}}^{\ell_3}.C|\bar{\mathsf{a}}^{\ell_3}.C$$

is not uniquely labelled anymore. However it is still *consistently labelled*, which means that each label has only one action name that it refers to.

The analysis information that describes a particular CCS expression is given by its *exposed actions*, usually denoted by their labels. Those are action/labels that are prefixes of CCS expressions that can be involved in the derivation of the transitions according to the SOS rules for CCS [Mil80]. For the running example expression

$$\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C)|\bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C$$

the exposed labels are ℓ_1, ℓ_2 and ℓ_3, while for its derivative expression

$$\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C)|\bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C|\bar{\mathsf{a}}^{\ell_3}.C|\bar{\mathsf{a}}^{\ell_3}.C$$

we have three exposed ℓ_3-labels. We can conclude that the number of exposed labels of the same kind can grow in derivative expressions. Therefore the number of exposed labels of the same kind is potentially unbounded.

The *kill* operator and the *generate* operator are applied to CCS expressions and return functions on labelled actions. They determine which actions/labels are *killed*, i.e. cease to be exposed, and which actions/labels are *generated*, i.e. become exposed, after the parameter of the function returned accordingly by the kill or the generate operator has been executed. For our example CCS expression

$$\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C)|\bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C$$

the execution of the label ℓ_1 will kill both the label ℓ_1 and its choice alternative ℓ_2 and generate the labels ℓ_1, ℓ_2 and ℓ_3. The execution of the label ℓ_2 has the

same effect. The execution of the label ℓ_3 kills the latter and generates the labels ℓ_1, ℓ_2 and ℓ_3.

The analysis results of the syntax of the CCS expression running example, i.e. its exposed labels and the results of the generate and kill operators, can be used by the Worklist Algorithm from [NN07] in order to construct a deterministic finite automaton that can simulate the semantics of the analysed CCS expression. This is possible due to the following considerations: first, any exposed label is executable (*enabled* according to the terminology from [NN07]) if it either refers to the τ-action or otherwise if there is another label which is also enabled that refers to the complementary action name (for example, a and $\bar{\mathsf{a}}$ are complementary). An issue with one of the labels being hidden and therefore not available for synchronisation does not arise, as CCS processes with action names with simultaneous global and local scopes are not allowed for the analysis [NN09]. Another requirement is that two complementary labels come from two parallel processes – such labels are called *compatible*. In order to determine whether two labels are compatible, an additional operator *comp* has been defined to keep track of pairs of compatible labels [NN09]. We will continue this line of thought in our own Pathway Analysis by introducing the *chains* operator in Section 3.4.4.

As it has been proved in [NN09], the generate and kill operators computed on some CCS expression correctly over- and underestimate accordingly generated and killed labels on all the derivatives of the CCS expression. Altogether, we get a safe estimate (i.e. an over-approximation) of exposed labels and can therefore simulate the semantics of the analysed CCS expression. In order to merge some of the states created by the Worklist Algorithm, the *widening operator* (for merging the analysis information on exposed labels) and the *granularity function* (for merging states of the created deterministic finite automaton) have been introduced in [NN07]. Merging some of the states is beneficial for obtaining deterministic finite automata with smaller state spaces and is unavoidable for CCS expressions with infinite semantic models.

See the automaton created by the Worklist Algorithm for our example CCS process in Figure 2.6. Every transition is decorated by the labels that are "executed" during the transition, i.e. participate in the derivation of the transition. From the figure it is clear that we have used the granularity function that merges all label's multiplicities larger than 1, i.e. we can have only 0, 1 or an infinite number of label occurrences. The widening operator can be defined accordingly: if one of the operands has the multiplicity of some label ℓ bigger than 1, then the result of the application of the widening operator will map ℓ to ∞. The number of states in the automaton is less than the maximal possible, which is $3^3 = 9$. If we would group together all multiplicities larger than 0, i.e. allow only 0 and infinite number of occurrences, the automaton would consist

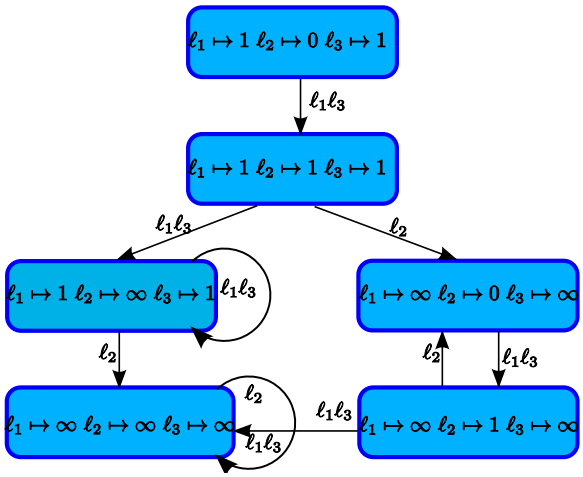

Figure 2.6 – Finite deterministic automaton simulating the semantics of the CCS process $\mathsf{let}\ C = (\mathsf{a}^{\ell_1}.C + \tau^{\ell_2}.C)|\bar{\mathsf{a}}^{\ell_3}.C\ \mathsf{in}\ C$.

of only two states – the initial state (characterised by the label multiplicities $\ell_1 \mapsto \infty, \ell_2 \mapsto 0, \ell_3 \mapsto \infty$) and the states with an infinite number of occurrences for all three labels.

Note that in our running example the exposed, generated and killed labels are computed correctly, i.e. they are not over-approximations of the actual exposed, generated and killed labels. This can be achieved in general if the analysed CCS expression is uniquely labelled. In case of a non-unique labelling if there are two different possibilities for the generate operator, then the least upper bound should be taken; if there are two different possibilities for the kill operator then their greatest lower bound should be taken. This ensures that the analysis results will be still correct, but possibly less precise than with the unique labelling. A non-unique labelling can be used in order to reduce the state space of the automaton constructed by the Worklist Algorithm already on the analysis stage and also to be able to choose explicitly which actions/labels to equalise in the analysis. For CCS processes with infinite semantic models the imprecision in the constructed automaton is unavoidable – the states to be merged are determined in this case by the granularity function.

Analysis extensions: Pathway Analysis and Modal Transition Systems

The original Data Flow Analysis of CCS has been applied to a number of other process calculi and has been extended in doing so in several ways. The Pathway Analysis of the BioAmbients calculus [RPS+04] extends the original Data Flow Analysis in order to be able to deal with the features of the BioAmbients calculus that are not present in CCS (see [NNPR04], [Pil07]). In BioAmbients communication over channels is possible which leads to name bindings; moreover, there is a notion of ambients that can merge, enter other ambients, etc. In the Pathway Analysis of BioAmbients an over-approximation of name bindings and of the spatial structure of ambients (which represent biological compartments) is "borrowed" from the results of the Control Flow Analysis [NRNPdR07] which has to be performed first.

Moreover, in [Pil07] the idea to determine a subclass of *well-formed* calculi expressions with the help of derivation rules has been introduced. The Pathway Analysis is guaranteed to be correct for well-formed expressions, while it is not necessarily the case for the non-well-formed ones. The advantage of derivation rules is their relative simplicity, so a calculus expression can be easily proved on well-formedness by induction on its syntax. We have used the same idea in order to limit the $\mathrm{IMC}^{\mathrm{G}}$ expression to the ones on which our own Pathway Analysis is both correct and precise (see Table 3.8 for well-formedness rules for $\mathrm{IMC}^{\mathrm{G}}$).

In [NNN07] Data Flow Analysis has been applied to the broadcast calculus bKlaim. The result of the analysis was the construction of so-called *abstract transition systems* that are finite abstractions for broadcast networks describable in bKlaim. The constructed transition systems are called "abstract" because some of the states in the concrete semantics of bKlaim processes are merged in the abstract transition systems according to the granularity function. The processes in bKlaim might have infinite semantic models therefore the use of the granularity function different from the identity, i.e. a granularity functions that assigns to one state several different informations on exposed labels, may be unavoidable.

Formulas in Action Computation Tree Logic (ACTL) [NV90] can be consequently interpreted over abstract transition systems, with the 3-valued results of the interpretations -"true", "false" and "unknown". Combining the interpretations on states to the interpretation of the whole formula occurs according to the rules of Kleene's strongest regular 3-valued logic [Kle52]. The "true" and "false" interpretations of a formula mean that the formula also evaluates to accordingly "true" or "false" on the corresponding concrete semantic model. The "unknown" result does not allow to make any conclusion concerning the

concrete semantic model.

The final result of Data Flow/Pathway Analysis of CCS, BioAmbients and bKlaim systems is building a finite deterministic automaton which could simulate an original system. In contrast to them the result of Data Flow Analysis of CCS processes in [NNN08] is constructing *modal transition systems*. These are systems which are 3-valued both in states and in transitions: we have instead of just presence / absence of an exposed action or transition the possibilities of *may*(possibly present) / *must*(necessary present) / absence(necessary absent). A modal transition system can be constructed because intervals are used during the analysis in order to store the multiplicities of exposed actions. Note that this kind of analysis combines the advantages of both over- and under-approximation of labels' multiplicities.

Abstracting a CCS system into a modal transition system allows to check not only safety properties, as in previous works, but also liveness properties. However in order to be able to create modal transition systems both over- and under-approximation of exposed actions needs to be performed. Instead of one number or infinity, the intervals are used in order to keep track of labels' multiplicities. The modal transition system can be then constructed in the following way: use must-transitions when the interval does not include 0; use may transition when the interval includes 0 but is bigger than 0, do not use any transition in case of the "0"-interval. As already mentioned, an additional operator has been introduced that determines whether two labels referring to compatible action names are in fact compatible; it is also determined whether two labels are just possibly compatible or definitely compatible, which increases the precision.

It has been proved in [NNN08] that a fragment of *Action Computation Tree Logic* (ACTL) [NV90] can be verified on modal transition systems constructed by the Worklist algorithm on CCS systems: the result is expressed in the Kleene's three-valued logic, with 0 expressing "definitely does not hold", 1 expressing "definitely holds" and 1/2 expressing "possibly holds". In the evaluation of logical formulas the difference between "may" and "must" transitions in modal transition systems is taken into account.

The Pathway Analysis for IMC that we will present in the next chapter differs in several ways from the previous work. For the first, we have to deal with a different synchronisation model. Synchronisation in CCS and BioAmbients involves only two components. Moreover, information about a second component is implicitly encoded into the action name: for example a and $\bar{\mathsf{a}}$ are synchronising with each other, b and $\bar{\mathsf{b}}$ etc. In IMC the number of synchronising actions can vary, even for the same action name. Moreover, we adopt a more permissive syntax of IMC which allows the number of synchronising actions to grow with the time. To deal with the first situation we will introduce another Pathway

Analysis operator, the "chains" operator. The second situation will be ruled out by our well-formedness condition.

Another difference of our own Pathway Analysis to the Data Flow/Pathway Analyses of process calculi which have been conducted before is that we limit ourselves to finite systems and we aim to achieve the exact modelling on the analysis stage. We do not use therefore an operator (i.e. a granularity function) for merging states. Future research can consist in extending our methods to infinite systems while using either a granularity function similar to the one in [NN07] or another methods for merging states. For example, we could merge first all bisimilar states together and then merge "more similar states" together before merging "less similar" states, depending on the number of states to which we want to reduce the automaton build by the Worklist Algorithm.

2.3 Model Checking of IMC

Model checking of IMC systems and systems similar to IMC in their expressiveness is currently a "hot topic" and a dynamically developing area of research. We will give here a short overview of the recent advances in this area. The reason for giving such an overview is that in this thesis we have looked into several questions which are usually either solved by model-checking methods (reachability, timed reachability) or can be considered as the first phase of model checking with the purpose of minimising the system that has to be verified (constructing bisimulation relations), so it might be interesting for the reader to compare our methods to the state-of-the-art. In fact, even if we do not treat the whole range of model checking problems on IMC systems with our methods, we are confident that Static Analysis techniques can be useful in answering many questions related to model checking.

Verification of a system usually starts with deciding which properties should be checked. The latter are often established by the choice of a particular *logic*. For CTMC-systems, for example, the usual choice of a logic is the *Continuous Stochastic Logic* or CSL [ASSB00]. The CSL is usually considered to be a continuous-time analogue of CTL (*Computational Tree Logic*) [CES86] or PCTL (*Probabilistic CTL*) [HJ94]. CSL contains two kinds of logical formulas: *state-formulas* and *path-formulas*. The logic is very powerful: for example, state formulas can specify intervals of probabilities of taking paths with properties expressed by path formulas. Path formulas can specify time intervals for the first transition on a path and until the states with some predefined properties (expressed by state formulas) are reached. Also steady-state probability intervals are expressible in CSL. The computation of the sets of states satisfying CSL

state formulas is relatively easy if the path formulas with probability intervals that are contained in the state formulas are checked beforehand. The latter is however not an easy task in general.

CSL has been model-checked over Continuous Time Markov Chains using the uniformisation technique in [BHHK03]. The uniformisation is performed in the following way: a constant rate e is chosen that is bigger or equal to the highest exit rate (the rate of leaving a state) in the system and each state with an exit rate, for example, r is equipped with a self-loop with the rate $e - r$. Then all the states will have the same exit rate. Consequently transition rates can be substituted by the probabilities of transitions. There are efficient iterative methods for computing transient probabilities. Finding steady-state probabilities requires the reachability analysis and solving linear equation systems.

For CTMDPs and IMC systems (that can be understood as a generalisation of CTMDPs) the situation is more complicated. There are non-deterministic choices that have to be resolved before the verification of, for example, CSL formulas. The problem that is often posed is to compute either the minimum or the maximum probabilities to reach a set of goal states in some non-empty time interval: for example, the minimum or the maximum probability to reach a state s within the time between 1 and 2 time units since the start of the system in the initial state. Non-deterministic choices are resolved by a so-called *scheduler* (also called *policy* or *adversary*, see, for example, [BK08] or [NSK09]).

Instead of having one fixed scheduler for resolving non-deterministic choices often the whole class of schedulers is considered. The most powerful class of schedulers for CTMDP systems contains *history-* and *time-dependent* schedulers. Those are schedulers that resolve non-determinism based both on the states that have been passed "on the way" to the current state from the initial state, and on the time that was spent on each of the transitions – the whole history including time is assumed to be known to the scheduler. The opposite of history-dependent schedulers are history-independent schedulers that do not know which states have been passed "on the way"; the opposite of time-dependent schedulers are time-abstract schedulers that do not know how much time has passed since the system has started in the initial state.

For globally uniform CTMDPs (with the same exit rate in each state) the CSL model checking is easier than in general case (see [BHH+09]). However IMC systems in general do not give rise to globally uniform CTMDPs. In order to transform non-uniform CTMDPs or IMC systems into uniform ones, some additional transformations, similar to the uniformisation of CTMCs which has been described above, are necessary. Uniformisation of IMC systems can be done in a compositional way: parallel composition, action hiding and strong bisimulations preserve the uniformity [Joh07]. For the parallel composition the

exit rate of all states in the resulting system $E \,{}_{||}A_{||}\, F$ is equal to the sum of the exit rate of the states in E and the exit rate of the states in F; in the other two cases the exit rates do not change.

These transformations however may lead to losing some of the information on time. For uniformised IMCs and CTMDPs that have not been uniform from the start, the model checking of CSL formulas for time- and history-dependent schedulers is not exact in general. This class of schedulers is strictly less powerful on uniformised systems, which means that the best reachability time computed on a uniformised system can actually be worse than the best reachability time achievable on a non-uniformised system [BHKH05]. The example CTMDP in Figure 2.3 has been taken from [BHKH05] where it was used in order to show that the expected time to reach the absorbing state is strictly smaller in the uniformised than in the non-uniformised system in the class of time- and history-dependent schedulers. This issue does not arise in CTMCs: computations of timed-bounded reachability on uniformised and non-uniformised CTMCs lead to the same results.

Without uniformisation the computation of the minimum or maximum probabilities of reaching any of the goal states over all time- and history-dependent schedulers requires solving an integral equation system over the minimum or maximum of functions, and this is not tractable [NZ10]. The alternative that has been proposed in [NZ10] is to discretise the time, i.e. divide the continuous time into small time slices and characterise the behaviour of an IMC system during one time slice. The generated discrete system is called in [NZ10] *Interactive Probabilistic Chains* (IPC): instead of rates it has now probabilities characterising the former Markovian transitions which are the probabilities to take the transitions within one time slice. Consequently the interval-bounded reachability problem for a constructed IPC has to be solved. It has been proved in [NZ10] that probabilities computed on IPCs converge to the probabilities in the corresponding IMCs with the diminishing of time slices. IPCs have probabilities instead of time, as their name suggests, therefore solving the reachability problems for them is easier – it is solved in [NZ10] by a modification of the value-iteration algorithm.

Besides uniformisation and discretisation, another commonly used technique in the model checking of IMC systems is abstraction. For example, after uniformisation some of the states can be merged and the probabilities of transitions into states which are to be merged can be abstracted by intervals of probabilities which contain all the abstracted probabilities [KKLW07]. Interactive transitions into states to be merged can be in their turn abstracted by *may*- and *must*-transitions [KKN09]. Approximations can be applied also during the model-checking phase: for example, in [BHKH05] the approximation algorithms for time-abstract but history-dependent schedulers are using history only up to

a certain depth.

Another kind of abstraction which can be used for CTMDPs is abstracting them as *continuous-time stochastic two-player games* (CTSTPG). Concrete states in the original systems can be partitioned into a smaller number of abstract states. The intervals for the minimal and maximal probabilities to reach a set of goal states under some time constraints can be computed on the resulting abstraction, i.e. the interval in which the actual minimal probability is contained and the interval in which the actual maximal probability is contained. The coarser the abstraction, the larger are the intervals – if the intervals are considered to be too large, the abstraction can be *refined*, i.e. a finer partition of concrete states into abstract states can be used. It is even possible to predict which abstract states to subdivide in order to achieve a quick precision improvement. For now this technique has only been implemented for MDPs and their corresponding *stochastic two-player games* (STPG) [KNP06].

Model checking is mostly done for properties related to state labellings, but for process calculi it is often more natural not to connect the checked properties to the states. The reason is that for systems describable in a particular process calculus the properties of states are defined though all the transitions that can be taken from them and through the other states reachable from them. Consequently the states usually do not have any additional atomic propositions associated with them. For such cases other kinds of temporal logics can be used, for example, ACTL, ACTL* [NV90] or ACSL [HKMKS00] logics (A stands for "action"). The logic of ACTL has been mentioned in Section 2.2.2 because a fragment of it can been verified (in the 3-valued Kleene's logic) on the modal transition systems created by Data Flow Analysis. It is possible to do a translation from ACTL to CTL and from ACSL to CSL formulas [HKMKS00] (which also involves a translation from action-labelled to state-labelled transition systems), but it often requires less effort to model-check ACTL or ACSL formulas directly.

There are also more "general" modal logics that can be interpreted over all labelled transition system – therefore also over Structural Operational Semantics of IMC systems. They do not have means to express probabilities or timed properties, but there exist on the other hand efficient algorithms for model checking of properties expressible in them. The Hennessy-Milner logic [HM85] fully characterises observational equivalence of two processes which are image finite – i.e. if each process has finitely many transitions decorated by the same action name. See Section 4.1 for a more detailed discussion of observational equivalences which are also called bisimulations. The Hennessy-Milner logic with recursion (see, for example, [Lar88]) admits recursively specified formulas and the minimal and maximal interpretations of the formulas. The Hennessy-Milner logic with recursion is very expressive and can be understood as a reformulation

of the μ-calculus [Koz83] (also called modal μ-calculus). The latter is equipped with a least fixed point operator μ and is strictly more powerful than, for example, the Hennessy-Milner logic, the temporal logics CTL [CES86], LTL [Pnu77], CTL* [EH86], etc.

Chapter 3

Pathway Analysis of IMC

3.1 General description

In this chapter we will introduce the Pathway Analysis of IMC systems which will be the basis for our developments in the rest of the thesis. We start by deciding on the syntax of IMC to be used further on. In order to simplify the proofs about the correctness and precision of our version of Pathway Analysis, we integrate the requirement that all valid IMC expressions should be guarded in order for their semantics to be well-defined directly into the syntax. Therefore we call our version of IMC syntax the *guarded* IMC or IMC$^{\mathrm{G}}$.

On the other hand, we seek to increase the compositionality of the syntax of IMC$^{\mathrm{G}}$. Therefore we have dropped the requirement that the internalisation and parallelisation operators are applied exclusively at the highest syntactic level – these operators had to be applied "on top" of the other syntactic constructs in the syntax of IMC in [BH00] and [Her02], see Section 2.1. However, allowing internalisation and parallelisation not only at the top syntactic level might lead to infinite semantic models which are not considered to be IMC systems [Her02]. We have therefore introduced additional so-called *well-formedness* conditions for excluding cases with infinite behaviour (actually, we are also ruling out some cases with finite syntactic models, for the sake of simplicity of the well-formedness conditions). Well-formed IMC$^{\mathrm{G}}$ expressions have finite semantic

models, and this means that they represent IMC systems [Her02]. On the other hand, any system written in the IMC syntax from [Her02] satisfies our definition of a well-formed IMC$^{\mathrm{G}}$ expression (apart from labels).

Another important enhancement in the syntax of IMC$^{\mathrm{G}}$ compared to the syntax of IMC is that we have added labels to all the action names and delay rates in the syntax of IMC$^{\mathrm{G}}$, similarly to the developments in [NN07] for the CCS calculus. This is done in order to differentiate between different occurrences of the same action name in the syntax. For example, in $\underline{X := \mathsf{a}.X + \mathsf{a}.\mathsf{b}.X}$ we have two occurrences of a with different behaviours. By assigning labels to them we can specify which of the occurrences is meant: for $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_3}.X}$ we could explicitly specify whether we mean a^{ℓ_1} or a^{ℓ_2}.

Moreover, semantic transitions become decorated by the labels of action names or delay rates that participate in the derivation of the transition through the prefix rule. We have adopted this feature from [NN07] as well. It is reminiscent of the transition decoration by the information on the derivation tree in [Pri95], as we are also recording transition derivation trees in a concise way, but we do not record unused information. An advantage of decorating transitions with labels is that information on labelled action names or delay rates that give rise to the transition is explicitly recorded. This is especially advantageous for especially well-behaved IMC$^{\mathrm{G}}$ systems called "IMC$^{\mathrm{G}}$ programs" (see Definition 3.4). In their semantic models different transitions (i.e. with different derivation trees) have different labels decorating them. Therefore we can deduce the effect of the transition from an expression, i.e. we can unambiguously determine E' for which $E \xrightarrow[C]{\alpha} E'$ holds from knowing E and C, with C recording the labels that have been examined during the transition derivation. Moreover, multirelations for delay transitions always consist of just one relation because all the outgoing transitions have different labels decorating them.

In Section 3.4 we develop the Pathway Analysis for IMC$^{\mathrm{G}}$. We have called our analysis the *Pathway Analysis* because it is closely related to the Pathway Analysis of BioAmbients [NNPR04, Pil07] where it was used in order to infer information on biological pathways. The Pathway Analysis in [NNPR04, Pil07] enriches Data Flow Analysis with in particular the well-formedness condition.

We adopt the main Data Flow / Pathway Analysis operators from [NN07], namely *exposed*, *generate* and *kill* operators. We introduce the additional *chains* operator, in order to deal with the synchronisation model of IMC which is multi-way synchronisation, and several auxiliary operators for convenience. We prove subsequently the correctness of the Pathway Analysis, i.e. that we can construct a finite automaton based on the Pathway Analysis results that can simulate the behaviour of the system that has been analysed – this is similar to what has been

proved in [NN07], [Pil07] and [NNN08] in relation to other calculi. Moreover, we also prove that our Pathway Analysis is precise – the finite automaton that we would build with the Worklist Algorithm from Section 3.6 is in fact bisimilar to the one induced by the semantics of the system. This means that we can explicitly choose a precision level while post-processing the analysis results – up to the exact computation.

3.2 IMC^{G} calculus

3.2.1 Syntax

When defining the syntax of the IMC^{G} calculus we assume a countable set of *external actions*, **Act**, and a distinguished *internal action*, τ, such that $\tau \notin \mathbf{Act}$. A countable set of constants, $\mathbf{Rate} \subseteq \mathbb{R}^{+}$, shall be used to describe Markovian delay rates. The fact that we are using $\mathbf{Rate} \subseteq \mathbb{R}^{+}$ instead of the whole $\mathbb{R}^{+}$ as in IMC (see Section 2.1) is not really a limitation because we will only use a finite number of rates. In the discussion below α will often range over $\mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$. Furthermore, we shall draw upon a countable set of labels, **Lab**, in order to annotate action names and delay rates that occur in processes. Finally, a countable set of process variables, also called process identifiers, **Var**, shall assist us in the definition of recursive processes.

The syntax of the calculus, which is shown in Table 3.1, comprises the following syntactic classes: *action-* (1) and *rate-guarded process variable* (2), *action-* (3) and *rate-prefixed process* (4), *sum* or *choice construct* (5), *internalisation* or *hiding construct* (6), *parallel composition* or *synchronisation construct* (7), *recursive process definition* (8), and *terminal process* (9). Internalisation and synchronisation constructs are parameterised by sets of action names from **Act**: these are namely actions that are internalised or hidden in the first case and on which to processes running in parallel synchronise in the second case.

The reason for using *guarded process variables* in the rules (1)-(2) in Table 3.1 is to ensure that process variables only occur in guarded positions, which will in particular guarantee that the semantics of IMC^{G} processes is well-defined: we are automatically excluding, for example, $\underline{X := X}$. Apart from this practical restriction, another difference from the syntax of IMC in [Her02] is that we allow the application of the internalisation and synchronisation constructs in the rules (6) and (7) in Table 3.1 not only at the highest syntactic level.

As we will see in Section 3.2.2, which describes the Structural Operational Se-

$$
\begin{array}{rcll}
P & ::= & \mathsf{a}^{\ell}.X & \mid \quad (1) \\
 & & \lambda^{\ell}.X & \mid \quad (2) \\
 & & \mathsf{a}^{\ell}.P & \mid \quad (3) \\
 & & \lambda^{\ell}.P & \mid \quad (4) \\
 & & P + P & \mid \quad (5) \\
 & & \mathsf{hide}\ A\ \mathsf{in}\ P & \mid \quad (6) \\
 & & P \,{}_{\shortparallel}A_{\shortparallel}\, P & \mid \quad (7) \\
 & & \underline{X := P} & \mid \quad (8) \\
 & & \mathbf{0} & \quad (9)
\end{array}
$$

Table 3.1 – Syntax of IMC$^{\mathrm{G}}$: $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $\lambda \in \mathbf{Rate}$, $\ell \in \mathbf{Lab}$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$.

mantics rules of IMC$^{\mathrm{G}}$, labels annotating action names and delay rates do not have a special semantic meaning, but we will make an active use of them in the definition of our analysis and in proving its correctness. As discussed earlier, synchronisation in IMC$^{\mathrm{G}}$, similarly to IMC, is defined in the CSP-style [Hoa85], i.e. any number of actions can synchronise with each other. Process definitions in IMC$^{\mathrm{G}}$, as in the IMC syntax from [BH00], perform two functions: they define process variables and represent process expressions at the same time. Therefore, for example, $\underline{X := P}$ represents X where $X = P$.

As usual, we consider only IMC$^{\mathrm{G}}$ processes with finite syntactic definitions. Therefore the actual number of action names, delay rates, variables and labels used in the description of a particular IMC$^{\mathrm{G}}$ process is always finite.

Similarly to IMC, an IMC$^{\mathrm{G}}$ expression is called *closed* if all process variables in it are *bound* or not *free*, i.e. are contained in corresponding recursive definitions complying with the syntactic rule (8) from Table 3.1. For example, $\underline{X := \mathsf{a}^{\ell}.X}$ is considered to be a closed expression, while $\mathsf{a}^{\ell_1}.X.\underline{X := \mathsf{a}^{\ell_2}.X}$ is not, because the first X is not bound. In the following we will mainly consider closed IMC$^{\mathrm{G}}$ expressions apart from the proofs where we will sometimes encounter expressions with free process variables.

We will most often assume that every occurring recursive definition in an IMC$^{\mathrm{G}}$ expression uses a different variable, because in this way we will not need to differentiate between different definitions of the same variable. If this is not the case for a particular IMC$^{\mathrm{G}}$ expression, we can easily rename process variables in order to achieve unique process variables' definitions. We will also mostly assume that IMC$^{\mathrm{G}}$ expressions are uniquely labelled, i.e. any appearance of a particular label inside the expression is unique. This property makes it possible

$$
\begin{array}{rl}
X \preceq \mathsf{a}^{\ell}.X & (1)\\
X \preceq \lambda^{\ell}.X & (2)\\
P \preceq \mathsf{a}^{\ell}.P & (3)\\
P \preceq \lambda^{\ell}.P & (4)\\
(P_1 \preceq (P_1 + P_2)) \wedge (P_2 \preceq (P_1 + P_2)) & (5)\\
P \preceq \mathsf{hide}\ A\ \mathsf{in}\ P & (6)\\
(P_1 \preceq (P_1 \ {}_{|\!|}A{}_{|\!|}\ P_2)) \wedge (P_2 \preceq (P_1 \ {}_{|\!|}A{}_{|\!|}\ P_2)) & (7)\\
P \preceq \underline{X := P} & (8)
\end{array}
$$

Table 3.2 – Subexpression relation $\preceq$ on IMC$^{\mathrm{G}}$ expressions.

$$
\begin{array}{rcll}
Labs(\mathsf{a}^{\ell}.X) & = & \{\ell\} & (1)\\
Labs(\lambda^{\ell}.X) & = & \{\ell\} & (2)\\
Labs(\mathsf{a}^{\ell}.P) & = & \{\ell\} \cup Labs(P) & (3)\\
Labs(\lambda^{\ell}.P) & = & \{\ell\} \cup Labs(P) & (4)\\
Labs(P_1 + P_2) & = & Labs(P_1) \cup Labs(P_2) & (5)\\
Labs(\mathsf{hide}\ A\ \mathsf{in}\ P) & = & Labs(P) & (6)\\
Labs(P_1 \ {}_{|\!|}A{}_{|\!|}\ P_2) & = & Labs(P_1) \cup Labs(P_2) & (7)\\
Labs(\underline{X := P}) & = & Labs(P) & (8)\\
Labs(\mathbf{0}) & = & \emptyset & (9)\\
Labs(X) & = & \emptyset & (9)
\end{array}
$$

Table 3.3 – Definition of the operator $Labs : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Act}}$.

to differentiate between two appearances of the same action name or the same delay rate: for example, in $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_3}.X}$ we can differentiate between a^{ℓ_1} and a^{ℓ_2}. This is important for obtaining the maximal precision of our analysis. If an expression is not uniquely labelled, then we can easily relabel it in a unique way.

We will often conduct proofs by induction on the syntax of IMC$^{\mathrm{G}}$ expressions. In doing so we will prove that if some property holds for all subexpressions of an expression in question then it also holds for the whole expression. Formally speaking, if a property holds for all $E' \preceq E$, then it also holds for E. The subexpression relation $\preceq$ is defined as the reflexive and transitive closure of the relation from Table 3.2.

We define in Table 3.3 the operator *Labs* that operates on IMC$^{\mathrm{G}}$ expressions and process variables, and returns a set of labels that occur inside the input expression. This operator will be useful in the definitions and proofs below.

$$
\begin{array}{rcll}
\mathfrak{M} & ::= & \mathbf{Lab} \to \mathbb{N}_0 & \\
\bot_{\mathfrak{M}}(\ell) & = & 0 & \text{for all } \ell \in \mathbf{Lab} \\
M_1 \leq M_2 & \Leftrightarrow & M_1(\ell) \leq M_2(\ell) & \text{for all } \ell \in \mathbf{Lab} \\
[M_1 + M_2](\ell) & = & M_1(\ell) + M_2(\ell) & \text{for all } \ell \in \mathbf{Lab} \\
[M_1 - M_2](\ell) & = & M_1(\ell) - M_2(\ell) & if\ M_1(\ell) \geq M_2(\ell) \\
[M_1 - M_2](\ell) & = & 0 & if\ M_1(\ell) < M_2(\ell) \\
dom(M) & = & \{\ell \in \mathbf{Lab} | M(\ell) > 0\} &
\end{array}
$$

Table 3.4 – Definition of $\mathfrak{M}$, its least element $\bot_{\mathfrak{M}}$ and the operations $\leq$, $+$, $-$ and *dom* on $\mathfrak{M}$. $M, M_1, M_2 \in \mathfrak{M}$.

3.2.2 Semantics

In this section we will present Structural Operational Semantics (SOS) of IMC$^{\mathrm{G}}$ and prove several results that will be useful in our further developments. We have adopted the Structural Operational Semantics of IMC$^{\mathrm{G}}$ from IMC in [BH00] with one exception: transitions are additionally decorated with so-called *multisets of labels* that are elements of the domain $\mathfrak{M}$. Decorating transitions with multisets will help us to prove the correctness of our analysis. The domain $\mathfrak{M}$ and several operations on it are formally introduced in Table 3.4.

We use *multisets* instead of *sets* of labels because in general there can be several labels of the same kind in the expression: for example, in $\mathsf{a}^{\ell}.\mathbf{0} \parallel_{\emptyset} \mathsf{a}^{\ell}.\mathbf{0}$ there are two ℓ labels that both can be executed in the next transition. We will show in Lemma 3.15 that this cannot happen for so-called IMC$^{\mathrm{G}}$ programs (see Definition 3.4) which are particularly "well-behaved" IMC$^{\mathrm{G}}$ expressions.

We define $\mathfrak{M}$ as a set of functions assigning each label from **Lab** a positive natural number or zero, i.e. each label has its corresponding number of occurrences. The least element of $\mathfrak{M}$ is denoted $\bot_{\mathfrak{M}}$ and is defined in a natural way, as a function that assigns zero to all the labels. The sum operation on the elements from $\mathfrak{M}$ is defined in a straightforward way, while for the subtraction we have to pay attention that labels cannot be assigned negative numbers in $\mathfrak{M}$: the smallest number of label's occurrences is zero. The domain of an element M from $\mathfrak{M}$ (returned by the function *dom*) is a set of labels that are assigned positive natural numbers in M.

The rules in Table 3.5 show how syntactic terms from IMC$^{\mathrm{G}}$ can be put into correspondence to labelled transition systems in a compositional manner. Most of the rules closely follow usual SOS rules for process calculi: for example, the rules for prefixing (1) and (10), choice (2)-(3) and (11)-(12), synchronisation (4)-

$$\mathsf{a}^{\ell}.\,E \xrightarrow[\perp_{\mathfrak{M}}\,[\ell \mapsto 1]]{\mathsf{a}} E \quad (1) \qquad (\lambda^{\ell}).\,E \xrightarrow[\perp_{\mathfrak{M}}\,[\ell \mapsto 1]]{\lambda} E \quad (10)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E'}{E + F \xrightarrow[C]{\mathsf{a}} E'} \quad (2) \qquad \frac{E \xrightarrow[C]{\lambda} E' \qquad F \overset{\tau}{\not\rightarrow}}{E + F \xrightarrow[C]{\lambda} E'} \quad (11)$$

$$\frac{F \xrightarrow[C]{\mathsf{a}} F'}{E + F \xrightarrow[C]{\mathsf{a}} F'} \quad (3) \qquad \frac{F \xrightarrow[C]{\lambda} F' \qquad E \overset{\tau}{\not\rightarrow}}{E + F \xrightarrow[C]{\lambda} F'} \quad (12)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \qquad \mathsf{a} \notin A}{E \,{}_{\shortparallel}A_{\shortparallel}\, F \xrightarrow[C]{\mathsf{a}} E' \,{}_{\shortparallel}A_{\shortparallel}\, F} \quad (4) \qquad \frac{E \xrightarrow[C]{\lambda} E' \qquad F \overset{\tau}{\not\rightarrow}}{E \,{}_{\shortparallel}A_{\shortparallel}\, F \xrightarrow[C]{\lambda} E' \,{}_{\shortparallel}A_{\shortparallel}\, F} \quad (13)$$

$$\frac{F \xrightarrow[C]{\mathsf{a}} F' \qquad \mathsf{a} \notin A}{E \,{}_{\shortparallel}A_{\shortparallel}\, F \xrightarrow[C]{\mathsf{a}} E \,{}_{\shortparallel}A_{\shortparallel}\, F'} \quad (5) \qquad \frac{F \xrightarrow[C]{\lambda} F' \qquad E \overset{\tau}{\not\rightarrow}}{E \,{}_{\shortparallel}A_{\shortparallel}\, F \xrightarrow[C]{\lambda} E \,{}_{\shortparallel}A_{\shortparallel}\, F'} \quad (14)$$

$$\frac{E \xrightarrow[C_1]{\mathsf{a}} E' \qquad F \xrightarrow[C_2]{\mathsf{a}} F' \qquad \mathsf{a} \in A}{E \,{}_{\shortparallel}A_{\shortparallel}\, F \xrightarrow[C_1 + C_2]{\mathsf{a}} E' \,{}_{\shortparallel}A_{\shortparallel}\, F'} \quad (6)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \qquad \mathsf{a} \notin A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\mathsf{a}} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (7) \qquad \frac{E \xrightarrow[C]{\lambda} E'}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\lambda} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (15)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \qquad \mathsf{a} \in A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\tau} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (8)$$

$$\frac{E\{\underline{X := E}/X\} \xrightarrow[C]{\mathsf{a}} E'}{\underline{X := E} \xrightarrow[C]{\mathsf{a}} E'} \quad (9) \qquad \frac{E\{\underline{X := E}/X\} \xrightarrow[C]{\lambda} E'}{\underline{X := E} \xrightarrow[C]{\lambda} E'} \quad (16)$$

Table 3.5 – Structural Operational Semantics of IMC$^{\mathrm{G}}$: $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $C \in \mathfrak{M}$, $\lambda \in \mathbf{Rate}$, $\ell \in \mathbf{Lab}$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$.

(6) and (13)-(14), hiding (7)-(8) and (15), and recursion unfolding (9) and (16). If several labelled actions are involved in the derivation of a transition, then all the labels are added to the multiset decorating the transition – see rule (6). The term $E\{\underline{X := E}/X\}$ in rules (9) and (16) denotes an expression E with every free occurrence of the variable X in it substituted by the expression $\underline{X := E}$. In the following we may refer to a labelled action name or delay rate, a label or a multiset of labels as "executable" or "executed" if they occur in the rules that are used in the derivation of respectively some executable or executed transition.

Similarly to IMC, actions are executed by IMC$^{\mathrm{G}}$ processes instantaneously, while rates denote exponentially distributed waiting time. Delay rates cannot synchronise and they cannot be internalised. In case several action-transitions are possible, the choice between them occurs nondeterministically, while the choice between several enabled Markovian transitions is probabilistic and is subject to a probabilistic race (see Section 2.1). The Structural Operational Semantics of IMC$^{\mathrm{G}}$ is defined as a multirelation for delay transitions, similarly to IMC. See the definition of SOS for IMC in [BH00] for more details.

Similarly to IMC, the choice between action-transitions and Markovian transitions if both are possible is made nondeterministically except for internal actions: if an IMC$^{\mathrm{G}}$ process has something internal to do, it will win over waiting for some time period. In order to describe this situation we have conditions respectively $F \stackrel{\tau}{\not\rightarrow}$ and $E \stackrel{\tau}{\not\rightarrow}$ for Markovian transitions of the choice and synchronisation constructs in the rules (11)-(14). The denotation $E \stackrel{\tau}{\not\rightarrow}$ means that there is no multiset of labels C and no IMC$^{\mathrm{G}}$ expression E' such that there is a transition $E \xrightarrow[C]{\tau} E'$.

Two IMC$^{\mathrm{G}}$ expressions E and E' are connected by a transition relation if $E \xrightarrow[C]{\alpha} E'$ can be derived from the rules in Table 3.5 for some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and $C \in \mathfrak{M}$. We call a transition $E \xrightarrow[C]{\alpha} E'$ "enabled" or "executable" for E. We may also say that we have obtained E' from E after the execution of the transition $\xrightarrow[C]{\alpha}$. We will often call E' a *derivative expression* or just a *derivation* of E. *Labelled Transition System* (LTS) is constructed for an IMC$^{\mathrm{G}}$ expression E by registering all possible transition relations for E and its derivative expressions.

We write $E \xrightarrow{\alpha} E'$ if there exists some $C \in \mathfrak{M}$ such that a transition $E \xrightarrow[C]{\alpha} E'$ is derivable for E from the rules in Table 3.5. We write $E \underset{C}{\longrightarrow} E'$ if there exists some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ such that a transition $E \xrightarrow[C]{\alpha} E'$ is derivable for E from the rules in Table 3.5. Finally, we write $E \longrightarrow E'$ if there exist some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and some $C \in \mathfrak{M}$ such that a transition $E \xrightarrow[C]{\alpha} E'$ is derivable for E from the rules in Table 3.5. We write $E \xrightarrow{*} E'$ if E' is in the reflexive and transitive closure of the relations $\xrightarrow[C]{\alpha}$ for all $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and $C \in \mathfrak{M}$. If $E \xrightarrow{*} E'$ then we can call E' a *derivative expression* or a *derivation* of E similarly to the case $E \longrightarrow E'$. We will also talk about a *sequence* of transitions $E \xrightarrow[C_1]{\alpha_1} ... \xrightarrow[C_n]{\alpha_n} E'$ or a *path* from E to E' while explicitly referring to a particular transition sequence between E and E'.

We will prove now two useful lemmas concerning the SOS rules of $\mathrm{IMC}^{\mathrm{G}}$. Lemma 3.1 asserts a rather evident fact that closed $\mathrm{IMC}^{\mathrm{G}}$ expressions can give rise only to closed $\mathrm{IMC}^{\mathrm{G}}$ expressions as a result of any number of semantic transitions.

Lemma 3.1 (Preservation of $\mathrm{IMC}^{\mathrm{G}}$ syntax)**.** *Given a closed* $\mathrm{IMC}^{\mathrm{G}}$ *expression* E *and* $E \longrightarrow E'$*, then* E' *is also a closed* $\mathrm{IMC}^{\mathrm{G}}$ *expression.*

Proof. See Appendix A. □

Lemma 3.2 proves a useful fact about recursion unfoldings in the rules (9) and (16) in Table 3.5. We will show that we can perform substitutions in these rules also after the transition instead of performing it before the transition.

Lemma 3.2 (Transition from process definition)**.** *Given a closed* $\mathrm{IMC}^{\mathrm{G}}$ *expression* $\underline{X := E}$*, then* $\underline{X := E} \xrightarrow[C]{\alpha} E'$ *if and only if there exists* E'' *such that* $E \xrightarrow[C]{\alpha} E''$ *and* $E' = E''\{\underline{X := E}/X\}$.

Proof. See Appendix A. □

The result proved in Lemma 3.2 will make it easier for us to derive facts concerning an $\mathrm{IMC}^{\mathrm{G}}$ expression E' from already known properties of another $\mathrm{IMC}^{\mathrm{G}}$ expression F by induction on the structure of F provided $F \xrightarrow[C]{\alpha} E'$. In case $F = \underline{X := E} \xrightarrow[C]{\alpha} E'$ we could namely derive from $E \xrightarrow[C]{\alpha} E''$ some properties of E'' and adapt them to $E''\{\underline{X := E}/X\}$.

3.3 Well-formedness

In this section we will present conditions under which an $\mathrm{IMC}^{\mathrm{G}}$ expression will be considered "well-formed". Our purpose is that for $\mathrm{IMC}^{\mathrm{G}}$ expressions that comply with these conditions their properties that are relevant for the Pathway Analysis stay invariant under any number of transitions. We have got

$$
\begin{array}{rcll}
\mathbf{fn}(\mathsf{a}^{\ell}.X) & = & \{\mathsf{a}\} & (1)\\
\mathbf{fn}(\lambda^{\ell}.X) & = & \emptyset & (2)\\
\mathbf{fn}(\mathsf{a}^{\ell}.P) & = & \{\mathsf{a}\} \cup \mathbf{fn}(P) & (3)\\
\mathbf{fn}(\lambda^{\ell}.P) & = & \mathbf{fn}(P) & (4)\\
\mathbf{fn}(P_1 + P_2) & = & \mathbf{fn}(P_1) \cup \mathbf{fn}(P_2) & (5)\\
\mathbf{fn}(\mathsf{hide}\ A\ \mathsf{in}\ P) & = & \mathbf{fn}(P) \backslash A & (6)\\
\mathbf{fn}(P_1 \between A \between P_2) & = & \mathbf{fn}(P_1) \cup \mathbf{fn}(P_2) & (7)\\
\mathbf{fn}(\underline{X := P}) & = & \mathbf{fn}(P) & (8)\\
\mathbf{fn}(\mathbf{0}) & = & \emptyset & (9)
\end{array}
$$

Table 3.6 – Rules for the operator $\mathbf{fn} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Act}}$ returning free (i.e. non-internalised) names of $\mathrm{IMC}^{\mathrm{G}}$ expressions.

an inspiration for the well-formedness rules for $\mathrm{IMC}^{\mathrm{G}}$ calculus from [Pil07] where well-formedness conditions have been devised for the BioAmbients calculus. Our rules are however simpler and easier to understand.

Before presenting the well-formedness conditions we will introduce two auxiliary operators on $\mathrm{IMC}^{\mathrm{G}}$ expressions. We will make use of them in the definition of well-formedness. The operator $\mathbf{fn}$ captures the notion of so-called *free names* of $\mathrm{IMC}^{\mathrm{G}}$ expressions. These are action names that appear in $\mathrm{IMC}^{\mathrm{G}}$ expressions following the syntactic rules (1) and (3) in Table 3.1 and have not been internalised, i.e. are not contained in any set A of the hide-construct from rule (6) in Table 3.1 applied on top of the action name appearance. Internalised action names are deleted from the set returned by $\mathbf{fn}$ in rule (6) of Table 3.6. For example, $\mathbf{fn}(\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_1}.\mathbf{0}) = \emptyset$ but $\mathbf{fn}(\mathsf{a}^{\ell_1}.\mathbf{0}) = \{\mathsf{a}\}$.

Free and hidden action names differ in two aspects. The first is that free action names decorate corresponding semantic transitions in an unchanged way, while hidden names decorate corresponding transitions with τ: for example, $\mathsf{a}^{\ell_1}.\mathbf{0} \xrightarrow[\perp_{\mathfrak{M}}\,[\ell_1 \mapsto 1]]{\mathsf{a}} \mathbf{0}$, however $\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_1}.\mathbf{0} \xrightarrow[\perp_{\mathfrak{M}}\,[\ell_1 \mapsto 1]]{\tau} \mathbf{0}$. The second difference is that free actions can synchronise with each other, while internalised actions do not participate in the synchronisation anymore. For example, there is a transition decorated by the action a for the $\mathrm{IMC}^{\mathrm{G}}$ process $\mathsf{a}^{\ell_1}.\mathbf{0} \between \{\mathsf{a}\} \between \mathsf{a}^{\ell_2}.\mathbf{0}$, but not for the process $(\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_1}.\mathbf{0}) \between \{\mathsf{a}\} \between \mathsf{a}^{\ell_2}.\mathbf{0}$, because in the second case the action a is internalised in the process on the left.

The inductive definition of the operator $\mathbf{fn} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Act}}$ is presented in Table 3.6. We are basically collecting all action names that we encounter while parsing an $\mathrm{IMC}^{\mathrm{G}}$ expression and throw at the same time those of them away for which we encounter the hide-construct.

$$
\begin{array}{rcll}
\mathbf{fpi}(\mathsf{a}^{\ell}.X) & = & \{X\} & (1)\\
\mathbf{fpi}(\lambda^{\ell}.X) & = & \{X\} & (2)\\
\mathbf{fpi}(\mathsf{a}^{\ell}.P) & = & \mathbf{fpi}(P) & (3)\\
\mathbf{fpi}(\lambda^{\ell}.P) & = & \mathbf{fpi}(P) & (4)\\
\mathbf{fpi}(P_1 + P_2) & = & \mathbf{fpi}(P_1) \cup \mathbf{fpi}(P_2) & (5)\\
\mathbf{fpi}(\mathsf{hide}\ A\ \mathsf{in}\ P) & = & \mathbf{fpi}(P) & (6)\\
\mathbf{fpi}(P_1 \ {}_{|\!|}A{}_{|\!|}\ P_2) & = & \mathbf{fpi}(P_1) \cup \mathbf{fpi}(P_2) & (7)\\
\mathbf{fpi}(\underline{X := P}) & = & \mathbf{fpi}(P) \backslash \{X\} & (8)\\
\mathbf{fpi}(\mathbf{0}) & = & \emptyset & (9)
\end{array}
$$

Table 3.7 – Rules for the operator $\mathbf{fpi} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Var}}$ returning free process identifiers in $\mathrm{IMC}^{\mathrm{G}}$ expressions.

Another auxiliary operator that we will use in the well-formedness rules is the *free process identifiers* operator **fpi**. It returns for a given $\mathrm{IMC}^{\mathrm{G}}$ expression a set of process variables that occur in it unbound. For example, $\mathbf{fpi}(\mathsf{a}^{\ell}.X) = \{X\}$ but $\mathbf{fpi}(\underline{X := \mathsf{a}^{\ell}.X}) = \emptyset$. The rules for the operator $\mathbf{fpi} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Var}}$ are presented in Table 3.7 and are self-explanatory. Bound process variables are deleted from the set of process variables returned by **fpi** in rule (8) in Table 3.7. Obviously an $\mathrm{IMC}^{\mathrm{G}}$ expression P is closed if and only if $\mathbf{fpi}(P) = \emptyset$.

The well-formedness rules are listed in Table 3.8. We characterise an $\mathrm{IMC}^{\mathrm{G}}$ process P as well-formed relative to a set of action names $S \subseteq \mathbf{Act}$ if $\vdash_S P$ can be derived from the rules in Table 3.8. In the following, in case we will call an $\mathrm{IMC}^{\mathrm{G}}$ expression P well-formed without explicitly specifying the corresponding action set, we will assume the latter to be equal to $\mathbf{fn}(P)$, i.e. P will be called well-formed if $\vdash_{\mathbf{fn}(P)} P$ holds.

Generally speaking, an $\mathrm{IMC}^{\mathrm{G}}$ expression P is considered to be well-formed relative to an action set S if the following conditions are fulfilled:

- there is no action name from S that occurs internalised in some subexpression of P (rule (6) in Table 3.8);
- the parallel composition can be applied only to closed subexpressions of P (the side condition to rule (7) in Table 3.8);
- moreover, free action names of a subexpression of P to which the parallel composition has been applied do not occur internalised in that subexpression (rule (7) in Table 3.8).

$$\vdash_S \mathsf{a}^{\ell}.X \tag{1}$$

$$\vdash_S \lambda^{\ell}.X \tag{2}$$

$$\frac{\vdash_S P}{\vdash_S \mathsf{a}^{\ell}.P} \tag{3}$$

$$\frac{\vdash_S P}{\vdash_S \lambda^{\ell}.P} \tag{4}$$

$$\frac{\vdash_S P_1 \qquad \vdash_S P_2}{\vdash_S P_1 + P_2} \tag{5}$$

$$\frac{\vdash_S P}{\vdash_S \mathsf{hide}\ A\ \mathsf{in}\ P}\ \textit{if}\quad A \cap S = \emptyset \tag{6}$$

$$\frac{\vdash_{S \cup \mathbf{fn}(P_1)} P_1 \qquad \vdash_{S \cup \mathbf{fn}(P_2)} P_2}{\vdash_S P_1 \ {}_{\shortparallel}A_{\shortparallel}\ P_2}\ \textit{if}\quad (\mathbf{fpi}(P_1) = \emptyset) \wedge (\mathbf{fpi}(P_2) = \emptyset) \tag{7}$$

$$\frac{\vdash_S P}{\vdash_S \underline{X := P}} \tag{8}$$

$$\vdash_S X \tag{9}$$

$$\vdash_S \mathbf{0} \tag{10}$$

Table 3.8 – Well-formedness rules for IMC$^{\mathrm{G}}$ expressions: $S \subseteq \mathbf{Act}$.

By checking whether $\vdash_{\mathbf{fn}(P)} P$ holds for a closed IMC$^{\mathrm{G}}$ expression P we are excluding cases where the same action name appears both free and hidden in P: this might lead to situations where initially non-internalised actions will become internalised after a number of semantic steps, due to the recursion unfolding. Moreover, the number of recursion unfoldings until the action name is internalised might be non-deterministically determined which basically means that the semantic model of such IMC$^{\mathrm{G}}$ expression is infinite. For example, two different transitions are derivable for $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X}$:

$$\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \xrightarrow[\bot_{\mathfrak{M}}\,[\ell_1 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X}$$

and

$$\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \xrightarrow[\bot_{\mathfrak{M}}\,[\ell_2 \mapsto 1]]{\tau} \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X}.$$

In the first case the action a^{ℓ_1} remains free in the right-hand side expression and

in the second case it becomes hidden. The action a^{ℓ_1} might be executed both as a and as τ in the next semantic step. In order to rule out such situations we consider such IMC$^{\mathrm{G}}$ expressions to be not well-formed because the condition of rule (6) in Table 3.8 is not satisfied.

It appears, however, that in order to make the semantic behaviour of IMC$^{\mathrm{G}}$ expressions "predictable" it is not enough to only rule out expressions where globally free action names might be subsequently hidden. Consider the following IMC$^{\mathrm{G}}$ transitions:

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}) \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1,\, \ell_3 \mapsto 1]]{\tau}$$

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \parallel \{\mathsf{a}\} \parallel \mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.\underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y})$$

and

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}) \xrightarrow[\perp_{\mathfrak{M}}[\ell_2 \mapsto 1]]{\tau}$$

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X}) \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}).$$

The action a is globally hidden in all the expressions above. However as a appears "locally" both free and hidden inside a synchronising process, it will leave us with a nondeterministic choice whether an a-transition decorated with the multiset $\perp_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_4 \mapsto 1]$ will be possible in the next step or not. If it is (the first case) then the action b will be executed consequently. Otherwise (the second case) b will never be executed. The initial IMC$^{\mathrm{G}}$ expression is considered to be not well-formed because the condition of rule (7) in Table 3.8 is not satisfied.

Finally, the reason for excluding synchronising IMC$^{\mathrm{G}}$ processes with free process identifiers (the side condition of rule (7) in Table 3.8) is the following: such expressions can potentially "grow" by means of creating several copies of some original IMC$^{\mathrm{G}}$ subexpression after recursion unfoldings. Consider the following example:

$$\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_2}.Y} \xrightarrow{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X} \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_2}.Y} \parallel \{\mathsf{a}\} \parallel \underline{Y := \mathsf{a}^{\ell_2}.Y}.$$

We are excluding such cases from being considered well-formed because the growth can be both non-deterministic and unbounded, i.e. an unbounded number of copies of some IMC$^{\mathrm{G}}$ process can be created which can mean that the semantic model of such IMC$^{\mathrm{G}}$ expression might be infinite. We do not analyse

such expressions with our Pathway Analysis.

We consider process identifiers and terminal processes to be well-formed (rules (9) and (10) in Table 3.8). The first may be somewhat unexpected but we have introduced rule (9) solely for technical convenience in the proofs below. Note that in spite of rule (9) in Table 3.8 a process variable is not considered a well-formed $\mathrm{IMC}^{\mathrm{G}}$ expression because it is not a valid $\mathrm{IMC}^{\mathrm{G}}$ expression according to the syntax rules in Table 3.1. We will be a bit sloppy in the definition of Pathway Analysis operators below in Section 3.4, by saying that they take as an input $\mathrm{IMC}^{\mathrm{G}}$ expressions, while they will also be defined on variables. The reason is, similarly to the well-formedness rules, that it is convenient to define the operators inductively on the syntax of $\mathrm{IMC}^{\mathrm{G}}$ by defining them also on process variables.

Note the symmetry in the well-formedness conditions on the "global level" of the whole $\mathrm{IMC}^{\mathrm{G}}$ expression and on the "local level" of its synchronising subprocesses: we require that in both cases we do not have any free process identifiers and do not hide free names. These two conditions are directly stated for synchronising subexpressions in rule (7) from Table 3.8, while for the whole expression we have to check them explicitly, by including the set of globally free action names in the set S and by considering closed $\mathrm{IMC}^{\mathrm{G}}$ expressions.

We will prove in Lemma 3.3 that any well-formed and closed $\mathrm{IMC}^{\mathrm{G}}$ expression remains well-formed after any number of semantic steps. This statement is not only useful for our further proofs but also shows the stability of the well-formedness concept under semantic transitions which assures us in its non-transient nature.

Lemma 3.3 (Preservation of well-formedness)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *expression* E *such that* $\vdash_{\mathbf{fn}(E)} E$ *holds and* $Labs(E_1) \cap Labs(E_2) = \emptyset$ *for all* $E_1 \,{}_{\shortparallel}A_{\shortparallel}\, E_2 \preceq E$*, then for all* E' *such that* $E \xrightarrow[C]{\alpha} E'$ *also* $\vdash_{\mathbf{fn}(E')} E'$ *holds and* $Labs(E_1) \cap Labs(E_2) = \emptyset$ *for all* $E_1 \,{}_{\shortparallel}A_{\shortparallel}\, E_2 \preceq E'$*.*

Proof. See Appendix A. □

Derivatives of an initially uniquely labelled $\mathrm{IMC}^{\mathrm{G}}$ expression are not necessarily uniquely labelled themselves. For example, the uniquely labelled expression $\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.X)}$ gives rise after one semantic transition to the non-uniquely labelled $\mathsf{b}^{\ell_2}.\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.X)} + \mathsf{c}^{\ell_3}.\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.X)}$.

From Lemma 3.3 follows however that we can deduce some guarantees concerning labelling of well-formed expressions: in particular, labels of synchronising processes will still be disjoint after any number of semantic steps.

We will define now "well-behaved" IMC$^{\mathrm{G}}$ expressions that we will call IMC$^{\mathrm{G}}$ *programs*. Most of our results in the rest of the chapter and in the following chapters refer to IMC$^{\mathrm{G}}$ programs and their derivative expressions.

Definition 3.4 (IMC$^{\mathrm{G}}$ programs)**.** *An* IMC$^{\mathrm{G}}$ *expression F which is well-formed (relative to the set of action names* $\mathbf{fn}(F)$*), uniquely labelled, closed and contains unique process identifier definitions is called an* IMC$^{\mathrm{G}}$ program.

3.4 Pathway Analysis

In this section we will introduce the Pathway Analysis for IMC$^{\mathrm{G}}$. As we have explained in Section 2.2.2, the purpose of conducting Data Flow / Pathway Analysis in the previous work (see [NN07], [NNPR04], [Pil07], [NNN07]) was to safely over-approximate the, possibly infinite, state space of the process defined by a given process calculus expression by a finite automaton. Our goal in doing the Pathway Analysis of IMC$^{\mathrm{G}}$ was slightly different. The semantic models of IMC$^{\mathrm{G}}$, as well as IMC, are always finite – contrary to CCS, BioAmbients and bKlaim. Therefore we had as an aim to conduct the Pathway Analysis of IMC$^{\mathrm{G}}$ in such a way that we can build consequently the finite automaton which does not only over-approximate the state space of the IMC$^{\mathrm{G}}$ system but the approximation is as small as possible.

The advantage of a precise analysis is that we have better control over the merging of states if we decide to do so while building the finite automaton. We can also relax the precision requirement on the analysis stage by assigning non-unique labels to the algebraic process description. We can use the analysis results for post-processing (see Chapters 4 and 5) – here the precision of the analysis can be handy in proving that the computed properties indeed hold. In particular, we can make conclusions not only about safety properties, as it is usual for Data Flow / Pathway Analysis ("nothing bad will ever happen"), but also about liveness properties ("something good will eventually happen") – this is possible because the approximation of the execution scenarios is tight.

While extending Pathway Analysis methods to IMC$^{\mathrm{G}}$ we had two major tasks: first, to adapt Pathway Analysis to the synchronisation model of IMC$^{\mathrm{G}}$ that is different from CCS; second, to be able to construct after conducting the

analysis a labelled transition system which will be strongly bisimilar to the one induced by the semantics of IMC^G. Similarly to [Pil07], we apply the Pathway Analysis to a subclass of IMC^G on which our analysis can be guaranteed to deliver correct results: to IMC^G programs. The previous applications of Data Flow / Pathway Analysis in [NN07], [NNPR04], [Pil07] and [NNN07] were using in their correctness proofs the fact that monotone functions on complete lattices always have the least fixed points. As we need to prove that our analysis is not only correct but also precise, we had to change some definitions and proofs in a considerable way.

We have not only achieved that the analysis is precise, but there is another positive effect as well. We could namely renounce a granularity function which has been necessary in the previous work (see Section 2.2.2) in order to guarantee the termination of the Worklist Algorithm. The negative side effect of the application of granularity functions was that, for example, model checking could not be made "on the fly", during construction of the finite automaton by the Worklist Algorithm. There was a possibility that some states would later be merged by the Worklist Algorithm (if the granularity function would map them to the same state) and so previously calculated model-checking results would become obsolete. This is not a problem anymore with our method, because we have no granularity functions, therefore states cannot be merged and each state is visited only once by the Worklist Algorithm. Accordingly we can develop in the future some "on the fly" verifications of interesting system properties.

3.4.1 Pathway Analysis operators

We have devised a number of Pathway Analysis operators that accept IMC^G expressions as input. The theoretical results that we have proved about these operators apply however only to IMC^G programs. We have also devised a number of auxiliary operators – for example, an operator that saves the correspondence between labels on the one hand and action names and rates on the other hand.

We have defined the following Pathway Analysis operators on IMC^G expressions: *exposed* operator, *chains* operator, *generate* operator and *kill* operator. Three of the operators – exposed, generate and kill – are adjustments from the corresponding definitions in the previous work, in particular in [NN07] where they have been introduced. In the named work these three operators have been defined on expressions written in the CCS calculus syntax. We have additionally devised the *chains* operator, which has not been introduced in the previous work, in order to save the "synchronising structure" of the IMC^G expression and thus to be able to determine which actions synchronise with each other. The introduction of the chains operator has mainly been triggered by the dif-

ference between the synchronisation model of IMC and accordingly IMC$^{\mathrm{G}}$ and the synchronisation model of CCS. In CCS only two actions can synchronise with each other, moreover, they are explicitly denoted as two synchronisation sides. In IMC any number of actions can synchronise (multi-way synchronisation); whether a particular action participates in the synchronisation depends on the overall syntactic structure of the whole IMC expression and not on the action name itself.

We will shortly describe now the ideas behind the different operators. A more detailed account will be given below, where we will formally define each operator. The exposed operator returns a multiset of labels of the actions or delays which may be executed in the next semantic derivation step. For example, in $\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}$ both ℓ_1 and ℓ_2 are exposed. For delays we can make a clear statement: if the label of some delay is exposed, then there exists a transition, derivable through the rules of Table 3.5, decorated by the corresponding delay and label. For actions, things are more complicated because a corresponding action may participate in some synchronisation. We can however state that if there exists for some IMC$^{\mathrm{G}}$ expression a transition decorated with an action name and a multiset of labels, then all the labels from that multiset are returned by the exposed operator with the expression as an input.

The generate and kill operators take as input an IMC$^{\mathrm{G}}$ expression and return a mapping from labels to multiset of labels. Generally speaking they return for each label those labels that after the "execution" of the first label will either become newly exposed (generate operator) or cease to be exposed (kill operator). We characterise a label as having been executed in case it is included in the multiset decorating the transition which has been carried out. In the example above $\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}$ the label ℓ_1 kills both ℓ_1 and ℓ_2 but generates only ℓ_1.

If an IMC$^{\mathrm{G}}$ expression is uniquely labelled then the generate and kill operators return unambiguous results for each label. The generate and kill operators' results on the syntax of IMC$^{\mathrm{G}}$ programs are applicable to all their derivative expressions (this will be proved in Section 3.5). Namely, the generate operator on a derivative expression will return the same results for all the labels present in the derivative expression and the kill operator will return the same result if we take into account only exposed labels of the derivative expression.

The chains operator returns for an IMC$^{\mathrm{G}}$ expression a set of multisets. We will call any returned multiset a *chain* as the multiset "connects" the labels that are to be executed simultaneously. For example, the only chain for the expression $\mathsf{a}^{\ell_1}.\mathbf{0} \mid\mid a \mid\mid \mathsf{a}^{\ell_2}.\mathbf{0}$ consists of labels ℓ_1 and ℓ_2. It is clear that labels in each chain should correspond to the same action name or delay. Chains that have in their domains more than one label correspond only to action names and not to delays:

no failing synchronisation can prevent the execution of a delay. We will prove in Section 3.5 that chains computed on an IMC^{G} program are applicable to any IMC^{G} expression derivable from it.

In the following section we will give formal definitions for all the operators. After that we will prove a number of theoretical results concerning the operators' application to IMC^{G} programs and what is especially important – concerning the relations between the results returned by the operators on some IMC^{G} program and the labelled transition system induced by the semantics of that program.

3.4.2 Exposed operator

We will now formally define the exposed operator $\mathcal{E}$. It accepts an IMC^{G} expression or a process variable and returns a value from $\mathfrak{M}$ which characterises *exposed labels*, also sometimes called *exposed prefixes*. It is enough to save only exposed labels in case their correspondence to action names or delay rates can be unambiguously determined. Exposed are labels that may be executed in a transition directly derivable for the corresponding expression. For example, the exposed operator returns on $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.X}$ the multiset $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ because the following transitions are derivable for the input expression: $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.X} \xrightarrow[\bot_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.X}$ and $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.X} \xrightarrow[\bot_{\mathfrak{M}}[\ell_2 \mapsto 1]]{\mathsf{b}} \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.X}$.

The result of the application of the operator $\mathcal{E}$ is defined inductively on the syntactic structure of its input. The rules for $\mathcal{E}$ are presented in Table 3.9. Most of the rules are clear: the only exposed label of a prefixed expression is a label of the prefix (rules (1)-(4)), exposed labels of two expressions connected by the choice or synchronisation constructs include exposed labels of both expressions (rules (5) and (7)) etc. We need however to explain rule (9).

The operator $\mathcal{E}$ is parameterised with an environment $\Gamma \in 2^{\mathbf{Var} \times \mathfrak{M}}$ which is a set of mappings from process variables to multisets of labels. We make use of an environment in case we need to determine exposed labels of a process variable – in these cases we will just look up mappings for a particular variable in Γ. The operator $\mathcal{E}$ returns $\bot_{\mathfrak{M}}$ if there are no mappings for a particular process variable in Γ and returns a sum of corresponding multisets of labels if there are several such mappings.

In the following, as we will consider only those IMC^{G} expressions that have

$$
\begin{aligned}
\mathcal{E}_\Gamma[\![\mathsf{a}^\ell.X]\!] &= \bot_{\mathfrak{M}}\,[\ell \mapsto 1] && (1)\\
\mathcal{E}_\Gamma[\![\lambda^\ell.X]\!] &= \bot_{\mathfrak{M}}\,[\ell \mapsto 1] && (2)\\
\mathcal{E}_\Gamma[\![\mathsf{a}^\ell.P]\!] &= \bot_{\mathfrak{M}}\,[\ell \mapsto 1] && (3)\\
\mathcal{E}_\Gamma[\![\lambda^\ell.P]\!] &= \bot_{\mathfrak{M}}\,[\ell \mapsto 1] && (4)\\
\mathcal{E}_\Gamma[\![P_1 + P_2]\!] &= \mathcal{E}_\Gamma[\![P_1]\!] + \mathcal{E}_\Gamma[\![P_2]\!] && (5)\\
\mathcal{E}_\Gamma[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] &= \mathcal{E}_\Gamma[\![P]\!] && (6)\\
\mathcal{E}_\Gamma[\![P_1 \,{}^{\shortparallel}A^{\shortparallel}\, P_2]\!] &= \mathcal{E}_\Gamma[\![P_1]\!] + \mathcal{E}_\Gamma[\![P_2]\!] && (7)\\
\mathcal{E}_\Gamma[\![\underline{X := P}]\!] &= \mathcal{E}_\Gamma[\![P]\!] && (8)\\
\mathcal{E}_\Gamma[\![X]\!] &= \begin{cases} \bot_{\mathfrak{M}} & \text{if } (X, M) \notin \Gamma\ \forall M \in \mathfrak{M} \\ \displaystyle\sum_{(X,M)\in\Gamma} M & \text{otherwise.} \end{cases} && (9)\\
\mathcal{E}_\Gamma[\![\mathbf{0}]\!] &= \bot_{\mathfrak{M}} && (10)
\end{aligned}
$$

Table 3.9 – Definition of the exposed operator $\mathcal{E} : \mathrm{IMC}^{\mathrm{G}} \to \mathfrak{M}$, $\Gamma \in 2^{\mathbf{Var}\times\mathfrak{M}}$.

$$
\begin{aligned}
\Gamma_{\mathbf{Var}}[\![\mathsf{a}^\ell.X]\!] &= \emptyset && (1)\\
\Gamma_{\mathbf{Var}}[\![\lambda^\ell.X]\!] &= \emptyset && (2)\\
\Gamma_{\mathbf{Var}}[\![\mathsf{a}^\ell.P]\!] &= \Gamma_{\mathbf{Var}}[\![P]\!] && (3)\\
\Gamma_{\mathbf{Var}}[\![\lambda^\ell.P]\!] &= \Gamma_{\mathbf{Var}}[\![P]\!] && (4)\\
\Gamma_{\mathbf{Var}}[\![P_1 + P_2]\!] &= \Gamma_{\mathbf{Var}}[\![P_1]\!] \cup \Gamma_{\mathbf{Var}}[\![P_2]\!] && (5)\\
\Gamma_{\mathbf{Var}}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] &= \Gamma_{\mathbf{Var}}[\![P]\!] && (6)\\
\Gamma_{\mathbf{Var}}[\![P_1 \,{}^{\shortparallel}A^{\shortparallel}\, P_2]\!] &= \Gamma_{\mathbf{Var}}[\![P_1]\!] \cup \Gamma_{\mathbf{Var}}[\![P_2]\!] && (7)\\
\Gamma_{\mathbf{Var}}[\![\underline{X := P}]\!] &= (X, \mathcal{E}_\emptyset[\![P]\!]) \cup \Gamma_{\mathbf{Var}}[\![P]\!] && (8)\\
\underline{\Gamma_{\mathbf{Var}}[\![X]\!]} &= \emptyset && (9)\\
\Gamma_{\mathbf{Var}}[\![\mathbf{0}]\!] &= \emptyset && (10)
\end{aligned}
$$

Table 3.10 – Definition of the operator $\Gamma_{\mathbf{Var}} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Var}\times\mathfrak{M}}$ returning exposed labels of process definitions.

unique process identifiers for all process definitions, we will only encounter cases where the environment Γ will contain a unique entry for each process variable occurring inside an $\mathrm{IMC}^{\mathrm{G}}$ expression P for which $\mathcal{E}_\Gamma[\![P]\!]$ will be computed. The environment Γ will mostly be an output of another operator $\Gamma_{\mathbf{Var}}$ on P. The rules for the operator $\Gamma_{\mathbf{Var}}$ are presented in Table 3.10. This operator collects exposed labels of all process definitions occurring in its input expression. If an input expression P is closed and contains unique process definitions, then $\Gamma_{\mathbf{Var}}[\![P]\!]$ is a union of unique mappings from variables in P to exposed labels of their corresponding process definitions.

In rule (8) of Table 3.10 we make use of an empty environment while computing exposed labels of an IMC$^{\mathrm{G}}$ expression P. This can be justified with the help of Lemma 3.5 which proves that the environment parameter is not important for computing exposed labels of an IMC$^{\mathrm{G}}$ expression (due to its guardedness), therefore we can parametrise the exposed operator in rule (8) of Table 3.10 with an empty environment for simplicity.

Lemma 3.5 (Exposed labels of IMC$^{\mathrm{G}}$ expressions)**.** *Given an* IMC$^{\mathrm{G}}$ *expression* E*, then* $\mathcal{E}_{\emptyset}[\![E]\!] = \mathcal{E}_{\Gamma}[\![E]\!]$ *holds for any environment* $\Gamma \in 2^{\boldsymbol{Var}\times\mathfrak{M}}$*.*

Proof. See Appendix A. □

We will now prove that the result of the application of the operator $\Gamma_{\mathbf{Var}}$ to an IMC$^{\mathrm{G}}$ expression is in a sense invariant under semantic transitions – which is what we would expect. In doing so we will need an additional Lemma 3.6 on variables' substitutions, which will also be useful in proving a number of theoretical results in the rest of this chapter.

Lemma 3.6 (Exposed labels under substitution)**.** *Given an* IMC$^{\mathrm{G}}$ *expression* E''*, then* $\mathcal{E}_{\Gamma}[\![E'']\!] = \mathcal{E}_{\Gamma}[\![E''\{\underline{X := E'}/X\}]\!]$ *holds for all environments* Γ *and all* IMC$^{\mathrm{G}}$ *expressions* $\underline{X := E'}$*. If* $E'' = X$ *then, provided that* $(X, \mathcal{E}_{\emptyset}[\![E']\!])$ *is a unique mapping for a variable* X *in* Γ*, i.e. for all* $(X, M) \in \Gamma$ *it holds that* $M = \mathcal{E}_{\emptyset}[\![E']\!]$*, then* $\mathcal{E}_{\Gamma}[\![E'']\!] = \mathcal{E}_{\Gamma}[\![E''\{\underline{X := E'}/X\}]\!]$ *holds as well.*

Proof. See Appendix A. □

We will now show in Lemma 3.7 that exposed labels of process definitions do not change after any number of semantic transitions: a set returned by the operator $\Gamma_{\mathbf{Var}}$ on derivative expressions of a closed E is always a subset of a set returned by the same operator on E. The application result can be strictly smaller if some recursive definitions disappear as a result of transitions. For example, for $\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{a}^{\ell_2}.Y} \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X}$, we have $\Gamma_{\mathbf{Var}}[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{a}^{\ell_2}.Y}]\!] = \{(X, \perp_{\mathfrak{M}}[\ell_1 \mapsto 1]), (Y, \perp_{\mathfrak{M}}[\ell_2 \mapsto 1])\}$ but only $\Gamma_{\mathbf{Var}}[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!] = \{(X, \perp_{\mathfrak{M}}[\ell_1 \mapsto 1])\}$.

Lemma 3.7 (Variable definitions under transitions)**.** *Given a closed* $\mathrm{IMC}^{\mathrm{G}}$ *expression* E *and a transition* $E \xrightarrow[C]{\alpha} E'$*, then* $\Gamma_{\boldsymbol{Var}}[\![E']\!] \subseteq \Gamma_{\boldsymbol{Var}}[\![E]\!]$ *holds.*

Proof. See Appendix A. □

Note that the exposed operator defines an equivalence relation on $\mathrm{IMC}^{\mathrm{G}}$ expressions: each equivalence class contains $\mathrm{IMC}^{\mathrm{G}}$ expressions with the same exposed labels. For example, $\mathcal{E}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!] = \mathcal{E}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.X}]\!]$ for any Γ. This equivalence relation does not have a particular meaning in the dimension of the whole $\mathrm{IMC}^{\mathrm{G}}$, but, as we will see later on, it is a strong bisimilarity relation when limited only to derivative expressions of a fixed $\mathrm{IMC}^{\mathrm{G}}$ program.

3.4.3 Generate and kill operators

In this section we will formally define the generate and kill operators. The generate operator $\mathcal{G}$ returns for an $\mathrm{IMC}^{\mathrm{G}}$ expression or process variable a result from $2^{\mathbf{Lab}\times\mathfrak{M}}$. It returns a set of mappings from all the labels occurring in its input expression to multisets of labels. Each multiset has in its domain those labels which will become newly exposed after the corresponding label mapped to them has been executed. For example, the set returned by the $\mathcal{G}$ operator on $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.X}$ contains $(\ell_1, \bot_{\mathfrak{M}}[\ell_2 \mapsto 1])$ as $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.X} \xrightarrow[\bot_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}}$ $\mathsf{a}^{\ell_2}.\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.X}$ and the exposed operator will return $\bot_{\mathfrak{M}}[\ell_2 \mapsto 1]$ on the derivative expression $\mathsf{a}^{\ell_2}.\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.X}$.

The rules for the generate operator are presented in Table 3.11. For prefixed expressions we map labels of prefixes to exposed labels of the residual expressions (rules (1)-(4)). For the rules (1)-(2) we will need to determine exposed labels of process variables. From the rules for the exposed operator in Table 3.9 we remember that exposed labels of a process variable are determined by corresponding mappings in an environment. We need therefore to parametrise the operator $\mathcal{G}$ by an environment $\Gamma \in 2^{\mathbf{Var}\times\mathfrak{M}}$, so that this environment can be "forwarded" to the exposed operator whenever necessary.

The rest of the rules in Table 3.11 are rather straightforward: the union of the results of the generate operator on subexpressions is taken in case we have choice, hiding, etc. constructs on the highest syntactic level. If one label occurs several times in an input expression, the operator $\mathcal{G}$ saves several mappings

$$
\begin{array}{rcll}
\mathcal{G}_\Gamma[\![\mathsf{a}^\ell.X]\!] & = & \{(\ell, \mathcal{E}_\Gamma[\![X]\!])\} & (1)\\
\mathcal{G}_\Gamma[\![\lambda^\ell.X]\!] & = & \{(\ell, \mathcal{E}_\Gamma[\![X]\!])\} & (2)\\
\mathcal{G}_\Gamma[\![\mathsf{a}^\ell.P]\!] & = & \{(\ell, \mathcal{E}_\Gamma[\![P]\!])\} \cup \mathcal{G}_\Gamma[\![P]\!] & (3)\\
\mathcal{G}_\Gamma[\![\lambda^\ell.P]\!] & = & \{(\ell, \mathcal{E}_\Gamma[\![P]\!])\} \cup \mathcal{G}_\Gamma[\![P]\!] & (4)\\
\mathcal{G}_\Gamma[\![P_1 + P_2]\!] & = & \mathcal{G}_\Gamma[\![P_1]\!] \cup \mathcal{G}_\Gamma[\![P_2]\!] & (5)\\
\mathcal{G}_\Gamma[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathcal{G}_\Gamma[\![P]\!] & (6)\\
\mathcal{G}_\Gamma[\![P_1 \,{}_{|\!|A|\!|}\, P_2]\!] & = & \mathcal{G}_\Gamma[\![P_1]\!] \cup \mathcal{G}_\Gamma[\![P_2]\!] & (7)\\
\mathcal{G}_\Gamma[\![\underline{X := P}]\!] & = & \mathcal{G}_\Gamma[\![P]\!] & (8)\\
\mathcal{G}_\Gamma[\![\underline{X}]\!] & = & \emptyset & (9)\\
\mathcal{G}_\Gamma[\![\mathbf{0}]\!] & = & \emptyset & (10)
\end{array}
$$

Table 3.11 – Definition of the generate operator $\mathcal{G} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times\mathfrak{M}}$, $\Gamma \in 2^{\mathbf{Var}\times\mathfrak{M}}$.

for it in the resulting set (rules (5) and (7) in Table 3.11). This will however never be the case for a uniquely labelled input expression. The definition of the generate operator on process variables (rule (9)) has been introduced for technical convenience and will be used in some of the proofs below.

The kill operator $\mathcal{K}$ also returns results from $2^{\mathbf{Lab}\times\mathfrak{M}}$, similar to the $\mathcal{G}$ operator. Multisets of labels mapped to a label have in their domains in this case those labels that will cease to be exposed after the corresponding label has been executed. The $\mathcal{K}$ operator will return on $\underline{X := \mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathbf{0}}$ the set $\{(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]), (\ell_2, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1])\}$ as two transitions are enabled: $\underline{X := \mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathbf{0}} \xrightarrow[\bot_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}} \mathbf{0}$ and, with the other choice option, $\underline{X := \mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathbf{0}} \xrightarrow[\bot_{\mathfrak{M}}[\ell_2 \mapsto 1]]{\mathsf{b}} \mathbf{0}$. In both cases we have $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ returned by the exposed operator on the initial expression and $\bot_{\mathfrak{M}}$ returned by the exposed operator on $\mathbf{0}$.

The definition of the $\mathcal{K}$ operator is divided into two parts for technical convenience: the kill operator $\mathcal{K}_{up}$ for the "upper" syntactic level of expressions and the kill operator $\mathcal{K}_{\overline{up}}$ for expressions with their upper syntactic level excluded. The output of the $\mathcal{K}$ operator on an $\mathrm{IMC}^{\mathrm{G}}$ expression is defined in Table 3.12 as a set union of the outputs of the operators $\mathcal{K}_{up}$ and $\mathcal{K}_{\overline{up}}$ on the same expression.

The two different kill operators $\mathcal{K}_{up}$ and $\mathcal{K}_{\overline{up}}$ have been introduced in order to deal with the choice construct: as we have already seen in the example above, for the expression $\underline{X := \mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathbf{0}}$ each executed label will kill all the exposed labels of the sum. We cannot therefore define the operator $\mathcal{K}$ on $P_1 + P_2$ in a purely inductive way. It would be a mistake, for example, to add $\mathcal{E}_\emptyset[\![P_2]\!]$ to all

$$\mathcal{K}[\![P]\!] \quad = \quad \mathcal{K}_{up}[\![P]\!] \cup \mathcal{K}_{\overline{up}}[\![P]\!]$$

Table 3.12 – Definition of the kill operator $\mathcal{K} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times\mathfrak{M}}$. See the definitions of the operators $\mathcal{K}_{up}$ and $\mathcal{K}_{\overline{up}}$ in Tables 3.13 and 3.14.

$$\begin{array}{rcll}
\mathcal{K}_{up}[\![\mathsf{a}^{\ell}.X]\!] & = & \{(\ell, \bot_{\mathfrak{M}}\,[\ell \mapsto 1])\} & (1)\\
\mathcal{K}_{up}[\![\lambda^{\ell}.X]\!] & = & \{(\ell, \bot_{\mathfrak{M}}\,[\ell \mapsto 1])\} & (2)\\
\mathcal{K}_{up}[\![\mathsf{a}^{\ell}.P]\!] & = & \{(\ell, \bot_{\mathfrak{M}}\,[\ell \mapsto 1])\} & (3)\\
\mathcal{K}_{up}[\![\lambda^{\ell}.P]\!] & = & \{(\ell, \bot_{\mathfrak{M}}\,[\ell \mapsto 1])\} & (4)\\
\mathcal{K}_{up}[\![P_1 + P_2]\!] & = & \bigcup_{(\ell,M)\in\mathcal{K}_{up}[\![P_1]\!]}\{(\ell, M + \mathcal{E}_{\emptyset}[\![P_2]\!])\}\cup & \\
 & & \bigcup_{(\ell,M)\in\mathcal{K}_{up}[\![P_2]\!]}\{(\ell, M + \mathcal{E}_{\emptyset}[\![P_1]\!])\} & (5)\\
\mathcal{K}_{up}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathcal{K}_{up}[\![P]\!] & (6)\\
\mathcal{K}_{up}[\![P_1 \,{}_{\shortparallel}A_{\shortparallel}\, P_2]\!] & = & \mathcal{K}_{up}[\![P_1]\!] \cup \mathcal{K}_{up}[\![P_2]\!] & (7)\\
\mathcal{K}_{up}[\![X := P]\!] & = & \mathcal{K}_{up}[\![P]\!] & (8)\\
\mathcal{K}_{up}[\![X]\!] & = & \emptyset & (9)\\
\mathcal{K}_{up}[\![\mathbf{0}]\!] & = & \emptyset & (10)
\end{array}$$

Table 3.13 – Definition of the operator $\mathcal{K}_{up} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times\mathfrak{M}}$ returning "killed" labels on the upper syntactic level.

M such that $(\ell, M) \in \mathcal{K}[\![P_1]\!]$ and $\mathcal{E}_{\emptyset}[\![P_1]\!](\ell) \geq 1$: we should take into account the possibility of several occurrences of the same label ℓ inside the expression P_1, so that we will not be able to differentiate between their corresponding mappings in $\mathcal{K}[\![P_1]\!]$. By computing separately $\mathcal{K}_{up}[\![P_1]\!]$ and $\mathcal{K}_{\overline{up}}[\![P_1]\!]$ we exclude any confusion. Note that, according to Lemma 3.5, $\mathcal{E}_{\emptyset}[\![P_2]\!] = \mathcal{E}_{\Gamma}[\![P_2]\!]$ for all environments Γ. We can therefore use the empty environment in rule (5) for the operator $\mathcal{K}_{up}$ in Table 3.13 for simplicity.

In rule (5) in Table 3.13 we reflect the fact that exposed labels of two expressions connected by the choice construct mutually "kill" each other – this is basically due to the semantic rules for the choice operator (2)-(3) and (11)-(12) in Table 3.5. The rest of the rules for the operator $\mathcal{K}_{up}$ in Table 3.13 are straightforward. For any prefixed expression the label of the prefix "kills" itself (rules(1)-(4)). We take a union of two sets returned by $\mathcal{K}_{up}$ on two expressions put in parallel (rule (7)) because a transition of one of two parallel processes does not influence the other one. Rule (9) has been added for simplicity in some proofs below, similarly to the generate operator.

The rules for the operator $\mathcal{K}_{\overline{up}}$ are presented in Table 3.14. We need to use

$$
\begin{array}{rcll}
\mathcal{K}_{\overline{up}}[\![\mathsf{a}^{\ell}.X]\!] & = & \emptyset & (1)\\
\mathcal{K}_{\overline{up}}[\![\lambda^{\ell}.X]\!] & = & \emptyset & (2)\\
\mathcal{K}_{\overline{up}}[\![\mathsf{a}^{\ell}.P]\!] & = & \mathcal{K}[\![P]\!] & (3)\\
\mathcal{K}_{\overline{up}}[\![\lambda^{\ell}.P]\!] & = & \mathcal{K}[\![P]\!] & (4)\\
\mathcal{K}_{\overline{up}}[\![P_1 + P_2]\!] & = & \mathcal{K}_{\overline{up}}[\![P_1]\!] \cup \mathcal{K}_{\overline{up}}[\![P_2]\!] & (5)\\
\mathcal{K}_{\overline{up}}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathcal{K}_{\overline{up}}[\![P]\!] & (6)\\
\mathcal{K}_{\overline{up}}[\![P_1 \,{}_{\shortparallel}{}_{A}{}_{\shortparallel}\, P_2]\!] & = & \mathcal{K}_{\overline{up}}[\![P_1]\!] \cup \mathcal{K}_{\overline{up}}[\![P_2]\!] & (7)\\
\mathcal{K}_{\overline{up}}[\![\underline{X := P}]\!] & = & \mathcal{K}_{\overline{up}}[\![P]\!] & (8)\\
\mathcal{K}_{\overline{up}}[\![X]\!] & = & \emptyset & (9)\\
\mathcal{K}_{\overline{up}}[\![\mathbf{0}]\!] & = & \emptyset & (10)
\end{array}
$$

Table 3.14 – Definition of the operator $\mathcal{K}_{\overline{up}} : \mathrm{IMC}^{\mathrm{G}} \rightarrow 2^{\mathbf{Lab}\times\mathfrak{M}}$ returning "killed" labels with the upper syntactic level excluded.

the operator $\mathcal{K}$ for the rules (1)-(4) but this is not a problem because we are applying the operator $\mathcal{K}$ to a strict subexpression. The rest of the rules are clear. We take a union of two sets returned by $\mathcal{K}_{\overline{up}}$ on two summands (rule (5)) or two parallel processes (rule (7)) because, unlike for the operator $\mathcal{K}_{up}$, results returned by $\mathcal{K}_{\overline{up}}$ on two summands do not need to be altered.

We can prove now Lemma 3.8 concerning application results of the generate and kill operators to well-formed expressions before and after a variable substitution has been performed. This lemma will be useful in order to prove Lemmas 3.10 and 3.11 below concerning the comparison of the output of the generate and kill operators on an $\mathrm{IMC}^{\mathrm{G}}$ program and on its derivative expressions. The substitution is important because it is conducted in the SOS rules (9) and (16) in Table 3.5 during the recursion unfolding.

Lemma 3.8 (Generate and kill operators under substitution)**.** *Given well-formed* $\mathrm{IMC}^{\mathrm{G}}$ *expressions* E' *and* $\underline{X := E}$, $\Gamma \in 2^{\boldsymbol{Var}\times\mathfrak{M}}$, $(X, \mathcal{E}_{\emptyset}[\![E]\!])$ *a unique mapping for* X *in* Γ, *then* $\mathcal{G}_{\Gamma}[\![E'\{\underline{X := E}/X\}]\!] \subseteq \mathcal{G}_{\Gamma}[\![E']\!] \cup \mathcal{G}_{\Gamma}[\![\underline{X := E}]\!]$ *and* $\mathcal{K}[\![E'\{\underline{X := E}/X\}]\!] \subseteq \mathcal{K}[\![E']\!] \cup \mathcal{K}[\![\underline{X := E}]\!]$ *hold.*

Proof. See Appendix A. □

$$
\begin{array}{rcll}
\Lambda_{\mathbf{Lab}}[\![\mathsf{a}^{\ell}.X]\!] & = & \{(\ell,\mathsf{a})\} & (1)\\
\Lambda_{\mathbf{Lab}}[\![\lambda^{\ell}.X]\!] & = & \{(\ell,\lambda)\} & (2)\\
\Lambda_{\mathbf{Lab}}[\![\mathsf{a}^{\ell}.P]\!] & = & \{(\ell,\mathsf{a})\} \cup \Lambda_{\mathbf{Lab}}[\![P]\!] & (3)\\
\Lambda_{\mathbf{Lab}}[\![\lambda^{\ell}.P]\!] & = & \{(\ell,\lambda)\} \cup \Lambda_{\mathbf{Lab}}[\![P]\!] & (4)\\
\Lambda_{\mathbf{Lab}}[\![P_1 + P_2]\!] & = & \Lambda_{\mathbf{Lab}}[\![P_1]\!] \cup \Lambda_{\mathbf{Lab}}[\![P_2]\!] & (5)\\
\Lambda_{\mathbf{Lab}}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \Lambda_{\mathbf{Lab}}[\![P]\!] & (6)\\
\Lambda_{\mathbf{Lab}}[\![P_1 \,{}_{\shortparallel}A_{\shortparallel}\, P_2]\!] & = & \Lambda_{\mathbf{Lab}}[\![P_1]\!] \cup \Lambda_{\mathbf{Lab}}[\![P_2]\!] & (7)\\
\Lambda_{\mathbf{Lab}}[\![X := P]\!] & = & \Lambda_{\mathbf{Lab}}[\![P]\!] & (8)\\
\Lambda_{\mathbf{Lab}}[\![X]\!] & = & \emptyset & (9)\\
\Lambda_{\mathbf{Lab}}[\![\mathbf{0}]\!] & = & \emptyset & (10)
\end{array}
$$

Table 3.15 – Definition of the $\Lambda_{\mathbf{Lab}} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$ operator.

3.4.4 Chains operator

In this section we will define the last of the major operators that we need for our Pathway Analysis of $\mathrm{IMC}^{\mathrm{G}}$– the chains operator $\mathfrak{T}$. In doing this we will need one more auxiliary operator – the operator $\Lambda_{\mathbf{Lab}}$ which returns for an $\mathrm{IMC}^{\mathrm{G}}$ expression a result from $2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$. This operator collects mappings from the labels occurring in an $\mathrm{IMC}^{\mathrm{G}}$ expression to their corresponding action names or delay rates.

The rules for the operator $\Lambda_{\mathbf{Lab}}$ are presented in Table 3.15. These rules are fairly straightforward – we take a union of all the encountered mappings. In case we encounter several appearances of the same label with different corresponding action names or delay rates, we save all of them in the resulting set. This will however never be the case for uniquely labelled $\mathrm{IMC}^{\mathrm{G}}$ expressions and their derivatives. It is easy to see that the correspondence between labels and action names or delay rates does not change after any number of semantic transitions.

We will need an auxiliary function $Name^{h}$ (see the definition in Table 3.16) that has a multiset of labels as its input, makes use of the environments $\Lambda \in 2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$ and $A \subseteq \mathbf{Act}$, and returns an action name or delay rate corresponding to labels in the domain of its input. The correspondence between labels and action names or relay rates is looked up in the environment Λ. In case the corresponding action names or delay rates of labels in the domain of the multiset differ, then "undefined" (i.e. symbol "?") is returned. For action names the internalisation will be taken into account – for internalised action names the internal action name τ will be returned. The internalised action names are considered to be those that are not contained in the environment A. This environment will usually contain free names of some $\mathrm{IMC}^{\mathrm{G}}$ expression.

$$Name_{\Lambda}(M) = \begin{cases} \alpha & \text{if } (M \neq \bot_{\mathfrak{M}}) \wedge (\forall \ell \in dom(M) \forall (\ell, \beta) \in \Lambda \ \beta = \alpha) \\ ? & \text{otherwise.} \end{cases}$$

$$Name^{h}_{\Lambda,A}(M) = \begin{cases} \tau & \text{if } Name_{\Lambda}(M) \in \mathbf{Act} \backslash A \\ Name_{\Lambda}(M) & \text{otherwise.} \end{cases}$$

Table 3.16 – Definition of the operators $Name_{\Lambda} : \mathfrak{M} \to \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate} \cup \{?\}$ and $Name^{h}_{\Lambda,A} : \mathfrak{M} \to \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate} \cup \{?\}$, $\Lambda \in 2^{\mathbf{Lab} \times (\mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate})}$, $A \subseteq \mathbf{Act}$, $M \in \mathfrak{M}$.

We will introduce now the important concept of "chains". A chain always refers to an IMC$^{\mathrm{G}}$ expression, i.e. it does not have a particular meaning without this relation. As a starting point we can declare that every multiset of labels that decorates a transition derivable from the SOS rules for a particular IMC$^{\mathrm{G}}$ expression is contained in the set of chains for that expression. For example, for $\underline{X} := \mathsf{b}^{\ell_3}.\mathsf{a}^{\ell_1}.\underline{X} \parallel_{\mathsf{a}} \underline{Y} := \mathsf{c}^{\ell_4}.\mathsf{a}^{\ell_2}.\underline{Y}$ the set of chains will contain the chains $\bot_{\mathfrak{M}}[\ell_3 \mapsto 1]$ and $\bot_{\mathfrak{M}}[\ell_4 \mapsto 1]$, because both of these multisets decorate the transitions derivable from it. A set of chains also contains multisets of labels that may become executable later on: the label multiset $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ is also contained in the set of chains of the above example, even though it will become executable no earlier than b^{ℓ_3} and c^{ℓ_4} have been executed. In fact we will prove below in Lemma 3.12 that for well-formed IMC$^{\mathrm{G}}$ expressions the set of their chains contains all the chains of their derivatives.

The chains operator $\mathfrak{T}$ is defined in Table 3.17. It operates on IMC$^{\mathrm{G}}$ expressions and returns elements from $2^{\mathfrak{M}}$. We parametrise the chains operator with an environment $\Lambda \in 2^{\mathbf{Lab} \times (\mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate})}$. We will usually assume that the operator $\Lambda_{\mathbf{Lab}}$ has been run beforehand on the same IMC$^{\mathrm{G}}$ expression and its output has been saved in the environment Λ.

The rules for the operator $\mathfrak{T}$ are easy to understand besides rule (7) for the parallel construct. We can directly add to the resulting set of chains all the chains from both IMC$^{\mathrm{G}}$ processes running in parallel that either refer to delay rates or to action names which are not in the synchronisation set or have been internalised. Chains with their corresponding action names being internalised "behave" as τ-chains (see the semantic rule (8) in Table 3.5), so the synchronisation construct cannot influence their execution anymore. We are using the function $Name^{h}_{\Lambda,A}$ defined in Table 3.16, in order to determine an action name or a delay rate corresponding to a particular chain.

Additionally chains from two parallel processes that correspond to the same non-internalised action name from the synchronisation set are combined in all

$$
\begin{array}{rcll}
\mathfrak{T}_\Lambda[\![\mathsf{a}^\ell.X]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} & (1)\\
\mathfrak{T}_\Lambda[\![\lambda^\ell.X]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} & (2)\\
\mathfrak{T}_\Lambda[\![\mathsf{a}^\ell.P]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} \cup \mathfrak{T}_\Lambda[\![P]\!] & (3)\\
\mathfrak{T}_\Lambda[\![\lambda^\ell.P]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} \cup \mathfrak{T}_\Lambda[\![P]\!] & (4)\\
\mathfrak{T}_\Lambda[\![P_1 + P_2]\!] & = & \mathfrak{T}_\Lambda[\![P_1]\!] \cup \mathfrak{T}_\Lambda[\![P_2]\!] & (5)\\
\mathfrak{T}_\Lambda[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathfrak{T}_\Lambda[\![P]\!] & (6)\\
\mathfrak{T}_\Lambda[\![P_1 \between A \between P_2]\!] & = & \{C | C \in \mathfrak{T}_\Lambda[\![P_1]\!], Name^h_{\Lambda,\mathbf{fn}(P_1)}(C) \notin A\} \cup & \\
 & & \{C | C \in \mathfrak{T}_\Lambda[\![P_2]\!], Name^h_{\Lambda,\mathbf{fn}(P_2)}(C) \notin A\} \cup & \\
 & & \{C_1 + C_2 | C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!], C_2 \in \mathfrak{T}_\Lambda[\![P_1]\!], & \\
 & & Name^h_{\Lambda,\mathbf{fn}(P_1)}(C_1) \cap Name^h_{\Lambda,\mathbf{fn}(P_2)}(C_2) \cap A \neq \emptyset\} & (7)\\
\mathfrak{T}_\Lambda[\![\underline{X := P}]\!] & = & \mathfrak{T}_\Lambda[\![P]\!] & (8)\\
\mathfrak{T}_\Lambda[\![\underline{X}]\!] & = & \emptyset & (9)\\
\mathfrak{T}_\Lambda[\![\mathbf{0}]\!] & = & \emptyset & (10)
\end{array}
$$

Table 3.17 – Definition of the chains operator $\mathfrak{T} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathfrak{M}}$, $\Lambda \in 2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$.

possible combinations and added to the resulting set of rule (7). Note that from the construction of chains follows that each chain has a unique corresponding action name or delay rate, i.e. $Name^h_{\Lambda,A}$ on a chain will always return an action name or delay rate. Also note that chains corresponding to delay rates will have only one label in their domain (with one occurrence), because delays do not synchronise in $\mathrm{IMC}^{\mathrm{G}}$.

We will prove in Lemma 3.9 that substitutions in $\mathrm{IMC}^{\mathrm{G}}$ expressions do not lead to any "new" chains. This lemma is analogue to Lemma 3.8 concerning the generate and kill operators results after the substitutions in $\mathrm{IMC}^{\mathrm{G}}$ expressions.

Lemma 3.9 (Chains operator under substitution)**.** *Given well-formed* $\mathrm{IMC}^{\mathrm{G}}$ *expressions* E *and* E'*, then* $\mathfrak{T}_\Lambda[\![E\{E'/X\}]\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!] \cup \mathfrak{T}_\Lambda[\![E']\!]$ *holds for all* $\Lambda \in 2^{\boldsymbol{Lab}\times(\boldsymbol{Act}\cup\{\tau\}\cup\boldsymbol{Rate})}$.

Proof. See Appendix A. □

3.5 Correctness and precision

In this section we will prove several theoretical results about the Pathway Analysis of IMC$^{\mathrm{G}}$ programs. These results will be enough in order to prove that our version of Pathway Analysis is correct and precise. While performing the Pathway Analysis we obtain therefore a form of the concise representation of the semantics of an IMC$^{\mathrm{G}}$ program, avoiding any imprecision stemming from the analysis technique. At the same time, as we will see in Chapters 4 and 5 below, the results of conducting the Pathway Analysis are often more convenient for further analysis then the syntax of IMC$^{\mathrm{G}}$.

We will first show the *correctness* of the Pathway Analysis: we will prove that the Pathway Analysis results on IMC$^{\mathrm{G}}$ programs are in some sense also applicable to all their derivative expressions. In particular, we will show that the results returned by the generate and chains operators on derivative expressions are subsets of the results returned by the same operators on the "initial" IMC$^{\mathrm{G}}$ program on which they have been computed. The situation is more complicated for the kill operator, as we do not necessary have the subset relation. We will however prove that the kill operator results on the initial IMC$^{\mathrm{G}}$ program and on its derivative expressions are equivalent when subtracted from the exposed labels of the derivative expression – and this is the only way we will use them in the future.

After this we will also prove several results concerning the *precision* of the Pathway Analysis on IMC$^{\mathrm{G}}$ programs. In particular, we will show how to use the Pathway Analysis results to predict transitions and we will prove that all enabled transitions of all IMC$^{\mathrm{G}}$ expressions derivable from the initial IMC$^{\mathrm{G}}$ program are predictable in this way. The generate and chains operator results on the initial expression are precise enough to exactly calculate the exposed labels of the derivative expressions after any number of steps. Finally, we will show that there cannot be more than one exposed label of the same kind for any derivative of an IMC$^{\mathrm{G}}$ program. In particular, this means that any transition derivable for it has a unique multiset of labels decorating it. All these results will lead us to the algorithm for building a labelled transition system which is strongly bisimilar to the labelled transition system induced by the semantics of the initial IMC$^{\mathrm{G}}$ program.

3.5.1 Correctness

In the first Lemma 3.10 we will prove that the generate operator's output on any derivative expression is included in the output of the same operator on the

initial IMC$^{\mathrm{G}}$ program from which it can be derived after several transitions. The inclusion relation may be strict, i.e. we may "lose" some behaviour as a result of the transition. For example, we have $\mathcal{G}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}]\!] = \{(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1]), (\ell_2, \bot_{\mathfrak{M}}[\ell_2 \mapsto 1])\}$, but for IMC$^{\mathrm{G}}$ expressions derivable from it after one transition we have either $\mathcal{G}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!] = \{(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1])\}$ or $\mathcal{G}_\Gamma[\![\underline{Y := \mathsf{b}^{\ell_2}.Y}]\!] = \{(\ell_2, \bot_{\mathfrak{M}}[\ell_2 \mapsto 1])\}$ while making use of the environment $\Gamma = \Gamma_{\mathbf{Var}}[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}]\!]$.

Lemma 3.10 (Generate operator under transitions)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F, $\Gamma = \Gamma_{\boldsymbol{Var}}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $E \longrightarrow E'$, *then* $\mathcal{G}_\Gamma[\![E']\!] \subseteq \mathcal{G}_\Gamma[\![E]\!]$ *holds.*

Proof. See Appendix A. □

As already mentioned, the relation of the kill operator results on the initial IMC$^{\mathrm{G}}$ expression and on its derivatives is not necessary a subset relation. We can however prove the equality after the subtractions:

Lemma 3.11 (Kill operator under transitions)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F, $\Gamma = \Gamma_{\boldsymbol{Var}}[\![F]\!]$, $F \xrightarrow{*} E$, $(\ell, M_1) \in \mathcal{K}[\![E]\!]$, $\bot_{\mathfrak{M}}[\ell \mapsto 1] \leq \mathcal{E}_\Gamma[\![E]\!]$, $(\ell, M_2) \in \mathcal{K}[\![F]\!]$, *then* $\mathcal{E}_\Gamma[\![E]\!] - M_1 = \mathcal{E}_\Gamma[\![E]\!] - M_2$ *holds.*

Proof. See Appendix A. □

Note that under the conditions in Lemma 3.11 it can actually happen that $\mathcal{K}[\![E]\!]$ contains two different multisets referring to the same label. In this case the result of their subtraction from the exposed labels of E will still be the same. Let us have a look at the following example: it holds that $(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1, \ell_3 \mapsto 1]) \in \mathcal{K}[\![\underline{Y := \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{c}^{\ell_2}.Y} + \mathsf{b}^{\ell_3}.\mathbf{0}}]\!]$. On the other hand, for the derivative IMC$^{\mathrm{G}}$ expression we have two different elements mapped to ℓ_1: $\mathcal{K}[\![\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{c}^{\ell_2}.Y} := \underline{\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{c}^{\ell_2}.Y} + \mathsf{b}^{\ell_3}.\mathbf{0}}]\!]$ contains both $(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1, \ell_3 \mapsto 1])$ and $(\ell_1, \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1])$.

It holds however that $\mathcal{E}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{c}^{\ell_2}.Y} := \underline{\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{c}^{\ell_2}.Y} + \mathsf{b}^{\ell_3}.\mathbf{0}}]\!]$ is equal to $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$, therefore two subtraction results are equal. It

holds that $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1] - \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1] = \bot_{\mathfrak{M}}$ and it also holds that $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1] - \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1, \ell_3 \mapsto 1] = \bot_{\mathfrak{M}}$, i.e. the result of the subtraction is the same in accordance with the statement in Lemma 3.11.

We will prove in Lemma 3.12 that there is an inclusion relation between the results of the chains operator on any derivative expression and on the initial expression if it is well-formed:

Lemma 3.12 (Chains inclusion)**.** *Given a well-formed* IMC^{G} *expression* F*,* $F \xrightarrow{*} E$*,* $\Lambda = \Lambda_{\boldsymbol{Lab}}[\![F]\!]$*, then* $\mathfrak{T}_\Lambda[\![E]\!] \subseteq \mathfrak{T}_\Lambda[\![F]\!]$ *holds.*

Proof. See Appendix A. □

Note that some chains may be "lost" after a transition, i.e. the inclusion relation in Lemma 3.12 can be strict. For example, for the transition $\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathsf{a}^{\ell_2}.\mathbf{0} \xrightarrow[\bot_{\mathfrak{M}}[\ell_1 \mapsto 1,\, \ell_2 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathbf{0}$ it holds $\mathfrak{T}_\Lambda[\![\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathsf{a}^{\ell_2}.\mathbf{0}]\!] = \{\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]\}$ but $\mathfrak{T}_\Lambda[\![\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathbf{0}]\!] = \emptyset$, if $\Lambda = \Lambda_{\mathbf{Lab}}[\![\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathsf{a}^{\ell_2}.\mathbf{0}]\!]$. The set of chains for the derivative expression is empty even though the process on the left ($\underline{X := \mathsf{a}^{\ell_1}.X}$) can still perform the action a, because no transition is derivable for the terminal process on the right.

3.5.2 Precision

In the following Lemma 3.13 we will prove that every chain of an IMC^{G} program is also a chain of any of its derivative expression, provided that all the labels from the chain domain occur in the derivative expression.

Lemma 3.13 (Chains preservation)**.** *Given an* IMC^{G} *program* F*,* $F \xrightarrow{*} E$*,* $\Lambda = \Lambda_{\boldsymbol{Lab}}[\![F]\!]$*,* $C \in \mathfrak{T}_\Lambda[\![F]\!]$*,* $dom(C) \subseteq Labs(E)$*, then* $C \in \mathfrak{T}_\Lambda[\![E]\!]$ *holds.*

Proof. See Appendix A. □

From the next Lemma 3.14 follows that we can predict all the enabled transitions (and only them) based only on exposed labels and on computed chains of an IMC^{G} expression if it is a derivation from some IMC^{G} program. The action name or delay rate decorating the transition can be determined based both on the output of the operator $\Lambda_{\mathbf{Lab}}$ on the IMC^{G} expression and (for action names) on the knowledge about free names of the expression. For internalised action names the transition is decorated with the τ-action. For delay-transitions the maximal progress assumption has to be taken into account.

Lemma 3.14 (Transition existence)**.** *Given an* IMC^{G} *program* F, $F \xrightarrow{*} E$, $\Lambda = \Lambda_{\boldsymbol{Lab}}[\![E]\!]$ *and* $\Gamma = \Gamma_{\boldsymbol{Var}}[\![E]\!]$, *then* $E \xrightarrow[C]{\alpha} E'$ *holds if and only if* $C \in \mathfrak{T}_\Lambda[\![E]\!]$ *and* $C \leq \mathcal{E}_\Gamma[\![E]\!]$, *and one of the following cases occurs:* $\alpha = Name^h_{\Lambda,\mathbf{fn}(E)}(C)$ *and* $\alpha \in \boldsymbol{Act} \cup \{\tau\}$ *or* $\alpha = Name^h_{\Lambda,\mathbf{fn}(E)}(C)$, $\alpha \in \boldsymbol{Rate}$ *and there is no chain* $C' \in \mathfrak{T}_\Lambda[\![E]\!]$, $C' \leq \mathcal{E}_\Gamma[\![E]\!]$ *with* $Name^h_{\Lambda,\mathbf{fn}(E)}(C') = \tau$.

Proof. See Appendix A. □

In the next Lemma 3.15 we will prove that multisets of exposed labels of IMC^{G} programs and all their derivatives belong in fact to a special type of multisets, with each label mapped either to zero or one. This is not astonishing taking into account the results on the precision of the Pathway Analysis that we have already proved. This result will make it easier to discuss the precision of the Pathway Analysis as we do not need to take into account the possibility of several exposed occurrences of the same label.

Lemma 3.15 (Unique exposed labels)**.** *Given an* IMC^{G} *program* F, $F \xrightarrow{*} E$ *and* $\Gamma = \Gamma_{\boldsymbol{Var}}[\![F]\!]$, *then either* $\mathcal{E}_\Gamma[\![E]\!](\ell) = 0$ *or* $\mathcal{E}_\Gamma[\![E]\!](\ell) = 1$ *holds for all* $\ell \in \boldsymbol{Lab}$. *Given two transitions* $E \xrightarrow[C_1]{} E'$ *and* $E \xrightarrow[C_2]{} E''$ *with different derivation trees, then* $C_1 \neq C_2$ *holds.*

Proof. See Appendix A. □

The next Theorem 3.16 is important because it proves that our analysis is exact in relation to exposed labels of an IMC^{G} expression derivable after one

semantic transition from another IMC$^{\mathrm{G}}$ expression the exposed labels of which are already known. We will show below in the proof of Theorem 3.17 that this result is extendable to any number of semantic transitions due to the lemmas proved above.

Theorem 3.16 (Pathway Analysis is precise)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F, $F \xrightarrow{*} E$ *and* $E \xrightarrow[C]{\alpha} E'$, *then* $\mathcal{E}_\Gamma[\![E]\!] - \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{K}_{up}[\![E]\!]\} + \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{G}_\Gamma[\![E]\!]\} = \mathcal{E}_\Gamma[\![E']\!]$.

Proof. See Appendix A. □

3.6 Worklist Algorithm

In the last section of this chapter we will put all the results that have been proved until now together. We show how to use the results of the Pathway Analysis in order to build a labelled transition system where all the states and transitions have their corresponding states and transitions in the labelled transition system induced by the semantic of the IMC$^{\mathrm{G}}$ program that has been analysed and the other way around. This clearly proves that our Pathway Analysis is correct and precise.

In the previous work on Data Flow / Pathway Analysis (see [NN07], [Pil07], [NNN07]) the definitions of the operators on the syntax of a particular process calculus have been followed by devising a so-called Worklist Algorithm. The purpose of the latter was to build a labelled transition system that could simulate the labelled transition system induced by the semantics of a process calculus expression for which the Data Flow / Pathway Analysis has been conducted. We will now present the schematic Worklist Algorithm that will complete our analysis. It is quite similar to the Worklist Algorithms from the previous works but it does not make use of a granularity function (the semantic models of IMC$^{\mathrm{G}}$ are always finite, therefore it is not necessary) and makes use of several auxiliary operators.

The Worklist Algorithm accepts as input an IMC$^{\mathrm{G}}$ program F and returns as output a labelled transition system with states denoted by multisets of exposed labels. It is schematically described in Table 3.18. The idea is to start with applying the Pathway Analysis operators to F. The auxiliary operators $\Gamma_{\mathbf{Var}}$,

1. Compute $\Gamma := \Gamma_{\mathbf{Var}}[\![F]\!]$ and $\Lambda := \Lambda_{\mathbf{Lab}}[\![F]\!]$.

2. Compute $\mathbf{fn}(F)$.

3. Compute $\mathcal{E}_\Gamma[\![F]\!]$: this label multiset will characterise the first state of the labelled transition system to be constructed. Add the first state to the *worklist*: $worklist := \{\mathcal{E}_\Gamma[\![F]\!]\}$. Assign it to the current state M of the Worklist Algorithm: $M := \mathcal{E}_\Gamma[\![F]\!]$.

4. Compute $\mathcal{G}_\Gamma[\![F]\!]$, $\mathcal{K}[\![F]\!]$ and $\mathfrak{T}_\Lambda[\![F]\!]$, i.e. the generate, kill and chains operators on F.

5. For each $C \in \mathfrak{T}_\Lambda[\![F]\!]$, such that $Name^h_{\Lambda,\mathbf{fn}(F)}(C) \in \mathbf{Act} \cup \{\tau\}$, check whether $C \leq M$. If yes, create a new state (if not already existent) $M' := M - \sum_{\ell \in dom(C)}\{M_1 | (\ell, M_1) \in \mathcal{K}[\![F]\!]\} + \sum_{\ell \in dom(C)}\{M_2 | (\ell, M_2) \in \mathcal{G}_\Gamma[\![F]\!]\}$ and add it to *worklist*. Create a transition from the state M to the state M': the transition is decorated by the action name $\alpha = Name^h_{\Lambda,\mathbf{fn}(F)}(C)$ and by the label multiset C.

6. If no transitions decorated with τ have been created in p.5, repeat p.5 for each $C \in \mathfrak{T}_\Lambda[\![F]\!]$ such that $Name^h_{\Lambda,\mathbf{fn}(F)}(C) \in \mathbf{Rate}$.

7. Check whether *worklist* is empty. If yes, we are done. If no, choose an arbitrary state out of the *worklist*, assign it to M and proceed with p.5.

Table 3.18 – The Worklist Algorithm which builds a labelled transition system given an IMC^{G} program F.

$\Lambda_{\mathbf{Lab}}$ and $\mathbf{fn}$ are applied to F first (can be applied in any order). Consequently the exposed, generate, kill and chain operators are computed (their order is in fact not important as well). Then, starting with the multiset of exposed labels of F, all the possible label multisets reachable from the exposed multiset of F in the sense of Theorem 3.16 are created. Note that the maximal progress is taken into account in rule (6): rate-transitions are created only from the states that do not have any outgoing τ-transitions.

We will prove in Theorem 3.17 that the Worklist Algorithm constructs a labelled transition system where there is a clear correspondence between the states added by the Worklist Algorithm and the expressions derivable from F. We say that a

state M corresponds to E such that $F \xrightarrow{*} E$ if $M = \mathcal{E}_\Gamma[\![E]\!]$ with $\Gamma = \Gamma_{\mathbf{Var}}[\![F]\!]$. Moreover, no states are added by the Worklist Algorithm for which there is no corresponding derivative of F. The transitions between pairs of states created by the Worklist Algorithm correspond to the transitions in the semantics of F as well. There is also a termination guarantee for the algorithm.

Theorem 3.17 (Worklist algorithm)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F*, then the Worklist Algorithm terminates on* F*. The Worklist Algorithm creates a state* M *if and only if there exists an* $\mathrm{IMC}^{\mathrm{G}}$ *expression* E *such that* $F \xrightarrow{*} E$ *and* $M = \mathcal{E}_\Gamma[\![E]\!]$ *with* $\Gamma = \Gamma_{\boldsymbol{Var}}[\![F]\!]$*. Moreover, from* $F \xrightarrow{*} E_1$*,* $F \xrightarrow{*} E_2$*,* $M_1 = \mathcal{E}_\Gamma[\![E_1]\!]$ *and* $M_2 = \mathcal{E}_\Gamma[\![E_2]\!]$ *follows that* $E_1 \xrightarrow[C]{\alpha} E_2$ *if and only if the Worklist Algorithm creates the transition* $M_1 \xrightarrow[C]{\alpha} M_2$*.*

Proof. See Appendix A. □

In Chapter 4 we will introduce a notion of *strong bisimilarity* on $\mathrm{IMC}^{\mathrm{G}}$ systems. With this knowledge we will be able to show that the Worklist Algorithm constructs a labelled transition system which is strongly bisimilar to the labelled transition system induced by the semantics of F.

Chapter 4

Bisimulation relations and Pathway Analysis

4.1 Bisimulations and logical equivalences

In this chapter we will discuss a special class of binary relations which can be established on the states of labelled transition systems, namely, bisimulation relations. These relations are equivalence relations (reflexive, symmetric and transitive) of a special kind: states that are in a bisimulation relation with each other (called "bisimilar" states) "behave" in a similar way. The bisimulation equivalence is not the same as the trace equivalence and is in general finer than the latter (see, for example, [BK08] for a comparison), as it is the possible behaviour of states that is compared and not only the behavioural traces without reference to the states that were passed through while producing the trace.

The understanding of what a similar behaviour means can be very different, also the labelled transition systems can be of different types, therefore there exist lots of types of bisimulation relations. For example, for labelled transition systems with labelling of transitions by external action names it is usual to apply a notion of a so-called *strong* bisimulation. In a strong bisimulation two states are bisimilar when each transition from one of the states has a corresponding transition from the other one of the states – for transitions to be corresponding

means that both transitions have the same labelling and go to states that are again bisimilar. This is a notion of bisimulation that has been introduced by Park [Par81] and Milner [Mil89].

Many other types of bisimulation have been introduced later on – weak bisimulation [Mil90], probabilistic bisimulation [LS89], Markovian bisimulation [HR94] etc. There have also been defined bisimulations based on state labellings (in contrast to transition labellings) but these two kinds of bisimulations are mutually convertible if the labelling is qualitative (e.g. by action names) [BK08]. Such conversion is however not possible for transitions decorated by quantitative labels (for example, by rates), as in order to calculate the probabilities of the transitions we should know all transition rates to other states and not only the cumulative exit rate of a state.

It is usual to look for the coarsest bisimulation relations, i.e. the coarsest relations that satisfy the definition of the particular bisimulation. The coarsest bisimulation relations are usually called *bisimilarities*. Bisimulations and bisimilarities are often directly linked to logics: for many bisimulation/bisimilarity relations there has been constructed a corresponding logic that in some sense captures the "meaning" of the bisimulation relation. The logic is sound or both sound and complete for a particular bisimulation relation.

We understand soundness in such a way that formulas in the logic either hold or do not hold on all bisimilar states. By completeness we mean that for each pair of non-bisimilar states there exists a formula in a particular logic with holds on one of the states and does not hold on the other one, i.e. there is a formula in the logic that can *discriminate* non-bisimilar states. For example, on finite labelled transition systems without terminal states, both CTL and CTL* fully characterise strong bisimilarity [BK08], i.e. they are sound and complete for strong bisimilarity. The same result can be established for infinite transition systems with finite branching but not for infinite transition systems in general were there can exist non-bisimilar states which are equivalent according to all CTL formulas [BK08].

Bisimulation relations on states can be naturally extended to relations on transition systems: for a fixed bisimulation relation on states we consider two transition systems to be in a bisimulation relation if every initial state of each of the systems has a corresponding bisimilar initial state in the other system. On the other hand, for those bisimulation relations that are equivalences it is possible to build *bisimulation quotients* of transition systems: in a bisimulation quotient each equivalence class is represented by only one state, which can lead to a considerable state space reduction. Equivalent states are thus merged or in a sense "compressed" together thus obtaining a smaller system for further analysis. A bisimulation quotient is clearly smaller for coarser bisimulations. The smallest

bisimulation quotient is computed based on the coarsest bisimulation, i.e. on the bisimilarity.

There are standard methods for computing bisimulation relations which usually are aiming at computing the coarsest bisimulations. The idea is to start with all states with the same labelling considered as potentially bisimilar and refining equivalence classes until no refinement is possible. The exact way how a partition should be chosen for the refinement in the next step can be different and therefore leaves space for improvements [BK08].

Building bisimulation quotients is a preferred way of dealing with state space explosion [BK08] and often yields a considerable performance gain. In case the bisimulation relation based on which the states are merged is not the coarsest one, then model-checking of a corresponding logic on a smaller system is still sound [BK08] even though some states may be indistinguishable in the logic. This is an application that we envision for our approach as we will be constructing bisimulations (see Sections 4.3 and 4.4) and not bisimilarities in general.

It is very useful if a bisimulation relation is also a congruence relation for the main composition operators of process calculi (for example, synchronisation, choice or internalisation operators). Then bisimilarity-minimisation can be carried out before the application of, for example, synchronisation, rather then after it, and can be applied many times after the application of each compositional operator, which leads for many systems to a considerable decrease in the complexity of computing the bisimulation quotient [ASSB94]. A similar method can be used for abstractions, where abstractions are applied many times on system subcomponents rather than once on a whole system [KKN09].

Hermanns has proposed an algorithm for computing the strong bisimilarity on IMC systems in [Her02]. His algorithm is based on combining the standard algorithms for computing the strong bisimilarity on labelled transition systems (see, for example, [KS83]) and the Markovian bisimilarity on CTMCs [Her93]. We will show how his algorithm can be transferred into our setting before going on with our own algorithms for computing bisimulations based on the results of the Pathway Analysis of $\mathrm{IMC}^{\mathrm{G}}$.

There have been proposed improvements of the algorithm from [Her02] for systems that do not have cycles [CHZ08]. Those can be, for example, systems that model dynamic fault trees [BCS07]. In acyclic labelled transition systems it is possible to construct the bisimilarity relation by visiting each state only once, starting with absorbing states, this is why the complexity of construction is only $O(m)$, where m is a number of transitions [CHZ08]. However, as we are mainly interested in studying systems with a repetitive behaviour, we will not discuss the algorithms for acyclic IMCs further on.

We have discussed in Section 2.1.3 that those IMC expressions that do not have interactive cycles in their semantics can be transformed into CTMDPs [Joh07]. Algorithms for computing bisimulation relations on CTMDPs are therefore relevant also for IMC systems. In [BHH$^+$09] an algorithm for computing bisimulations on CTMDPs (which can be considered as a subclass of IMCs) has been proposed which uses optimisation techniques based on BDDs. BDDs are used both for assigning states to partition blocks and for computing "signatures" of the states: signatures denote whether there is a transition decorated by the fixed action to any of the states of the transition block. Moreover a so-called "backward signature" is computed for each block in a current state partition in order to know which blocks might be influenced if the current partition would be split – this allows to devise a splitting algorithm which is more efficient in the general case than the usual one. The worst-case complexity is however the same as for the algorithm without BDDs.

Our own approach differs in several ways from the usual methods for computing bisimulations. On the one hand, we are not computing the coarsest bisimulations (i.e. bisimilarities) but rather bisimulations that are easy enough to compute – our aim is that the complexity of computations is polynomial in the length of the syntactic description of an IMC$^{\mathrm{G}}$ system and not on the number of states in the labelled transition system representing the semantics of the IMC$^{\mathrm{G}}$ system. We are achieving this by analysing labels (i.e. the generate and kill operator on them and chains containing them) for being bisimilar and using the results of this analysis for concluding (with an easy enough procedure) that two multisets are or are not bisimilar.

In the rest of the chapter we will first present a definition of a strong bisimulation on IMC systems (Section 4.2.1), its adaptation to label multisets (Section 4.2.2) and a standard algorithm for computing bisimulations adopted to the label multiset setting (Section 4.2.4). After this we will present two equivalence relations on multisets – a so-called chain-bisimulation (Section 4.3) and a synchronisation-bisimulation (Section 4.4) – of which the first one is easier to compute than the second one. On the other hand, chain-bisimulation is contained in synchronisation-bisimulation. We prove that both relations are indeed bisimulations and that there is an inclusion relation between them.

Note that if we can derive that two multisets are contained either in a chain- or synchronisation-bisimulation, then they are in fact contained in a bisimulation in the usual sense. However, if two multisets are not contained in the coarsest either chain- or synchronisation-bisimulation, then it does not necessarily mean that they are not contained in the coarsest bisimulation (i.e. bisimilarity) in the usual sense – it can namely be the case that our procedure is not strong enough to identify them as bisimilar.

We do not currently analyse systems with rates. The reason is that states with enabled delay transitions cannot be merged in the same way as states with only instantaneous action transition enabled. Two different enabled action transitions do not influence each other, while the probabilities and expected durations of delay transitions depend on all the delay transitions enabled in a particular state (see Section 2.1.2). We could however first compute the coarsest chain- or synchronisation-bisimulation for a system with rates, not taking labels corresponding to rates into account during the computation, and then do some additional splitting based on transition rates between equivalence classes as in the usual algorithm for computing the coarsest bisimulation relations on IMC (see Table 4.5).

4.2 Bisimilar IMC systems

In this section we will introduce a definition of strong bisimulations on IMC systems. After this we will explain how this definition can be "transferred" in a straightforward way to label multisets that can be constructed based on the Pathway Analysis results of IMC^{G} systems and represent states reachable in the Structural Operational Semantics of IMC^{G}.

4.2.1 Definition and properties of strong bisimulations

The usual definition of a strong bisimulation on processes, which can also be regarded as states of a Labelled Transition System, is that it is a binary relation such that states matched by it can both do a transition decorated by the same action name and move after that transitions into states which are again matched by the strong bisimulation relation [Mil89]:

Definition 4.1. *A binary relation R on a set of states/processes that can only execute instantaneous actions is a strong bisimulation if for all $(E_1, E_2) \in \mathcal{R}$ holds:*

1. *if $E_1 \xrightarrow{\mathsf{a}} E_1'$ for some action a then there exists $E_2' \in \text{IMC}^{\text{G}}$ such that $E_2 \xrightarrow{\mathsf{a}} E_2'$ and $(E_1', E_2') \in \mathcal{R}$;*

2. *if $E_2 \xrightarrow{\mathsf{a}} E_2'$ for some action a then there exists $E_1' \in \text{IMC}^{\text{G}}$ such that $E_1 \xrightarrow{\mathsf{a}} E_1'$ and $(E_1', E_2') \in \mathcal{R}$;*

Note that strong bisimulation defined in this way is an equivalence relation. It is obviously reflexive, it is symmetric (follows from the two conditions in Definition 4.1), and it is easy to show that it is transitive.

This notion of bisimulation is however not directly applicable to IMC/IMC$^{\mathrm{G}}$ systems for several reasons. On the one hand, we need to take into account delay transitions in some way because, for example, $\underline{X := \mathsf{a}^{\ell_1}.X}$ and $\underline{X := \mathsf{a}^{\ell_2}.X} + \lambda^{\ell_3}.\mathbf{0}$ obviously should not be considered bisimilar. On the other hand, if we treat action- and delay-transitions in the same way, then we might be too restrictive. For example, we would not be able to equate the processes $\lambda_1^{\ell_1}.\mathsf{a}^{\ell_2}.\mathbf{0} + \lambda_2^{\ell_3}.\mathsf{a}^{\ell_4}.\mathbf{0}$ and $\lambda_3^{\ell_5}.\mathsf{a}^{\ell_6}.\mathbf{0}$ in case $\lambda_3 = \lambda_1 + \lambda_2$, while they have in fact the same behaviour according to the rules of exponential distribution. Therefore for delay transitions we cannot just compare transition labels, but rather need to compute *joint* or *cumulative* transition rates. At last, cumulative transition rates are only important if no internal transitions are possible, i.e. there are no enabled τ-transition. Thus, $\lambda_1^{\ell_1}.\mathsf{a}^{\ell_2}.\mathbf{0} + \lambda_2^{\ell_3}.\mathsf{a}^{\ell_4}.\mathbf{0} + \tau^{\ell_5}.\mathbf{0}$ and $\lambda_3^{\ell_6}.\mathsf{a}^{\ell_7}.\mathbf{0} + \tau^{\ell_1}.\mathbf{0}$ are bisimilar even if $\lambda_3 \neq \lambda_1 + \lambda_2$.

There is one more aspect that is not necessarily required for non-stochastic systems but is required for systems involving probability and time – we require, namely, that any bisimulation relation is an equivalence relation for such systems. This is due to the necessity to compute rates, probabilities, time to pass, etc. into sets of states, which are all pairwise contained in the bisimulation relation. If it is not possible to subdivide states into equivalence classes in case the bisimulation relation is not an equivalence relation, because it is not clear how to compute cumulative probabilities, rates, etc. into sets of bisimilar states in this case. Therefore, for example, probabilistic bisimulations have been introduced from the beginning as equivalence relations [LS89].

All these aspects have been taken into account by the definition of strong bisimulations on IMC systems by Hermanns in [Her02]. We directly adopt his definition for IMC$^{\mathrm{G}}$ systems in Definition 4.4. We will usually assume that all states in a strong bisimulation relation from Definition 4.4 are derivatives of some IMC$^{\mathrm{G}}$ program. The mentioned above cumulative transition rates are introduced in Definition 4.3. There are however some technical differences with the definition in [Her02] which make our definition even a bit simpler.

First, in most cases we just ignore the labelling of the transitions by chains as these are not relevant for the comparison of the behaviour of states – a chain that has been executed is not visible and does not influence either transition duration or the interaction with the environment. On the other hand, chains decorating transitions from some derivative expression of an IMC$^{\mathrm{G}}$ program are unique for each transition derivation (see Lemma 4.2) due to the uniqueness of exposed labels. Therefore we do not need to have a multiset of delay transitions

as it is necessary for IMC, but rather just a set of delays transition as we shall illustrate below.

Take the following example as an illustration. There are two different, i.e. involving different chains, transitions from the IMC$^{\mathrm{G}}$ process $\underline{X := \lambda^{\ell_1}.X + \lambda^{\ell_2}.X}$:

$$\underline{X := \lambda^{\ell_1}.X + \lambda^{\ell_2}.X} \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\lambda} \underline{X := \lambda^{\ell_1}.X + \lambda^{\ell_2}.X}$$

and

$$\underline{X := \lambda^{\ell_1}.X + \lambda^{\ell_2}.X} \xrightarrow[\perp_{\mathfrak{M}}[\ell_2 \mapsto 1]]{\lambda} \underline{X := \lambda^{\ell_1}.X + \lambda^{\ell_2}.X},$$

but there is only one transition

$$\underline{X := \lambda.X + \lambda.X} \xrightarrow{\lambda} \underline{X := \lambda.X + \lambda.X}$$

in IMC. In IMC$^{\mathrm{G}}$ we can distinguish between two delay transition based on the involved chains, while in IMC we need to have a multiset of transitions remembering that the transition from $\underline{X := \lambda.X + \lambda.X}$ to itself can be derived in two different ways from its syntax.

Lemma 4.2 (Transition derivation)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F, $F \xrightarrow{*} E$, *then there exists only one transition derivation for each transition* $E \xrightarrow[C]{\alpha} E'$.

Proof. The proof is by induction on the path length $F \xrightarrow{*} E$. For each number of steps the proof is by structural induction on E – in this way we are also proving the statement for all subexpressions of E. □

We will define in Definition 4.3 a so-called *cumulative rate*: it is in fact a delay transition rate between a state and a set of states. It is necessary for the definition of a strong bisimulation on IMC$^{\mathrm{G}}$ below.

Definition 4.3. *A* cumulative rate *between* $E \in \mathrm{IMC}^{\mathrm{G}}$ *and a set* $S \in 2^{\mathrm{IMC}^{\mathrm{G}}}$ *is denoted* $\gamma(E, S)$ *and is equal to* $\sum_{E \xrightarrow[C]{\lambda} E' \ st \ E' \in S} \lambda$. *An* outgoing rate *of* E *is a cumulative rate between* E *and all* IMC$^{\mathrm{G}}$ *expressions derivable from* E *in one step. It is denoted* $Rate(E)$ *and is equal to* $\sum_{E \xrightarrow[C]{\lambda} E'} \lambda$.

We are now ready to give a formal definition of a strong bisimulation relation on $\mathrm{IMC}^{\mathrm{G}}$ processes: it is an equivalence relation such that processes in this relation have the same enabled actions and the same cumulative rates (in case there are no enabled internal transitions) into the equivalence classes.

Definition 4.4. *An equivalence relation* $\mathcal{R} \subseteq \mathrm{IMC}^{\mathrm{G}} \times \mathrm{IMC}^{\mathrm{G}}$ *is a* strong bisimulation *relation if for all* $(E_1, E_2) \in \mathcal{R}$ *holds:*

1. *if* $E_1 \xrightarrow[C_1]{\mathsf{a}} E_1'$ *for some* $\mathsf{a} \in \boldsymbol{Act} \cup \{\tau\}$ *and* $C_1 \in \mathfrak{M}$ *then there exist* $E_2' \in \mathrm{IMC}^{\mathrm{G}}$ *and* $C_2 \in \mathfrak{M}$ *such that* $E_2 \xrightarrow[C_2]{\mathsf{a}} E_2'$ *and* $(E_1', E_2') \in \mathcal{R}$*;*
2. *if* $E_1 \overset{\tau}{\nrightarrow}$ *then* $\gamma(E_1, S) = \gamma(E_2, S)$ *for all* $S \in \mathrm{IMC}^{\mathrm{G}}/\mathcal{R}$*.*

Note that from $E_1 \overset{\tau}{\nrightarrow}$ and $(E_1, E_2) \in \mathcal{R}$ for some strong bisimulation relation $\mathcal{R}$ follows $E_2 \overset{\tau}{\nrightarrow}$, as $\mathcal{R}$ is defined to be an equivalence relation. Therefore either both equivalent in the sense of strong bisimulation $\mathrm{IMC}^{\mathrm{G}}$ processes have at least one τ-transition and their outgoing delay-transitions are not taken into account, or none of them has an outgoing τ-transition and their cumulative rates to sets of states constituting an equivalence class of $\mathcal{R}$ are equal for all equivalence classes.

The following two definitions are standard and have their direct correspondence in the definitions for IMC in [Her02]:

Definition 4.5. *Two* $\mathrm{IMC}^{\mathrm{G}}$ *processes* E_1 *and* E_2 *are strongly bisimilar, denoted* $E_1 \sim E_2$*, if they are contained in some strong bisimulation.*

Definition 4.6. *The coarsest strong bisimulation relation on* $\mathrm{IMC}^{\mathrm{G}} \times \mathrm{IMC}^{\mathrm{G}}$ *is called the* strong bisimilarity.

The next two lemmas are standard results for strong bisimilarities. The proofs are easy adaptations from the proofs in [Her02].

Lemma 4.7. *The strong bisimilarity on* $\mathrm{IMC}^{\mathrm{G}}$ *is:*

1. *an equivalence relation on* $\mathrm{IMC}^{\mathrm{G}}$*;*

2. a strong bisimulation on $\mathrm{IMC}^{\mathrm{G}}$*;*

3. the coarsest strong bisimulation on $\mathrm{IMC}^{\mathrm{G}}$*.*

Lemma 4.8. *The strong bisimilarity on* $\mathrm{IMC}^{\mathrm{G}}$ *is substitutive with respect to parallel composition and hiding operators, i.e. from* $E_1 \sim E_2$ *and* $E_1' \sim E_2'$ *follows* $E_1 \,|\!|_A\, E_1' \sim E_2 \,|\!|_A\, E_2'$ *and* **hide** A **in** $E_1 \sim$ **hide** A **in** E_2 *for all* $A \subseteq \boldsymbol{Act}$*.*

4.2.2 Definition of bisimulation relations on $\mathfrak{M}$

In this section we will adopt the definition of a strong bisimulation on $\mathrm{IMC}^{\mathrm{G}}$ to label multisets from $\mathfrak{M}$: corresponding relations on $\mathfrak{M}$ will be called simply "bisimulations". We will introduce a notation which will be used in the rest of the chapter and prove the link between strong bisimulations on $\mathrm{IMC}^{\mathrm{G}}$ and bisimulations on $\mathfrak{M}$.

In the discussion below we will assume that we have already conducted the Pathway Analysis of some $\mathrm{IMC}^{\mathrm{G}}$ program F. We will group the results of the Pathway Analysis and several additional functions that are using these results into a tuple and assign it to the function $\mathcal{F}$ as in Definition 4.9. In the definitions and proofs below we will often refer to the functions from the tuple $\mathcal{F}$ where the last has been computed for some fixed $\mathrm{IMC}^{\mathrm{G}}$ program F. This will simplify the notation a lot as we will not need to index the functions G, K, etc. from Table 4.1 with their corresponding F, and we will not need to denote each time in which way they have been computed.

Note that normally all labels and chains to which the operators $Name$, $Name_{ch}$ etc. from Table 4.1 will be applied will be labels or will contain only labels from the syntactic description of F, therefore the information on their corresponding action names or rates will be available through the operator $\Lambda_{\mathbf{Lab}}$. Also all multisets to which the operators $Name_{ch}$ and $Name_{ch}^{h}$ will be applied will be chains from $\mathfrak{T}$, therefore all their constituting labels will have the same corresponding action name or rate and it will be enough to check only one of the labels for its action name or rate.

Definition 4.9. *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F*, then we define a tuple*

$$\mathcal{F} = (G, K, G_{ch}, K_{ch}, \mathfrak{T}, Name, Name^{h}, Name_{ch}, Name_{ch}^{h}, \mathfrak{M}_F, \mathrm{IMC}_F^{\mathrm{G}}, parents)$$

using the definitions from Table 4.1.

$$
\begin{array}{rcl}
\Gamma & = & \Gamma_{\mathbf{Var}}[\![F]\!] \\
\Lambda & = & \Lambda_{\mathbf{Lab}}[\![F]\!] \\
G & = & \mathcal{G}_{\Gamma}[\![F]\!] \\
K & = & \mathcal{K}[\![F]\!] \\
G_{ch}(C) & = & \sum_{\ell \in dom(C)} G(\ell) \\
K_{ch}(C) & = & \sum_{\ell \in dom(C)} K(\ell) \\
\mathfrak{T} & = & \mathfrak{T}_{\Lambda}[\![F]\!] \\
Name(\ell) & = & \begin{cases} \alpha & \text{if } \exists \alpha \ st \ ((\ell,\alpha) \in \Lambda) \wedge (\nexists \beta \ st \ ((\ell,\beta) \in \Lambda) \wedge (\alpha \neq \beta)) \\ ? & \text{otherwise.} \end{cases} \\
Name^{h}(\ell) & = & \begin{cases} \tau & \text{if } Name(\ell) \in \mathbf{Act} \backslash \mathbf{fn}(F) \\ Name(\ell) & \text{otherwise.} \end{cases} \\
Name_{ch}(C) & = & \begin{cases} \alpha & \text{if } (|C| > 0) \wedge (\forall \ell \in dom(C) \ Name(\ell) = \alpha) \\ ? & \text{otherwise.} \end{cases} \\
Name^{h}_{ch}(C) & = & \begin{cases} \tau & \text{if } Name_{ch}(C) \in \mathbf{Act} \backslash \mathbf{fn}(F) \\ Name_{ch}(C) & \text{otherwise.} \end{cases} \\
\mathfrak{M}_{F} & = & \{M \in \mathfrak{M} | \exists E \in \mathrm{IMC}^{\mathrm{G}} \ st \ (F \xrightarrow{*} E) \wedge (M = \mathcal{E}_{\Gamma}[\![E]\!])\} \\
\mathrm{IMC}^{\mathrm{G}}_{F} & = & \{E \in \mathrm{IMC}^{\mathrm{G}} | F \xrightarrow{*} E\} \\
parents(\ell) & = & \{\ell' \in Labs(F) | \ell \in dom(G(\ell'))\}
\end{array}
$$

Table 4.1 – Definition of a number of operators on an $\mathrm{IMC}^{\mathrm{G}}$ program F, $\ell \in \mathbf{Lab}$, $C \in \mathfrak{M}$.

Assume therefore that an initial $\mathrm{IMC}^{\mathrm{G}}$ program F is fixed, we have performed the Pathway Analysis of F and saved the computed operators in the function $\mathcal{F}$ as in Definition 4.9. Then we can formulate a rather self-evident definition of a bisimulation relation on $\mathfrak{M}$ – see Definition 4.11. The multisets from $\mathfrak{M}$ that we are interested in are clearly only those that are equal to exposed multisets of $\mathrm{IMC}^{\mathrm{G}}$ expressions derivable from F. However we often do not know which expressions are derivable from F, so we accept bisimulation relations involving also unreachable states. It is clear that we can in any case exclude elements of $\mathfrak{M}$ that contain labels not occurring in F.

Analogously to the definitions of a cumulative rate, strong bisimulation and bisimilarity for $\mathrm{IMC}^{\mathrm{G}}$ processes in Section 4.2, we introduce the definitions of a cumulative rate, bisimulation and bisimilarity relations on the elements of $\mathfrak{M}$.

Definition 4.10. *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *and* $\mathcal{F}$ *as in Definition 4.9, then a* cumulative rate *between* $M \in \mathfrak{M}$ *and a set* $S \in 2^{\mathfrak{M}}$ *is denoted* $\gamma_{\mathfrak{M}}(M, S)$ *and is equal to*

$\sum_{\{C|(C\in\mathfrak{T})\wedge(C\leq M)\wedge(Name_{ch}(C)\in\textbf{\textit{Rate}})\wedge(M-K_{ch}(C)+G_{ch}(C)\in S)\}} Name_{ch}(C)$.

An outgoing rate *of M is a cumulative rate between M and all multisets derivable from it in one step. It is denoted $Rate_{\mathfrak{M}}(M)$ and is equal to*

$\sum_{\{C|(C\in\mathfrak{T})\wedge(C\leq M)\wedge(Name_{ch}(C)\in\textbf{\textit{Rate}})\}} Name_{ch}(C)$.

Definition 4.11. *Given an* IMC$^{\mathrm{G}}$ *program F and $\mathcal{F}$ as in Definition 4.9, then a* bisimulation *relation on $\mathfrak{M}$ is an equivalence relation $\mathcal{R} \subseteq \mathfrak{M} \times \mathfrak{M}$ such that for all $(M_1, M_2) \in \mathcal{R}$ holds:*

1. *for all $C_1 \in \mathfrak{T}$ and $M_1 \in \mathfrak{M}$ such that $C_1 \leq M_1$ and $M_1' = M_1 - K_{ch}(C_1) + G_{ch}(C_1)$ there exists $C_2 \in \mathfrak{T}$ and $M_2 \in \mathfrak{M}$ such that $C_2 \leq M_2$, $Name^h_{ch}(C_1) = Name^h_{ch}(C_2)$, $M_2' = M_2 - K_{ch}(C_2) + G_{ch}(C_2)$ and $(M_1', M_2') \in \mathcal{R}$;*
2. *if $\nexists C \in \mathfrak{T}$ such that $C \leq M_1$ and $Name^h_{ch}(C) = \tau$ then $\gamma_{\mathfrak{M}}(M_1, S) = \gamma_{\mathfrak{M}}(M_2, S)$ for all equivalence classes $S \in \mathfrak{M}/\mathcal{R}$.*

Definition 4.12. *Given an* IMC$^{\mathrm{G}}$ *program F and $\mathcal{F}$ as in Definition 4.9, then two elements $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$ are bisimilar, denoted $M_1 \sim M_2$, if they are contained in some bisimulation.*

Definition 4.13. *Given an* IMC$^{\mathrm{G}}$ *program F and $\mathcal{F}$ as in Definition 4.9, then the coarsest bisimulation relation on $\mathfrak{M} \times \mathfrak{M}$ is called* bisimilarity *on $\mathfrak{M}$.*

Bisimulations and bisimilarities in Definitions 4.12 and 4.13 are defined on the whole $\mathfrak{M}$ as it is often the case that we do not know which multisets from $\mathfrak{M}$ are reachable from the exposed labels of F. Also note that there are not only many different bisimulations but also many different bisimilarities on $\mathfrak{M}$ – namely, depending on F for which the Pathway Analysis has been performed.

We prove in Lemma 4.14 below that there is a direct correspondence between strong bisimulations/the bisimilarity on IMC$^{\mathrm{G}}$ on the one hand and strong bisimulations/bisimilarities on $\mathfrak{M}$ on the other hand. Note that we are constraining strong bisimulations on IMC$^{\mathrm{G}}$ and bisimulations on $\mathfrak{M}$ only to states reachable accordingly from an IMC$^{\mathrm{G}}$ program F or from its exposed labels. This is because we can only talk about bisimulations on $\mathfrak{M}$ in the context of some initial IMC$^{\mathrm{G}}$ program.

Lemma 4.14 (Bisimulation on $\mathfrak{M}$ and strong bisimulation)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *and* $\mathcal{F}$ *as in Definition 4.9, then the following statements hold:*

1. *if* $\mathcal{R} \in \mathfrak{M}_F \times \mathfrak{M}_F$ *is a bisimulation relation on* $\mathfrak{M}$ *then the induced relation* $\mathcal{R}' \in \mathrm{IMC}_F^{\mathrm{G}} \times \mathrm{IMC}_F^{\mathrm{G}}$ *defined as* $\mathcal{R}' = \{(E_1, E_2) | (\mathcal{E}_\Gamma[\![E_1]\!], \mathcal{E}_\Gamma[\![E_2]\!]) \in \mathcal{R}\}$ *is a strong bisimulation relation on* $\mathrm{IMC}^{\mathrm{G}}$*;*
2. *if a relation* $\mathcal{R} \in \mathrm{IMC}_F^{\mathrm{G}} \times \mathrm{IMC}_F^{\mathrm{G}}$ *is a strong bisimulation relation on* $\mathrm{IMC}^{\mathrm{G}}$ *then the induced relation* $\mathcal{R}' \in \mathfrak{M}_F \times \mathfrak{M}_F$ *defined as* $\mathcal{R}' = \{(\mathcal{E}_\Gamma[\![E_1]\!], \mathcal{E}_\Gamma[\![E_2]\!]) | (E_1, E_2) \in \mathcal{R}\}$ *is a bisimulation relation on* $\mathfrak{M}$*;*
3. *if a relation* $\mathcal{R} \in \mathfrak{M}_F \times \mathfrak{M}_F$ *is the bisimilarity relation on* $\mathfrak{M}$ *constrained to* $\mathfrak{M}_F$ *then the induced relation* $\mathcal{R}' \in \mathrm{IMC}_F^{\mathrm{G}} \times \mathrm{IMC}_F^{\mathrm{G}}$ *defined as* $\mathcal{R}' = \{(E_1, E_2) | (\mathcal{E}_\Gamma[\![E_1]\!], \mathcal{E}_\Gamma[\![E_2]\!]) \in \mathcal{R}\}$ *is the strong bisimilarity relation on* $\mathrm{IMC}^{\mathrm{G}}$ *constrained to* $\mathrm{IMC}_F^{\mathrm{G}}$*;*
4. *if a relation* $\mathcal{R} \in \mathrm{IMC}_F^{\mathrm{G}} \times \mathrm{IMC}_F^{\mathrm{G}}$ *is the strong bisimilarity relation on* $\mathrm{IMC}^{\mathrm{G}}$ *constrained to* $\mathrm{IMC}_F^{\mathrm{G}}$ *then the induced relation* $\mathcal{R}' \in \mathfrak{M}_F \times \mathfrak{M}_F$ *defined as* $\mathcal{R}' = \{(\mathcal{E}_\Gamma[\![E_1]\!], \mathcal{E}_\Gamma[\![E_2]\!]) | (E_1, E_2) \in \mathcal{R}\}$ *is the bisimilarity relation on* $\mathfrak{M}$ *constrained to* $\mathfrak{M}_F$*.*

Proof. The proof is based on the fact that for all $E_1 \in \mathrm{IMC}_F^{\mathrm{G}}$ all derivations of transitions from E_1 have one-to-one correspondence with the enabled chains among the exposed labels of E_1. I.e. for each $E_1 \xrightarrow[C_1]{} E_2$, $E_1 \xrightarrow[C_2]{} E_2'$ such that $E_2 \neq E_2'$ holds $C_1 \neq C_2$. Let us prove this statement by proving an equivalent statement: for all transitions $E \xrightarrow[C]{} E'$, $E_1 \xrightarrow[C]{} E''$ such that E is an $\mathrm{IMC}^{\mathrm{G}}$ expression and the exposed labels of E are pairwise different, it holds that $E' = E''$.

We prove this by induction on the transition derivation based on the SOS rules for $\mathrm{IMC}^{\mathrm{G}}$ in Table 3.5. The base cases are rules (1) and (10) in Table 3.5 and the statement clearly holds for them because there is only one possible enabled chain and one derivation. For the rest of the rules the statement follows from the induction hypothesis and from the uniqueness of exposed labels besides may be the rules (9) and (16). It is however the case that we can apply the induction hypothesis for $E\{\underline{X := E}/X\}$ instead of $\underline{X := E}$ (because the transition derivation for $E\{\underline{X := E}/X\}$ is smaller) as exposed labels for $E\{\underline{X := E}/X\}$ are pairwise different according to Lemma 3.6, therefore the induction hypothesis is applicable. As a whole we can deduce that for $E\{\underline{X := E}/X\} \xrightarrow[C]{} E'$ and $E\{\underline{X := E}/X\} \xrightarrow[C]{} E''$ holds $E' = E''$, therefore also for $\underline{X := E} \xrightarrow[C]{} E'$ and $\underline{X := E} \xrightarrow[C]{} E''$ holds $E' = E''$.

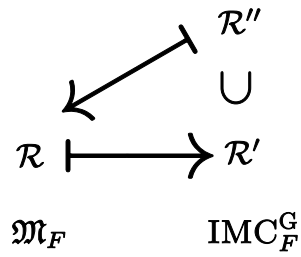

Figure 4.1 – Graphical illustration for the proof of Lemma 4.14: a bisimilarity on $\mathfrak{M}_F$ is the strong bisimilarity on $\mathrm{IMC}_F^{\mathrm{G}}$ because the presented situation is impossible.

It can also be easily proved that for all transitions $E_1 \xrightarrow[C_1]{} E_2$ and $E_1 \xrightarrow[C_2]{} E_2$ with the same transition derivation holds $C_1 = C_2$. Now, returning to proving the statements of the lemma, this establishes that we have a one-to-one correspondence between the transitions in $\mathrm{IMC}_F^{\mathrm{G}}$ and the transitions in $\mathfrak{M}_F$, therefore any bisimulation on $\mathrm{IMC}_F^{\mathrm{G}}$ is a bisimulation on $\mathfrak{M}_F$ and the other way round.

The coarsest bisimulation (i.e. bisimilarity) $\mathcal{R}$ on $\mathfrak{M}_F$ corresponds to the coarsest strong bisimulation (i.e. strong bisimilarity) $\mathcal{R}'$ on $\mathrm{IMC}_F^{\mathrm{G}}$ because it is a bisimulation according to the reasoning above, and if there would exist a coarser bisimulation $\mathcal{R}'' \supset \mathcal{R}'$ on $\mathrm{IMC}_F^{\mathrm{G}}$, then $\mathcal{R}''$ would have to correspond to $\mathcal{R}$. Schematically the situation is presented in Figure 4.1. This situation is however impossible as for any $(E, E') \in \mathcal{R}''$ for which $(\mathcal{E}_\Gamma[\![E]\!], \mathcal{E}_\Gamma[\![E']\!]) \in \mathcal{R}$ holds $(E, E') \in \mathcal{R}'$ as well.

On the other hand, the strong bisimilarity on $\mathrm{IMC}_F^{\mathrm{G}}$ corresponds to the bisimilarity on $\mathfrak{M}_F$ because the situation presented in Figure 4.2 is also impossible. For any $(M_1, M_2) \in \mathcal{R}''$ such that all $(E_1, E_2) \in R$ if $\mathcal{E}_\Gamma[\![E_1]\!] = M_1$ and $\mathcal{E}_\Gamma[\![E_2]\!] = M_2$ it obviously holds $(M_1, M_2) \in \mathcal{R}'$. □

Note that in the proof of Lemma 4.14 we do not necessarily have a one-to-one correspondence between the elements in $\mathfrak{M}_F$ and the elements in $\mathrm{IMC}_F^{\mathrm{G}}$: the correspondence can be between the elements in $\mathfrak{M}_F$ and the sets of elements from $\mathrm{IMC}_F^{\mathrm{G}}$. For example, two $\mathrm{IMC}^{\mathrm{G}}$ expressions

$$\underline{Y := X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.Y}$$

and the expression derived from it after one transition

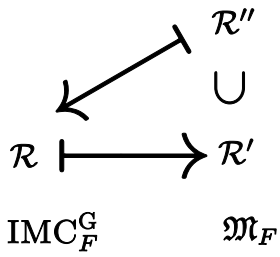

Figure 4.2 – Graphical illustration for the proof of Lemma 4.14: the strong bisimilarity on $\mathrm{IMC}_F^{\mathrm{G}}$ is a bisimilarity on $\mathfrak{M}_F$ because the presented situation is impossible.

$$\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.Y := \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.Y}}$$

have the same exposed labels, but are syntactically different. They both correspond however to the multiset of exposed labels $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$. There is still a strict one-to-one correspondence between the elements in $\mathfrak{M}_F$ and the sets of elements in $\mathrm{IMC}_F^{\mathrm{G}}$ and therefore Lemma 4.14 holds. Basically we are just simplifying things with mapping several $\mathrm{IMC}^{\mathrm{G}}$ expressions with essentially the same semantics that differ syntactically only due to the different number of process definitions in their syntax to one multiset of exposed labels.

4.2.3 Non-additivity of bisimilarity on $\mathfrak{M}$

Let us look more attentively on the properties of bisimulation relations on multisets. First of all we will find out that if two multisets are pairwise bisimilar to some other two multisets then it does not necessarily mean that their sums are also bisimilar:

$$\left.\begin{array}{l} M_1 \sim M_1' \\ M_2 \sim M_2' \end{array}\right\} \not\Rightarrow M_1 + M_2 \sim M_1' + M_2'$$

A counterexample is $F = \mathsf{a}^{\ell_1}.\mathbf{0} \between \mathsf{a}^{\ell_2}.\mathbf{0} + \mathsf{a}^{\ell_3}.\mathbf{0} \between \mathsf{a}^{\ell_4}.\mathbf{0} \between \mathsf{a}^{\ell_5}.\mathbf{0}$, with $M_1 = \bot_{\mathfrak{M}}[\ell_1 \mapsto 1]$, $M_2 = \bot_{\mathfrak{M}}[\ell_2 \mapsto 1]$, $M_1' = \bot_{\mathfrak{M}}[\ell_3 \mapsto 1]$ and $M_2' = \bot_{\mathfrak{M}}[\ell_4 \mapsto 1]$. Obviously $M_1 \sim M_1'$ and $M_2 \sim M_2'$ hold because there are no enabled chains in any of the multisets. However $M_1 + M_2 \not\sim M_1' + M_2'$ because $M_1 + M_2 =$

$\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ and there is an enabled chain in $M_1 + M_2$ while $M_1' + M_2' = \bot_{\mathfrak{M}}[\ell_3 \mapsto 1, \ell_4 \mapsto 1]$ and there is no enabled chain in $M_1' + M_2'$.

We can easily find a counterexample with minus instead of plus for which a bisimulation relation on $\mathfrak{M}$ is also not compositional. It is therefore no surprise that the effect of the execution of bisimilar chains on bisimilar multisets does not necessarily result in bisimilar multisets:

$$\left.\begin{array}{r} M_1 \sim M_2 \\ K_{ch}(C_1) \sim K_{ch}(C_2) \\ G_{ch}(C_1) \sim G_{ch}(C_2) \end{array}\right\} \not\Rightarrow M_1 - K_{ch}(C_1) + G_{ch}(C_1) \sim M_2 - K_{ch}(C_2) + G_{ch}(C_2)$$

We can use the following counterexample: $(\mathsf{c}^{\ell_1}.\underline{X := \mathsf{a}^{\ell_2}.X}) \parallel \mathsf{a} \parallel \underline{Y := \mathsf{b}^{\ell_3}.Y} + (\mathsf{c}^{\ell_4}.\underline{Z := \mathsf{a}^{\ell_5}.Z}) \parallel \emptyset \parallel \underline{W := \mathsf{b}^{\ell_6}.W}$, $M_1 = \bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_3 \mapsto 1, \ell_4 \mapsto 1, \ell_6 \mapsto 1]$, $M_2 = M_1$, $C_1 = \bot_{\mathfrak{M}}[\ell_1 \mapsto 1]$, $C_2 = \bot_{\mathfrak{M}}[\ell_4 \mapsto 1]$. It is easy to see that $K_{ch}(C_1) \sim K_{ch}(C_2)$ and $G_{ch}(C_1) \sim G_{ch}(C_2)$: compare, for example, $G_{ch}(C_1) = \bot_{\mathfrak{M}}[\ell_2 \mapsto 1]$ and $G_{ch}(C_2) = \bot_{\mathfrak{M}}[\ell_5 \mapsto 1]$. The execution of C_1 will lead to the $\mathrm{IMC}^{\mathrm{G}}$ process $\underline{X := \mathsf{a}^{\ell_2}.X} \parallel \mathsf{a} \parallel \underline{Y := \mathsf{b}^{\ell_3}.Y}$ while the execution of C_2 will lead to the $\mathrm{IMC}^{\mathrm{G}}$ process $\underline{Z := \mathsf{a}^{\ell_5}.Z} \parallel \emptyset \parallel \underline{W := \mathsf{b}^{\ell_6}.W}$ – it is clear that these two processes are not bisimilar as the first cannot execute the action a and the second can.

Note that the counterexamples that we have presented do not even involve rates but are rather based on blocking some labels by the environment in one case and not blocking the labels in the other case. Some kind of information is needed therefore about the chains in which labels that constitute multisets in question do not participate currently but might participate in the future. Also the information about the chains in which labels that can be generated in the future can participate should be preserved. We will go into more details of this discussion while presenting below two bisimulation relations defined by us which use the information on chains.

The discussion above has illustrated why we cannot use the following seemingly efficient strategy for determining if two multisets are bisimilar (assume for simplicity of illustration that there are no delay transitions):

1. determine whether each chain C_1 in the first multiset has a corresponding chain C_2 in the second multiset with the same action name;
2. determine whether $K(C_1) \sim K(C_2)$ and $G(C_1) \sim G(C_2)$.

It is also clear that we can execute two non-bisimilar chains, with non-bisimilar kill and generate label multisets, on two non-bisimilar multisets and arrive at bisimilar states: for example, in $(\mathsf{a}^{\ell_1}.\underline{X := \mathsf{b}^{\ell_2}.X}) \shuffle \mathsf{b} \shuffle \mathbf{0}$ and $\mathsf{b}^{\ell_3}.\mathbf{0}$ the execution of the labels ℓ_1 and ℓ_3 will in both cases lead to transitions to the terminal states, which are bisimilar.

"Non-additivity" of bisimilar multisets does not allow for much improvements in the algorithms for constructing bisimilarities in the general case: the algorithm that we will present in Section 4.2.4 is a direct adaptation of the algorithm from [Her02]. In Sections 4.3 and 4.4 we will however discuss how we can exploit the knowledge about the "internal structure" in multisets, i.e. the knowledge about labels' participation in chains, in order to devise more efficient algorithms.

4.2.4 Algorithm for computing bisimulations on $\mathfrak{M}$

In this section we will shortly present an algorithm for computing the coarsest bisimulation relation on a set of label multisets. As usual, we assume that an IMC$^{\mathrm{G}}$ program F is given, the Pathway Analysis has been conducted on it and the results have been saved in $\mathcal{F}$ as in Definition 4.9.

The multisets of labels in the set *states* on which the bisimulation algorithm will be executed may either be equal to exposed labels of the derivatives of F, i.e. $states = \{M \in \mathfrak{M} | \exists E \in \mathrm{IMC}^{\mathrm{G}}\ st\ (F \xrightarrow{*} E) \wedge (M = \mathcal{E}_\Gamma[\![E]\!])\}$, or they may be just all the multisets of labels whose domains contain only labels from $Labs(F)$, i.e. $states = \{M \in \mathfrak{M} | \exists \ell_1 ... \ell_n\ st\ (\bigwedge_{i=1..n}(\ell_i \in Labs(F))) \wedge (M = \sum_{i=1..n} \bot_{\mathfrak{M}} [\ell_i \mapsto 1])\}$ (this is clearly an easily computable safe over-approximation of states reachable from $\mathcal{E}_\Gamma[\![F]\!]$), or something in between. In Section 5.1 in the next chapter we will discuss how some of the states can be ruled out as unreachable without computing the whole semantics of F.

The algorithm for computing the coarsest bisimulation on *states* is essentially the same as the algorithm presented in [Her02] for computing the coarsest strong bisimulation on IMC systems. The advantage of our method is, first, that we do not need to derive transitions from the syntax of IMC$^{\mathrm{G}}$ expressions (going through the whole syntax in the worst case) but rather only compare exposed labels with chains. This is a general advantage of Pathway Analysis methods. The second advantage is that we do not need to build the semantics of IMC$^{\mathrm{G}}$ if *states* is not computed as a set of reachable states or at least do not need to save transitions between states – this leads to a lower space complexity. We can use BDDs in order to represent sets of states – it is straightforward as for each label there is a binary value, i.e. it is present or not present in a multiset.

$$
\begin{array}{rcl}
names(M) & = & \bigcup_{\{C\in\mathfrak{T}|C\leq M\}}\{Name^h_{ch}(C)|Name^h_{ch}(C)\in \mathbf{Act}\cup\{\tau\}\} \\
names\_set(S) & = & \bigcup_{M\in S}\{names(M)\} \\
rate\_set(S) & = & \bigcup_{M\in S}\{Rate_{\mathfrak{M}}(M)\}
\end{array}
$$

Table 4.2 – Definitions of the functions $names : \mathfrak{M} \to 2^{\mathbf{Act}\cup\{\tau\}}$, $names\_set : 2^{\mathfrak{M}} \to 2^{\mathbf{Act}\cup\{\tau\}}$, $rate\_set : 2^{\mathfrak{M}} \to 2^{\mathbb{R}^+_0}$, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9, $M \in \mathfrak{M}$, $S \in 2^{\mathfrak{M}}$.

The algorithm is based on the idea of *partition refinement*. It is a combination of the algorithms for strong bisimulation and Markovian bisimulation. It has a time complexity of $O((mI+mM)*log(n))$ where mI is the number of interactive transitions, mM the number of Markovian transitions and n is the number of states, as the original algorithm in [Her02]. The space complexity of the original algorithm is $O(mI + mM)$. The space complexity of our algorithm is $O(n)$ as we only save states – the existence of transitions is then checked by comparing exposed labels with precomputed chains. However the encoding of one state in our algorithm requires in general more space than in the original algorithm in [Her02] because we are encoding all the exposed labels of a state in the state name.

The algorithm consists of two main steps:

1. create an initial partition of states such that all states in one partition class have the same outgoing rate and the same set of action names decorating their enabled action transitions;

2. refine the partition according to the differences between the elements constituting partition classes in their cumulative rates into the other partition classes and in the presence/absence of action transitions decorated by fixed action names into the other partition classes until no further refinement is possible.

We define several auxiliary functions in Table 4.2. As usual, we assume that an initial IMC$^{\mathrm{G}}$ program F is fixed and the Pathway Analysis has been conducted on it. The fist function *names* in Table 4.2 has a multiset M as an input and returns all the action names that correspond to enabled chains in M. The functions *names_set* and *rate_set* have as their input a set of label multisets and return accordingly a set of action names of enabled action transitions in all the label multisets (*names_set*) or a set of all the outgoing rates of label multisets (*rate_set*).

$s_0 := \mathcal{E}_\Gamma[\![F]\!]$; $\pi_0 := \{s_0\}$;
$Part := \{\pi_0\}$; $states := \{s_0\}$; $WS := \{s_0\}$;
while $WS \neq \emptyset$ do
  choose $s \in WS$
  $WS := WS \backslash \{s\}$;
  if $\{C \in chains(s) | Name^h_{ch}(C) = \tau\} \neq \emptyset$
    then $Chains := \bigcup_{\{C' \in chains(s) | Name^h_{ch}(C') \in \mathbf{Act} \cup \{\tau\}\}} \{C'\}$;
    else $Chains := chains(s)$;
  for all $C \in Chains$
    $s' := s - K_{ch}(C) + G_{ch}(C)$;
    if $s' \notin states$ then
      $WS := WS \cup \{s'\}$;
      flag := *false*;
      for all $\pi \in Part$
        if $(names(s') = names\_set(\pi)) \wedge$
          $(\{Rate_{\mathfrak{M}}(s')\} = rate\_set(\pi))$
          then $\pi := \pi \cup \{s'\}$;
          flag := *true*;
      if (*flag* = *false*)
        then $\pi_{|Parts|+1} := \{s'\}$; $Part := Part \cup \pi_{|Parts|+1}$;
Return *Part*

Table 4.3 – Algorithm for computing the initial partition of states which is used by the algorithm determining the coarsest bisimulation relation on states, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9. Algorithm is taken from [Her02].

Assume that the set *states* contain only those label multisets that are reachable from $\mathcal{E}_\Gamma[\![F]\!]$. Then creating the initial partition can be done by the algorithm in Table 4.3 which groups together states with the same outgoing rate and the same action names decorating their outgoing transitions. The second part of the algorithm, i.e. the refinement of partition classes, is essentially the same algorithm as the one devised by Hermanns in [Her02]. We also sometimes use the notation from [Her02].

At the beginning of the refinement algorithm in Table 4.5 we divide the partition classes into "stable", i.e. with no enabled internal actions (S_Part), and "unstable", with at least one enabled internal action (U_Part). Unstable partition classes are refined only according to their action-transitions, while stable partition classes are refined both according to their action-transitions and according to their delay-transitions. Rate-transitions are irrelevant for unstable partition classes because according to the internal progress assumption no de-

$$
\begin{array}{rcl}
deriv(s,\alpha) & = & \bigcup_{\{C\in\mathfrak{T}|(C\leq s)\wedge(Name^{h}_{ch}(C)=\alpha)\}}\{s-K_{ch}(C)+G_{ch}(C)\} \\
p\_deriv(\pi_1,\alpha,\pi_2) & = & \{\bigcup_{\{s\in\pi_1|deriv(s,\alpha)\cap\pi_2\neq\emptyset\}}\{s\}\}\cup \\
 & & \{\bigcup_{\{s\in\pi_1|deriv(s,\alpha)\cap\pi_2=\emptyset\}}\{s\}\} \\
Refine(Part,\alpha,\pi) & = & (\bigcup_{\pi'\in Part} p\_deriv(\pi',\alpha,\pi))\backslash\emptyset \\
M\_Refine(Part,\pi) & = & (\bigcup_{\pi'\in Part}\bigcup_{r\in\mathbb{R}_0^+}\{\{s\in\pi'|\gamma_{\mathfrak{M}}(s,\pi)=r\}\})\backslash\emptyset
\end{array}
$$

Table 4.4 – Definitions of the functions *deriv*, *p_deriv*, *Refine*, *M_Refine*, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9. Definitions are taken from [Her02].

lay transition is possible from a state that has an enabled internal transition [Her02].

We define several auxiliary functions in Table 4.4. The function *deriv* returns for a multiset and an action name all the multisets that can be reached in one step from the input multiset by performing a transition decorated by the action name. The function *p_deriv* splits a partition class which is its first parameter according to the presence or absence of transitions from it into another partition class (third parameter) decorated by the action name specified by the second input parameter. The function $Refine$ uses the function *p_deriv* in order to split all partition classes in the current partition. The function M_Refine splits partition classes according to their cumulative transition rate into the partition class given as the second input parameter.

Using the functions defined in Table 4.4, we can write down the partition splitting algorithm – see Table 4.5.

Lemma 4.15 has its direct analogue in [Her02] and can be proved in a similar way:

Lemma 4.15 (Bisimilarity computation)**.** *Given an* IMC$^{\mathrm{G}}$ *program F and $\mathcal{F}$ as in Definition 4.9, then after the initialisation phase $Part$ is coarser then any bisimulation on $\{M|\exists E\ st\ (F \xrightarrow{*} E)\wedge(M=\mathcal{E}_\Gamma[\![E]\!])\}$. After each loop run of the splitting algorithm the partition $Part$ is coarser than any bisimulation on $\{M|\exists E\ st\ (F \xrightarrow{*} E)\wedge(M=\mathcal{E}_\Gamma[\![E]\!])\}$ and the algorithm returns the coarsest bisimulation.*

Proof. See [Her02]. □

$S\_Part := \{\pi \in Part | \nexists C \in chains(\pi)\ st\ Name^h_{ch}(C) = \tau\}$;
$U\_Part := Part \backslash S\_Part$;
$Spl := (fn(F) \cup \{\tau\}) \times (S\_Part \cup U\_Part)$
while $Spl \neq \emptyset$ do
  choose $(\alpha, \pi) \in Spl$
  $Old := S\_Part \cup U\_Part$;
  $S\_Part := Refine(S\_Part, \alpha, \pi)$;
  $U\_Part := Refine(U\_Part, \alpha, \pi)$;
  $S\_Part := M\_Refine(S\_Part, \pi)$;
  $New := (S\_Part \cup U\_Part) \backslash Old$
  $Spl := Spl \backslash \{(\alpha, \pi)\} \cup (fn(F) \cup \{\tau\}) \times New$
Return $S\_Part \cup U\_Part$

Table 4.5 – Algorithm for partition refinement with the purpose of determining the coarsest bisimulation relation on states, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9. Algorithm is taken from [Her02].

Altogether we have shown how the coarsest bisimulation relation can be computed on label multisets which are reachable from the multiset of exposed labels of an IMC$^{\mathrm{G}}$ program F on which the Pathway Analysis has been conducted.

4.3 Chain-bisimulations

4.3.1 Constructing bisimulations with Pathway Analysis

As we have pointed out in Section 4.2.4, computing the coarsest bisimulation relation on label multisets has the same time complexity as computing the coarsest strong bisimulation on the states of the labelled transition system representing the semantics of an IMC$^{\mathrm{G}}$ program. In this section we will explore the possibilities of determining some kind of equivalences firstly on labels, and then, using the gathered information, – on label multisets. In this way we can potentially devise algorithms for computing bisimulations with a lower complexity, ideally in the size of syntax – much in spirit of the Pathway Analysis in general. The downside of this approach is that we will compute in general not the coarsest possible bisimulation but a strictly smaller relation, as we will be partially missing information about the context in which the labels are executed. There is however a possibility that for some subclasses of IMC$^{\mathrm{G}}$ we can achieve the computation of the coarsest bisimulation with our methods. We leave this question

as a topic for future work.

In the following we will define two interdependent binary relations, one on labels and one on multisets, called accordingly a *label chain-bisimulation* and a *multiset chain-bisimulation*. The interdependence means that the relations cannot be defined separately but only in pair. In order to determine whether an arbitrary binary relation, for example, on multisets satisfies our definition we will need to know its "counterpart" on labels. The relations will depend on a fixed IMC^{G} program F for which the Pathway Analysis has been conducted.

Label chain-bisimulations will be relations on all labels from $Labs(F)$ and multiset chain-bisimulations will be relations on all multisets that contain labels from $Labs(F)$ – in this way we avoid computing the whole finite automaton representing Structural Operational Semantics of F before computing the relations. We will prove that every multiset chain-bisimulation is a bisimulation. The idea is thus first to compute a bisimulation relation on the syntax of F and then to merge bisimilar states "on the fly" while constructing the labelled transition system representing the semantics of F. Our approach has therefore several potential benefits, including low time and space complexities. As we do not compute the coarsest bisimulation in general, we will recognise some pairs of bisimilar states but not all of them.

For now we only consider IMC^{G} programs without rates. Multiset chain-bisimulations are compositional (i.e. additive), therefore dealing with conditions for bisimilar states and proving that we have in fact constructed bisimulations is much easier if we only have to consider actions. Dealing with rates is harder both because we have to consider *all* transition rates in a state and because we may need to take into account the maximal progress assumption, i.e. that no delay transitions can be executed if an internal action is enabled. We leave therefore considering rates in computing bisimulations as a topic for future work.

The coarsest label chain-bisimulation and its corresponding coarsest multiset chain-bisimulation will be called the label chain-bisimilarity (denoted $\sim^{ch}_{\mathbf{Lab}}$) and the multiset chain-bisimilarity (denoted $\sim^{ch}_{\mathfrak{M}}$). Computing the coarsest relations is preferable as we will be able to match a bigger number of bisimilar states with them. Labels contained in the label chain-bisimilarity will be called *label chain-bisimilar* and multisets contained in the multiset chain-bisimilarity will be called *multiset chain-bisimilar*.

In a sense chain-bisimulation equivalent labels do not only "behave in the same way", but moreover "cooperate" (i.e. participate in the same chains) with other labels following the same pattern. Two multisets are in the relation $\sim^{ch}_{\mathfrak{M}}$ and therefore bisimilar if their participating labels are either in the relation $\sim^{ch}_{\mathbf{Lab}}$ with each other or do not participate in any chain, therefore they can never

be executed and do not contribute to the behaviour of the multisets containing them.

Even though we are always talking about computing bisimulation relations on one particular IMC$^{\mathrm{G}}$ program, it is easy to see that we can compute the relation $\sim^{ch}_{\mathfrak{M}}$ on several IMC$^{\mathrm{G}}$ programs, by ensuring that their labelling is mutually disjoint. In particular, if we have two different programs F_1 and F_2, then we can conduct the Pathway Analysis on $F = \tau^{\ell_1}.F_1 + \tau^{\ell_2}.F_2$, if $\ell_1 \notin Labs(F_1) \cup Labs(F_2)$, $\ell_2 \notin Labs(F_1) \cup Labs(F_2)$ and $Labs(F_1) \cap Labs(F_2) = \emptyset$. Both F_1 and F_2 will be then derivatives of F and their generate, kill and chains functions will be included in the corresponding functions of F.

We can then obtain an answer "yes" or "no" concerning the question whether two particular IMC$^{\mathrm{G}}$ processes are bisimilar by checking whether the multisets of labels describing their initial configurations (in the example above these are exposed labels of accordingly F_1 and F_2) are in $\sim^{ch}_{\mathfrak{M}}$. However, as $\sim^{ch}_{\mathfrak{M}}$ is in general not the coarsest bisimulation relation on the multisets of labels representing derivatives of an IMC$^{\mathrm{G}}$ program, the answer "no" does not necessarily mean that two IMC$^{\mathrm{G}}$ programs are not bisimilar. For example, the IMC$^{\mathrm{G}}$ processes $\mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_1}.\mathbf{0} \,{}_{||}\{a, b\}_{||}\, \mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_2}.\mathbf{0}$ and $\mathsf{a}^{\ell_3}.\mathsf{b}^{\ell_3}.\mathbf{0}$ will not be in the relation $\sim^{ch}_{\mathfrak{M}}$, because chains with different number of components cannot be identified by $\sim^{ch}_{\mathfrak{M}}$. The bisimulation relations devised by us are still useful in order to reduce the state space of IMC$^{\mathrm{G}}$ systems and thus to make the further verifications easier.

4.3.2 Label and multiset chain-bisimulations

We will define label chain-bisimulation relations on $\mathbf{Lab} \times \mathbf{Lab}$ and multiset chain-bisimulation relations on $\mathfrak{M} \times \mathfrak{M}$. These relations always exist in pairs, i.e. the definition is common for a pair of such relations. They are always defined relative to the Pathway Analysis results $\mathcal{F}$ as in Definition 4.9 of a particular IMC$^{\mathrm{G}}$ program F without rates. In the following we will always assume F to be fixed and its analysis to be conducted previously to computing further operators or relations on F.

Definition 4.16. *Given an* IMC$^{\mathrm{G}}$ *program F without rates and $\mathcal{F}$ as in Definition 4.9, we call a binary relation R on labels and a binary relation R' on multisets accordingly a* label chain-bisimulation *and a* multiset chain-bisimulation *if the conditions (A1)-(A4) and (B1)-(B3) hold.*

If $\ell_1 R \ell_2$ then:

(A1) both labels are active, i.e. there exist $C_1 \in \mathfrak{T}$ *and* $C_2 \in \mathfrak{T}$ *such that* $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1] \leq C_1$ *and* $\bot_{\mathfrak{M}}[\ell_2 \mapsto 1] \leq C_2$*;*

(A2) $Name^h(\ell_1) = Name^h(\ell_2)$*;*

(A3) for all $C \in \mathfrak{T}$ *holds either*

- *(a)* $\ell_1 \in dom(C)$ *iff* $\ell_2 \in dom(C)$ *or*
- *(b) if* $\ell_1 \in dom(C)$ *then* $C - \bot_{\mathfrak{M}}[\ell_1 \mapsto 1] + \bot_{\mathfrak{M}}[\ell_2 \mapsto 1] \in \mathfrak{T}$*, if* $\ell_2 \in dom(C)$ *then* $C - \bot_{\mathfrak{M}}[\ell_2 \mapsto 1] + \bot_{\mathfrak{M}}[\ell_1 \mapsto 1] \in \mathfrak{T}$*;*

(A4) $G(\ell_1) R' G(\ell_2)$*.*

If $M_1 R' M_2$ *then there exist sums* $M_1 = \sum_{i=1..n} M_1^i + M_1'$ *and* $M_2 = \sum_{i=1..n} M_2^i + M_2'$ *such that*

(B1) all labels $\ell \in dom(M_1') \cup dom(M_2')$ *are not active;*

(B2) for all $1 \leq i \leq n$*,* $\ell_1 \in dom(M_1^i)$*,* $\ell_2 \in dom(M_2^i)$ *holds:* $\ell_1 R \ell_2$*;*

(B3) for all $1 \leq i \leq n$*,* $1 \leq k \leq 2$ *holds: if* $|M_k^i| > 1$ *then* $\forall \ell \in M_k^i$ *holds* $M_k^i = \sum_{(\ell' \in dom(K(\ell))) \wedge (\ell' \in dom(M_k))} \bot_{\mathfrak{M}}[\ell' \mapsto 1]$*.*

We call the coarsest relations R *and* R' *fulfilling the conditions (A1)-(A4) and (B1)-(B3) in Definition 4.16 for a fixed* $\mathrm{IMC}^{\mathrm{G}}$ *program* F label chain-bisimilarity *and* multiset chain-bisimilarity *accordingly. We denote* $\ell_1 \sim^{ch}_{\mathfrak{Lab}} \ell_2$ *(label chain-bisimilar) and* $M_1 \sim^{ch}_{\mathfrak{M}} M_2$ *(multiset chain-bisimilar) if* (ℓ_1, ℓ_2) *is contained in some label chain-bisimulation and* (M_1, M_2) *is contained in a corresponding to it multiset chain-bisimulation.*

Computing the coarsest label chain-bisimulation, given an $\mathrm{IMC}^{\mathrm{G}}$ program F for which the Pathway Analysis has been conducted, can be done by an algorithm similar to the algorithm for computing the coarsest bisimulation on states reachable from $\mathcal{E}_\Gamma[\![F]\!]$ that has been described in Section 4.2.4. The first difference is that we have labels instead of states; the second difference is that instead of transitions between states we have a generate-relation between labels, i.e. we refine the partition classes of labels according to whether for each ℓ_1 and ℓ_1' from the same partition class there exist ℓ_2 and ℓ_2' in another partition class such that both $\ell_1 \longrightarrow \ell_2$ and $\ell_1' \longrightarrow \ell_2'$ hold.

We can thus compute the initial partition of labels according to the conditions (A1)-(A3) and then do a refinement of the partition classes according to the condition (A4). Checking the splitting condition (i.e. checking the conditions (B1)-(B3) for the multisets generated by the labels) is not that straightforward as just checking the existence of transitions in the splitting algorithm in Table 4.5 but can be done linear in the size of generated multisets. The time complexity of computing the coarsest label chain-bisimulation will be then $O(m * log(n))$ and the space complexity will be $O(m)$, where $n = |Labs(F)|$ and m is the number of generate-relations, i.e. $m = |\{\ell_1 \in Labs(F), \ell_2 \in Labs(F) | \ell_1 \longrightarrow \ell_2\}|$. Deciding whether two multisets are chain-bisimilar can be then done in the linear time in the number of labels in the domains of two multisets.

Lemma 4.17 (Chain-bisimilarities)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9, the label and multiset chain-bisimilarities for* F *are equal accordingly to the unions of all label and multiset chain-bisimulation relations for* F*.*

Proof. This is a standard result that bisimilarities are unions of all bisimulations, see for example [BK08]. It can be proved by showing that on the one hand the union of two or more label/multiset chain-bisimulation relations is accordingly a label/multiset chain-bisimulation itself. This can be done by going through Definition 4.16 and showing that all the conditions are fulfilled for unions of relations. On the other hand, the coarsest chain-bisimulations are contained in the union of chain-bisimulations for the same F. □

Intuitively, we allow two equivalent multisets to contain each any number of "inactive" labels: as the latter do not participate in any chain, they can never be executed and therefore do not contribute to the behaviour of the state of the transition system which is described by the multiset. The rest of the labels need to be equivalent, i.e. according to the condition (A2) have the same corresponding action name (τ if hidden), which means that all transitions in which they participate will be decorated with the same action name. Moreover, their execution generates equivalent multisets. It is possible to have a correspondence between sets of labels instead of separate labels but in this case the whole set should be killed by the execution of each constituent label (condition (B3)). When the execution of one of the labels kills the whole set then the multisets will stay equivalent after the execution of equivalent labels.

Additionally the condition (A3) has been introduced: only those labels are considered equivalent that either participate or not participate together in each

chain from $\mathfrak{T}$ or are completely interchangeable, i.e. every chain with one of the labels exchanged for the other one is again in $\mathfrak{T}$. This is enough to achieve a bisimulation relation on multisets, as we can differentiate now between labels that have some "chain relation" between each other and the chains that don't. Without (A3), for example, the following multisets would be matched by the chain-bisimilarity: $E_1 = \mathsf{a}^{\ell_1}.\mathbf{0} \parallel_{\mathsf{a}} \mathsf{a}^{\ell_2}.\mathbf{0}$ and $E_2 = \mathsf{a}^{\ell_3}.\mathbf{0} \parallel_{\emptyset} \mathsf{a}^{\ell_4}.\mathbf{0}$ (both expressions may be derivatives of some IMC$^{\mathrm{G}}$ programs). Without (A3) we would establish $E_1 \sim^{ch}_{\mathfrak{M}} E_2$ because $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_3$ and $\ell_2 \sim^{ch}_{\mathbf{Lab}} \ell_4$. In the reality, however, $\mathsf{a}^{\ell_1}.\mathbf{0} \parallel_{\mathsf{a}} \mathsf{a}^{\ell_2}.\mathbf{0} \xrightarrow[C]{\mathsf{a}} \mathbf{0} \parallel_{\mathsf{a}} \mathbf{0}$ but $\mathsf{a}^{\ell_1}.\mathbf{0} \parallel_{\emptyset} \mathsf{a}^{\ell_2}.\mathbf{0} \xrightarrow[C']{\mathsf{a}} \mathbf{0} \parallel_{\emptyset} \mathsf{a}^{\ell_2}.\mathbf{0}$ for appropriate chains C and C', and two resulting multisets are not equivalent anymore. Obviously in order to improve this situation we need to take into account the information about the chains in which equivalent labels participate.

Note that in case $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ these two labels can either be present in two parallel processes (condition (A3a)) or they can be connected by the choice operator (condition (A3b)): the confusion is not possible because these two situations exclude each other and also because both conditions (A3a) and (A3b) are transitive. This means that if $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ and some third label ℓ_3 always participates or not participates in any chain together with ℓ_1, then it does the same with ℓ_2. If ℓ_3 is completely interchangeable with ℓ_1, then it is also completely interchangeable with ℓ_2. The situation that, for example, $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ due to the condition (A3a) and $\ell_3 \sim^{ch}_{\mathbf{Lab}} \ell_1$ due to the condition (A3b) is impossible: if ℓ_3 is interchangeable with ℓ_1 then there exists a chain where ℓ_2 is present but ℓ_1 is absent (if $\ell_1 \neq \ell_3$). We will use these facts in the proof of the transitivity of $\sim^{ch}_{\mathfrak{M}}$ below.

Note that the relation on multisets $\sim^{ch}_{\mathfrak{M}}$ is not the coarsest bisimulation on $\mathfrak{M}$. For example, different chains can have the same behaviour: in the process $\mathsf{b}^{\ell_5}.\mathsf{a}^{\ell_1}.\mathbf{0} \parallel_{\mathsf{a}} \mathsf{a}^{\ell_2}.\mathbf{0} + \mathsf{b}^{\ell_6}.\mathsf{a}^{\ell_3}.\mathbf{0} \parallel_{\mathsf{a}} \mathsf{a}^{\ell_4}.\mathbf{0}$ there are two chains, $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ and $\bot_{\mathfrak{M}}[\ell_3 \mapsto 1, \ell_4 \mapsto 1]$, with the same behaviour, but they will not be identified by our relation $\sim^{ch}_{\mathfrak{M}}$ because (A3) is not true for any of the pairs (ℓ_1, ℓ_3), (ℓ_1, ℓ_4), (ℓ_2, ℓ_3), (ℓ_2, ℓ_4). Another source of non-optimality of the relation $\sim^{ch}_{\mathfrak{M}}$ is that the conditions (A3a) and (A3b) in Definition 4.16 can in some sense "interfere" with each other: in the process $\mathsf{b}^{\ell_4}.\mathsf{a}^{\ell_1}.\mathbf{0} \parallel_{\{\mathsf{a}\}} (\mathsf{b}^{\ell_5}.\mathsf{a}^{\ell_2}.\mathbf{0} + \mathsf{b}^{\ell_6}.\mathsf{a}^{\ell_3}.\mathbf{0})$ two states corresponding to the multisets $\bot_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_5 \mapsto 1, \ell_6 \mapsto 1]$ on the one hand and $\bot_{\mathfrak{M}}[\ell_4 \mapsto 1, \ell_2 \mapsto 1]$ or $\bot_{\mathfrak{M}}[\ell_4 \mapsto 1, \ell_3 \mapsto 1]$ on the other hand are obviously bisimilar: both can execute the action b and then the action a. They cannot however be identified as such by the relation $\sim^{ch}_{\mathfrak{M}}$ because neither $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ nor $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_3$ holds ($\ell_2 \sim^{ch}_{\mathbf{Lab}} \ell_3$ holds).

The two described sources of non-optimality will be addressed in the definition of the equivalence relation $\sim^{syn}_{\mathfrak{M}}$ in Section 4.4 which will be proved to be an extension of the relation $\sim^{ch}_{\mathfrak{M}}$. Unfortunately these are not the only sources of

non-optimality: chains with equivalent behaviour may have different number of constituent labels, some labels may never be executed even if they participate in some chains because they are blocked by the environment or are generated only by actions blocked by the environment etc. The identification of such cases requires a more complex analysis. The equivalence relation $\sim^{ch}_{\mathfrak{M}}$ has nevertheless its advantages the main of which is the simplicity of its computation: it is computed using only the results of the chain, generate and kill operators. The computation, for example, of the relation $\sim^{syn}_{\mathfrak{M}}$ will require some additional operators on the analysed IMC$^{\mathrm{G}}$ program F.

4.3.3 Chain-bisimulations and "ordinary" bisimulations

Before considering the question of how chain-bisimulations are related to "ordinary" bisimulations let us discuss some of the properties of chain-bisimulations. Chain-bisimulations (including chain-bisimilarities) are not necessary preserved under subtraction: given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9, then from $M_1 \sim^{ch}_{\mathfrak{M}} M_1'$ and $M_2 \sim^{ch}_{\mathfrak{M}} M_2'$ does not necessarily follow $M_1 - M_1' \sim^{ch}_{\mathfrak{M}} M_2 - M_2'$, even if all involved multisets represent exposed labels of IMC$^{\mathrm{G}}$ expressions derivable from F. Take as an example the IMC$^{\mathrm{G}}$ program $\underline{Y := \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{a}^{\ell_2}.Y}}$. Then for $dom(M_1) = \{\ell_1\}$, $dom(M_2) = \{\ell_2\}$ and $dom(M_3) = \{\ell_1, \ell_2\}$ all three multisets are chain-bisimilar. However $M_3 - M_1$ is obviously neither chain-bisimilar nor bisimilar in the sense of Definition 4.11 to $M_3 - M_3$. Chain-bisimulations are not additive in general case as well due to the condition (B3) in Definition 4.16 which might be violated.

On the other hand, multiset chain-bisimilarities have some useful properties, in particular, they are equivalence relations on multisets of labels. This is useful because (in case they are also "ordinary" bisimulations – which we will prove in Lemma 4.20) we could group equivalent states together and construct in this way a smaller labelled transition system for the semantics of an IMC$^{\mathrm{G}}$ program without rates for which the Pathway Analysis has been conducted and the multiset chain-bisimilarity has been computed.

Lemma 4.18 (Chain-bisimilarities are equivalences)**.** *Given an* IMC$^{\mathrm{G}}$ *program F and $\mathcal{F}$ as in Definition 4.9, then the relation $\sim^{ch}_{\boldsymbol{Lab}}$ on active labels and the relation $\sim^{ch}_{\mathfrak{M}}$ on multisets of labels are equivalence relations. This means that, assuming $\boldsymbol{Lab}' \subseteq \boldsymbol{Lab}$ is a set of labels such that $\forall \ell \in \boldsymbol{Lab}'\ \exists C \in \mathfrak{T}$ such that $\ell \in dom(C)$, then for all labels $\{\ell_1, \ell_2, \ell_3\} \subseteq \boldsymbol{Lab}'$, for all mutisets $\{M_1, M_2, M_3\} \subseteq \mathfrak{M}$ holds:*

1. $\ell_1 \sim^{ch}_{\boldsymbol{Lab}} \ell_1$ *and* $M_1 \sim^{ch}_{\mathfrak{M}} M_1$

2. $\ell_1 \sim^{ch}_{\boldsymbol{Lab}} \ell_2 \Rightarrow \ell_2 \sim^{ch}_{\boldsymbol{Lab}} \ell_1$ *and* $M_1 \sim^{ch}_{\mathfrak{M}} M_2 \Rightarrow M_2 \sim^{ch}_{\mathfrak{M}} M_1$
3. $(\ell_1 \sim^{ch}_{\boldsymbol{Lab}} \ell_2) \wedge (\ell_2 \sim^{ch}_{\boldsymbol{Lab}} \ell_3) \Rightarrow \ell_1 \sim^{ch}_{\boldsymbol{Lab}} \ell_3$ *and* $(M_1 \sim^{ch}_{\mathfrak{M}} M_2) \wedge (M_2 \sim^{ch}_{\mathfrak{M}} M_3) \Rightarrow M_1 \sim^{ch}_{\mathfrak{M}} M_3$

Proof. The definitions of the relations $\sim^{ch}_{\mathbf{Lab}}$ and $\sim^{ch}_{\mathfrak{M}}$ in Definition 4.16 imply that these are the *coarsest* relations for which the conditions (A1)-(A4) and the conditions (B1)-(B3) hold. Therefore in order to prove the statements 1-3 from the formulation of the lemma it is enough to show that the identity relations on active labels and on multisets of labels satisfy accordingly the conditions (A1)-(A4) and (B1)-(B3); that for any two relations satisfying accordingly the conditions (A1)-(A4) and (B1)-(B3) holds that their inverse relations also satisfy accordingly the conditions (A1)-(A4) and (B1)-(B3) (then they both are contained in the coarsest relations according to Lemma 4.17); that two pair of relations satisfying accordingly the conditions (A1)-(A4) and (B1)-(B3) can be combined together and still satisfy accordingly the conditions (A1)-(A4) and (B1)-(B3) (then two pairs of relations as well as their combinations are contained in the coarsest relations according to Lemma 4.17).

A pair of the identity relations on accordingly active labels (i.e. on $\mathbf{Lab}'$) and multisets satisfy all the conditions (A1)-(A3): note that both (A3a) and (A3b) are satisfied in case $\ell_1 = \ell_2$. In the conditions for multisets we can ascertain that $M_1^i = M_2^i$ for $1 \leq i \leq n$ – then the consitions (A4) and (B1)-(B3) are satisfied as well.

Assume that a pair of relations R and R' satisfy the conditions (A1)-(A4) and (B1)-(B3). Then also a pair of relations $R^{-1} = \{(\ell_1, \ell_2) | (\ell_2, \ell_1) \in R\}$ and $R'^{-1} = \{(M_1, M_2) | (M_2, M_1) \in R'\}$ satisfy these conditions as all of the conditions (A1)-(A4) and (B1)-(B3) are in fact already symmetric by definition.

Assume that two pairs of relations R_1 and R'_1 on the one hand and R_2 and R'_2 on the other hand satisfy the conditions (A1)-(A4) and (B1)-(B3). Then we will show that also a pair of relations $R_1 \circ R_2 = \{(\ell_1, \ell_3) | (\exists(\ell_1, \ell_2) \in R) \wedge (\exists(\ell_2, \ell_3) \in R)\}$ and $R'_1 \circ R'_2 = \{(M_1, M_3) | (\exists(M_1, M_2) \in R'_1) \wedge (\exists(M_2, M_3) \in R'_2)\}$ satisfy these conditions.

Most of the conditions clearly hold besides the condition (A3). We have already mentioned in the discussion above that in case $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2 \sim^{ch}_{\mathbf{Lab}} \ell_3$ and both for the first and for the second relation holds the condition (A3a) then the condition (A3a) holds for ℓ_1 and ℓ_3. The same is true if it is the condition (A3b) instead of the condition (A3a) that holds for both $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ and $\ell_2 \sim^{ch}_{\mathbf{Lab}} \ell_3$. Two remaining configurations, i.e. (A3a) holds for $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$ and (A3b) holds for

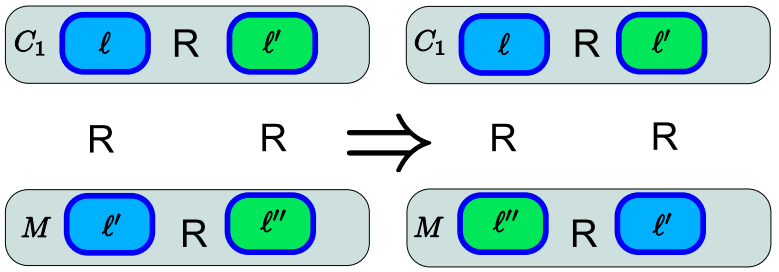

Figure 4.3 – Graphical illustration of the proof of Lemma 4.19: labels in two chain-bisimilar multisets can be "rearranged" in such a way that the same labels are always grouped together.

$\ell_2 \sim^{ch}_{\mathbf{Lab}} \ell_3$ and the other way round, are impossible in case all three labels are different. In case some of the labels are the same, for example, $\ell_1 = \ell_2$, then both (A3a) and (A3b) hold for this pair and we are back to the previously considered case. □

We show an important result that, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9, then for two multisets which are in a chain-bisimulation relation which is also an equivalence holds: one of them is a chain from $\mathfrak{T}$ if and only if the other one is also a chain from $\mathfrak{T}$. It is a necessary step in proving that any chain-bisimulation that is an equivalence is an "ordinary" bisimulation – this would obviously not be the case if out of two chain-bisimilar multisets one would be a chain and the other not.

Lemma 4.19 (Chains and chain-bisimulations)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F *without rates,* $\mathcal{F}$ *as in Definition 4.9 and a pair of equivalence relations* (R, R') *that are accordingly a label and a multiset chain-bisimulations,* $M_1 \in \mathfrak{M}$, $M_2 \in \mathfrak{M}$, *such that* $M_1 R' M_2$, *then* $\exists C_1 \in \mathfrak{T}$ *such that* $C_1 \leq M_1$ *iff* $\exists C_2 \in \mathfrak{T}$ *such that* $C_2 \leq M_2$ *and* $C_1 R' C_2$.

Proof. Note that due to the fact that R' is an equivalence it is enough to prove the statement only one way: the other way will follow because from $M_1 R' M_2$ follows $M_2 R' M_1$ for any $M_1 \in \mathfrak{M}$, $M_2 \in \mathfrak{M}$.

For any $C_1 \leq M_1$ such that $C_1 \in \mathfrak{T}$ and $C_1 = \sum_{i=1..n} \perp_{\mathfrak{M}} [\ell_i \mapsto 1]$ there exists $M \leq M_2$ such that $M = \sum_{i=1..n} \perp_{\mathfrak{M}} [\ell'_i \mapsto 1]$ and for all $1 \leq i \leq n$ holds $\ell_i R \ell'_i$. If for all ℓ'_i holds that ℓ'_i is interchangeable with ℓ_i (condition (A3b) in Definition

4.16) then $M \in \mathfrak{T}$ clearly holds. Otherwise we can "rearrange" the relations of labels in M to the labels in C_1 in such a way that the condition (A3b) will always be true.

Assume that there exists some $\bot_{\mathfrak{M}}[\ell' \mapsto 1] \leq M$ such that $\ell R \ell'$ for some $\bot_{\mathfrak{M}}[\ell \mapsto 1] \leq C_1$ and the condition A3(a) in Definition 4.16 holds. This means that it also holds $\ell' \in C_1$ (remember that the condition A3(a) requires that ℓ and ℓ' participate in all chains "together"). Additionally there will exist $\bot_{\mathfrak{M}}[\ell'' \mapsto 1] \leq M$ such that $\ell' R \ell''$. We can do a "rearrangement" of labels such that $\ell' R \ell'$ and $\ell R \ell''$ (remember that the relation R is symmetric and transitive). See the schematic picture of the rearrangement of labels in Figure 4.6. We can continue the rearrangement until all the labels in C_1 are interchangeable with their corresponding labels in M. □

We are now ready to prove the main result of this section – namely, that chain-bisimulations that are equivalences on the states from $\mathfrak{M}$ are also "ordinary" bisimulations in the sense of Definition 4.11, i.e. that our definitions of chain-bisimulations are correct with respect to the semantics of an IMC$^{\mathrm{G}}$ program to be analysed.

Theorem 4.20 (Chain-bisimulations are bisimulations)**.** *Given an* IMC$^{\mathrm{G}}$ *program F without rates, $\mathcal{F}$ as in Definition 4.9 and a pair of equivalence relations (R, R') that are accordingly a label and a multiset chain-bisimulation, then R' is a bisimulation relation on $\mathfrak{M}$. In particular, from $M_1 \sim^{ch}_{\mathfrak{M}} M_2$ follows $M_1 \sim M_2$ for any $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$.*

Proof. We do not have rates in the syntactic description of F, therefore it is enough to show that only condition 1 in Definition 4.11 of the bisimulations on $\mathfrak{M}$ holds. Also note that if we prove the first statement of the lemma this will also prove that from $M_1 \sim^{ch}_{\mathfrak{M}} M_2$ follows $M_1 \sim M_2$: the synchronisation-bisimilarity is also a synchronisation-bisimulation, it is an equivalence according to Lemma 4.18 and it would be contained in the relation $\sim$ because the bisimilarity on $\mathfrak{M}$ is a union of all bisimulations on $\mathfrak{M}$.

We only need to show therefore that for all $C_1 \in \mathfrak{T}$ and $M_1' \in \mathfrak{M}$ such that $C_1 \leq M_1$ and $M_1' = M_1 - K_{ch}(C_1) + G_{ch}(C_1)$ there exists $C_2 \in \mathfrak{T}$ with $C_2 \leq M_2$, $Name^h_{ch}(C_1) = Name^h_{ch}(C_2)$, $M_2' = M_2 - K_{ch}(C_2) + G_{ch}(C_2)$ and $(M_1', M_2') \in R'$. From Lemma 4.19 follows that there in fact exists a chain $C_2 \in \mathfrak{T}$ such that $C_2 \leq M_2$ and $C_1 R' C_2$. Accordingly $Name^h_{ch}(C_1) = Name^h_{ch}(C_2)$ holds due to the condition (A2) in Definition 4.16. Note that if both chains contain only

one label in their domains then it is possible that they correspond to originally different but hidden action names. If two chains contain more than one label in their domains then this is not possible.

We can observe that $(M_1 - K_{ch}(C_1))R'(M_2 - K_{ch}(C_2))$. This is due to the condition (B3) in Definition 4.16: the execution of one of the labels in a containing more than one label in its domain M_k^i for $1 \leq k \leq 2$ and $1 \leq i \leq n$ kills all the labels from M_k^i and no other labels. We can add accordingly $G_{ch}(C_1)$ and $G_{ch}(C_2)$ because in this particular case we have the additivity property. It is namely the case that all newly generated labels can neither be killed or kill already exposed labels – this can be shown by induction on the syntactic structure of F. Altogether this proves $(M_1 - K_{ch}(C_1) + G_{ch}(C_1))R'(M_2 - K_{ch}(C_2) + G_{ch}(C_2))$ which is enough to show that the statement of the lemma holds. □

We have thus proved that $\sim_{\mathfrak{M}}^{ch}$ is a bisimulation on multisets of exposed labels of the derivative expressions of the IMC$^{\mathrm{G}}$ program that has been analysed. It is easy to see that in general $\sim_{\mathfrak{M}}^{ch} \neq \sim$ and we have already presented examples of that. In the next section we will devise another bisimulation relation on multisets $\sim_{\mathfrak{M}}^{syn}$ such that $\sim_{\mathfrak{M}}^{ch} \subseteq \sim_{\mathfrak{M}}^{syn}$ and $\sim_{\mathfrak{M}}^{ch} \neq \sim_{\mathfrak{M}}^{syn}$ in general. The enlargement of the equivalence relation will be achieved by relaxing the requirements to the equivalence on labels and by additionally taking into account the relations of labels *inside* two equivalent multisets and not only *between* them as in the relation $\sim_{\mathfrak{M}}^{ch}$.

We conjecture that chain-bisimilarities are equal to the "ordinary" bisimilarity on linear IMC$^{\mathrm{G}}$ programs, i.e. with no parallel operator (because chains do not play any role in the linear fragment of IMC$^{\mathrm{G}}$), and on a symmetrical composition in the sense of [Her02], i.e. on a parallel execution of several identical (but uniquely labelled) linear IMC$^{\mathrm{G}}$ processes. We also surmise that it might "recognise" situations when linear components in a symmetrical composition are not syntactically equal but still have the same semantics. These suppositions need the further research to be conducted in order to verify them.

4.4 Synchronisation-bisimulations

In this section we will enlarge bisimulation relations constructed above in Section 4.3. Remember that in Section 4.3.2 in case we would like to determine whether two multisets are chain-bisimilar we only had to determine whether they

consist of pairwise chain-bisimilar active labels. Analogously to label and multiset chain-bisimulations we will define below label and multiset synchronisation-bisimulations. The coarsest label and multiset synchronisation-bisimulations will be denoted accordingly $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$. Similarly to chain-bisimilar labels, synchronisation-bisimilar labels will have the same associated action name (or both will be associated either with an internalised action name or with a τ-action) and will generate synchronisation-bisimilar multisets.

On the other hand, synchronisation-bisimulations on multisets will use information about two different relations on labels. The relations of the first type, i.e. label synchronisation-bisimulations, are similar to label chain-bisimulations: they relate labels with similar behaviour. In fact it holds that $\sim^{ch}_{\mathbf{Lab}} \subseteq \sim^{syn}_{\mathbf{Lab}}$. Relations of the second kind map pairs of labels to sets of external action names. The latter contain those action names on which two IMC$^{\mathrm{G}}$ subprocesses, each containing a label from the labels' pair, are synchronising. The reason for introducing this second type of relations is in a sense that label synchronisation-bisimulations are "too permissive": labels equivalent according to them do not have the same pattern of participation in the same chains with other labels. We correct this circumstance with the use of the second relations.

It is required therefore that the relations of the second kind between "active" (i.e. participating in at least one chain) labels inside two multisets follow the same pattern. We will say that synchronisation-bisimilar multisets have the same (potential) "synchronisation structure". A synchronisation structure of a multiset will retain the information about all the chains which contain the labels that can be generated after any number of semantic steps by labels in a multiset. This information will be computed by an operator **Syn** beforehand on an IMC$^{\mathrm{G}}$ program (using a number of auxiliary operators) and saved in the mapping $\mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$. The resulting mapping returns for all pairs of labels an over-approximation of action names on which the labels generated by them could be synchronised.

Differences in synchronisation structures will help us to determine, for example, that the processes $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.X} \,||\mathsf{b}||\, \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{b}^{\ell_4}.Y}$ and $\underline{X := \mathsf{a}^{\ell_5}.\mathsf{b}^{\ell_6}.X} \,||\, \emptyset||\, \underline{Y := \mathsf{a}^{\ell_7}.\mathsf{b}^{\ell_8}.Y}$ cannot be considered synchronisation-bisimilar: there is a potential synchronisation on b in the first case and there is no such synchronisation in the second case. The processes are indeed not bisimilar in the sense of strong bisimulation as the second process can execute a sequence of actions a, a, b, b which the first cannot. We can even handle a "spurious" synchronisation in some cases: both $\underline{X := \mathsf{a}^{\ell_1}.X} \,||\mathsf{b}||\, \underline{Y := \mathsf{a}^{\ell_2}.Y + \mathsf{b}^{\ell_3}.Y}$ and $\underline{X := \mathsf{a}^{\ell_4}.X} \,||\mathsf{b}||\, \underline{Y := \mathsf{a}^{\ell_5}.Y}$ will not be considered as synchronising on b according to our operator **Syn**.

Additionally we will require that for any multiset which is a chain all multisets that are synchronisation-bisimilar to it are chains as well. This is nec-

essary in order to determine, for instance, that the processes $\underline{X := \mathsf{b}^{\ell_1}.X}$ and $\underline{X := \mathsf{b}^{\ell_2}.X} \,||\{\mathsf{a},\mathsf{b}\}||\, \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{b}^{\ell_4}.Y}$ are not bisimilar: the first process will execute the action b infinite number of times and the second process is stuck. Note that we cannot determine this fact neither from the analysis of individual labels (a^{ℓ_3} is not active and is therefore not taken into account) nor from the analysis of the results returned by the operator **Syn**.

The final requirement concerns the possibility to identify n-to-m correspondences of labels. The idea is that we can identify groups of labels that are synchronisation-bisimilar and are excluding the execution of each other (by "killing" each other) – this means that they are choice alternatives. A group of n such labels will be then synchronisation-bisimilar to a group of m such labels, which is also intuitively explainable: it does not matter how many alternatives we have in case we can only choose one of them. It is also easy to identify such situations – we only need to group synchronisation-bisimilar labels that are mutually "killing" each other.

Consequently we will prove that synchronisation-bisimulation relations are bisimulations on $\mathfrak{M}$. We will also show that synchronisation-bisimilarities contain chain-bisimilarities (for the same IMC$^{\mathrm{G}}$ program F) and are in general strictly coarser than them. We can as a result determine many interesting bisimulation cases: for example, match the processes $\underline{X := \mathsf{a}^{\ell_1}.X}$ and $\underline{Y := \mathsf{a}^{\ell_2}.Y + \mathsf{a}^{\ell_3}.Y}$ (different number of choice alternatives with the same behaviour), $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.X}$ and $\underline{Y := \mathsf{a}^{\ell_3}.\mathsf{b}^{\ell_4}.\mathsf{a}^{\ell_5}.\mathsf{b}^{\ell_6}.Y}$ (behaviour repetition), $\mathsf{hide\ a\ in}\ (\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.X} \,||\, \mathsf{b}||\, \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{b}^{\ell_4}.Y})$ and $\mathsf{hide\ c\ in}\ (\underline{X := \mathsf{c}^{\ell_5}.\mathsf{b}^{\ell_6}.X} \,||\, \mathsf{b}||\, \underline{Y := \mathsf{c}^{\ell_7}.\mathsf{b}^{\ell_8}.Y})$ (different action names before internalisation, the same synchronisation structure in two processes).

Failing to recognise all bisimilar IMC$^{\mathrm{G}}$ processes comes in particular from the circumstances that they may have different synchronisation structures due to the labels that are actually unreachable or that we only consider one-to-one correspondences between labels and not one-to-two etc. correspondences. To illustrate the first circumstance consider the following example: $\mathsf{c}^{\ell_1}.(\underline{X := \mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_3}.X} \,||\, \{\mathsf{a},\mathsf{b}\}||\, \underline{Y := \mathsf{b}^{\ell_4}.\mathsf{a}^{\ell_5}.Y})$ and $\mathsf{c}^{\ell_6}.\mathbf{0}$. In the first process all the labels besides ℓ_1 cannot be executed because of the different action order but the processes will not be identified as synchronisation-bisimilar. The second fact can be illustrated by the processes $\underline{X := \mathsf{a}^{\ell_1}.X}$ and $\underline{Y := \mathsf{a}^{\ell_2}.Y} \,||\mathsf{a}||\, \underline{Z := \mathsf{a}^{\ell_3}.Z}$, where in the latter case two synchronising processes are proceeding "in lock". We cannot however identify them due to different number of their exposed "active" labels.

$$
\begin{array}{rcll}
\mathbf{fn}_\Phi(\mathsf{a}^\ell.X) & = & \{\mathsf{a}\} \cup \Phi(X) & (1)\\
\mathbf{fn}_\Phi(\lambda^\ell.X) & = & \Phi(X) & (2)\\
\mathbf{fn}_\Phi(\mathsf{a}^\ell.P) & = & \{\mathsf{a}\} \cup \mathbf{fn}_\Phi(P) & (3)\\
\mathbf{fn}_\Phi(\lambda^\ell.P) & = & \mathbf{fn}_\Phi(P) & (4)\\
\mathbf{fn}_\Phi(P_1 + P_2) & = & \mathbf{fn}_\Phi(P_1) \cup \mathbf{fn}_\Phi(P_2) & (5)\\
\mathbf{fn}_\Phi(\mathsf{hide}\ A\ \mathsf{in}\ P) & = & \mathbf{fn}_\Phi(P) \backslash A & (6)\\
\mathbf{fn}_\Phi(P_1 \shortmid\!\shortmid A \shortmid\!\shortmid P_2) & = & \mathbf{fn}_\Phi(P_1) \cup \mathbf{fn}_\Phi(P_2) & (7)\\
\mathbf{fn}_\Phi(\underline{X := P}) & = & \mathbf{fn}_\Phi(P) & (8)\\
\mathbf{fn}_\Phi(\mathbf{0}) & = & \emptyset & (9)
\end{array}
$$

Table 4.6 – Modified operator $\mathbf{fn} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Act}}$ with an environment $\Phi \in \mathbf{Var} \to 2^{\mathbf{Act}}$.

4.4.1 Operator Syn

In this section we will define an operator **Syn** that we have already mentioned before. The operator takes as an input a syntactic description of an $\mathrm{IMC}^{\mathrm{G}}$ program and returns a mapping where for each pair of labels there is a set of external actions. We will also prove that the way **Syn** is defined corresponds to our expectations: i.e. if it returns on a pair of labels (ℓ_1, ℓ_2) a set of external actions A, then for each $\mathsf{a} \in A$ holds that both labels from the pair can generate after a number of steps two labels, e.g. ℓ_1' and ℓ_2', such that both ℓ_1' and ℓ_2' correspond to the action name a and participate in at least one chain together.

Before defining a **Syn** operator itself, we will need to additionally define several instrumental operators. In particular, we will define an operator **fnvar** which returns free external action names in process definitions and a modified operator **fn** which takes the information gathered by the operator **fnvar** into account while assessing which free action names occur in an $\mathrm{IMC}^{\mathrm{G}}$ expression. This means that if some variable X occurs in an $\mathrm{IMC}^{\mathrm{G}}$ expression E and an action a is free in the definition of X, then the modified operator **fn** will return on E a set of free action names A with $\mathsf{a} \in A$ even if a strictly speaking does not occur in E. For example, for $Y := \mathsf{a}^{\ell_1}.\underline{X := \mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.Y}$ the operator **fnvar** will return $Y \mapsto \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and therefore the modified operator **fn** with an environment will return $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ as free names in $\underline{X := \mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.Y}$ if it will use the result returned by **fnvar** as an environment. See the definitions of the operators **fnvar** and **fn** in Tables 4.6 and 4.7.

We define $\bot_{\mathbf{fnvar}}$ as the least element of the mapping $\mathbf{Var} \to 2^{\mathbf{Act}}$. The mapping can be regarded as a complete lattice with a partial order such that smaller elements in $\mathbf{Var} \to 2^{\mathbf{Act}}$ have smaller sets of action names to which process vari-

$$
\begin{array}{rcll}
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\mathsf{a}^{\ell}.X]\!] & = & \Phi & (1)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\lambda^{\ell}.X]\!] & = & \Phi & (2)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\mathsf{a}^{\ell}.P]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}}[\![P]\!] & (3)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\lambda^{\ell}.P]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}}[\![P]\!] & (4)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![P_1 + P_2]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}}[\![P_1]\!] \cup \mathbf{fnvar}_{\mathbf{\Phi}}[\![P_2]\!] & (5)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}}[\![P]\!] & (6)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![P_1 \ {}_{\shortparallel A \shortparallel}\ P_2]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}}[\![P_1]\!] \cup \mathbf{fnvar}_{\mathbf{\Phi}}[\![P_2]\!] & (7)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\underline{X := P}]\!] & = & \mathbf{fnvar}_{\mathbf{\Phi}[X \mapsto \mathbf{fn}_{\mathbf{\Phi}}(\mathbf{P})]}[\![P]\!] & (8)\\
\mathbf{fnvar}_{\mathbf{\Phi}}[\![\mathbf{0}]\!] & = & \Phi & (9)
\end{array}
$$

Table 4.7 – Free names of process definitions. The operator $\mathbf{fnvar} : \mathrm{IMC}^{\mathrm{G}} \to \mathbf{Var} \to 2^{\mathbf{Act}}$ with an environment $\Phi \in \mathbf{Var} \to 2^{\mathbf{Act}}$.

ables are mapped. Then clearly in $\bot_{\mathbf{fnvar}}$ each variable is mapped to the empty set. We will usually calculate $\mathbf{fn}_{\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]}(E)$ for some $\mathrm{IMC}^{\mathrm{G}}$ program F and its subexpression $E \preceq F$. Remember that from the well-formedness condition for F follows that any action names that are not hidden in F will remain not hidden in any of its derivatives.

As a next step we define in Table 4.8 the operator $\mathbf{psyn} : \mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$ which computes for all pairs of labels the external actions on which their generated labels can be potentially synchronising. This is a rather coarse approximation: "potentially" refers to the fact that we take into account only parallelisation constructs and which action names are free in subexpressions (see rule (7) in Table 4.8) but not the generated labels itself. Thus **psyn** will return, for example, $\{\mathsf{a}\}$ for the pair (ℓ_1, ℓ_2) and the $\mathrm{IMC}^{\mathrm{G}}$ process $\underline{X := \mathsf{b}^{\ell_1}.X} \ {}_{\shortparallel}\mathsf{a}_{\shortparallel}\ \underline{Y := \mathsf{c}^{\ell_2}.Y + \mathsf{a}^{\ell_3}.Y}$. We will consequently filter out some of the external actions with the operator **Syn** – in particular, **Syn** will return the empty set on the pair (ℓ_1, ℓ_2) in the above example.

We consider the codomain of **psyn** to be a complete lattice with a partial order such that smaller elements in $\mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$ have smaller sets of action names to which pairs of labels are mapped. Then clearly $\bot_{\mathbf{syn}}$ (defined as $\bot_{\mathbf{syn}}((\ell_1, \ell_2)) = \emptyset$ for all $\ell_1 \in \mathbf{Lab}$ and $\ell_2 \in \mathbf{Lab}$) is the least element of $\mathrm{IMC}^{\mathrm{G}} \to \mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$.

We are now ready to define the operator **Syn** in Table 4.9. If the input of **Syn** is an $\mathrm{IMC}^{\mathrm{G}}$ program F then the environments R and Φ will usually be equal to accordingly $\mathbf{psyn}[\![F]\!]$ and $\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]$. In Lemma 4.22 we will prove that the operator **Syn** does return the result with the expected properties. The statement of the lemma is graphically demonstrated in Figure 4.4. The

$$
\begin{array}{rcll}
\mathbf{psyn}[\![\mathsf{a}^{\ell}.X]\!] & = & \bot_{\mathbf{syn}} & (1)\\
\mathbf{psyn}[\![\lambda^{\ell}.X]\!] & = & \bot_{\mathbf{syn}} & (2)\\
\mathbf{psyn}[\![\mathsf{a}^{\ell}.P]\!] & = & \mathbf{psyn}[\![P]\!] & (3)\\
\mathbf{psyn}[\![\lambda^{\ell}.P]\!] & = & \mathbf{psyn}[\![P]\!] & (4)\\
\mathbf{psyn}[\![P_1 + P_2]\!] & = & \mathbf{psyn}[\![P_1]\!] \sqcup \mathbf{psyn}[\![P_2]\!] & (5)\\
\mathbf{psyn}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathbf{psyn}[\![P]\!] & (6)\\
\mathbf{psyn}[\![P_1 \between A \between P_2]\!] & = & \mathbf{psyn}[\![P_1]\!] \sqcup \mathbf{psyn}[\![P_2]\!] \sqcup R \text{ where} & (7)\\
B & := & \bigsqcup_{\mathsf{a}\in A\ st\ \exists C\in\mathfrak{T}_{\Lambda}[\![P_1\between A\between P_2]\!]\ st\ Names(C)=\{\mathsf{a}\}} \{\mathsf{a}\} & \\
R & := & \bigsqcup_{\ell_1\in Labs(P_1)} \bigsqcup_{\ell_2\in Labs(P_2)} \bot_{\mathbf{syn}}[(\ell_1,\ell_2)\mapsto B] & \\
\mathbf{psyn}[\![\underline{X := P}]\!] & = & \mathbf{psyn}[\![P]\!] & (8)\\
\mathbf{psyn}[\![\mathbf{0}]\!] & = & \bot_{\mathbf{syn}} & (9)
\end{array}
$$

Table 4.8 – The operator for potentially synchronizing labels $\mathbf{psyn} : \mathrm{IMC}^{\mathrm{G}} \to \mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$, $\bot_{\mathbf{syn}}((\ell_1,\ell_2)) = \emptyset$ for all $\ell_1 \in \mathbf{Lab}$ and $\ell_2 \in \mathbf{Lab}$.

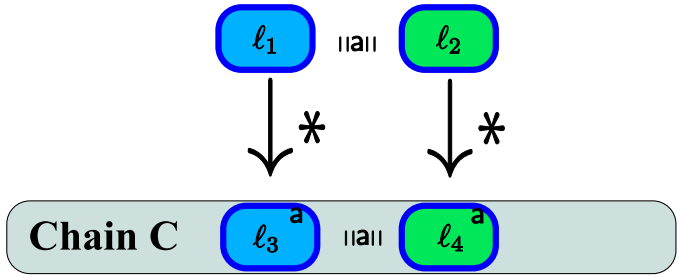

Figure 4.4 – Graphical illustration of the statement of Lemma 4.22.

Definition 4.21 is used in order to ease the formulation of Lemma 4.22.

Definition 4.21 (Label path)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9, a* label transition $\ell_1 \longrightarrow \ell_2$ *denotes the fact that* $\ell_2 \in dom(\mathcal{G}(\ell_1))$*. A* label path $\ell_1 \xrightarrow{*} \ell_n$ *denotes the fact that there exists a sequence of label transitions such that* $\ell_i \longrightarrow \ell_{i+1}$ *holds for all* $1 \leq i < n$*.*

Lemma 4.22 (**Syn** operator)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9,* $S_F = \mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}[\![F]\!]}}[\![F]\!]$*, it holds that:*

$$
\begin{array}{rcll}
\mathbf{Syn}_{R,\Phi}[\![\mathsf{a}^{\ell}.X]\!] & = & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell,\ell_2) \mapsto \mathbf{fn}_{\Phi}(\mathsf{a}^{\ell}.X) \cap R((\ell,\ell_2))\right] \sqcap & \\
& & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell_2,\ell) \mapsto \mathbf{fn}_{\Phi}(\mathsf{a}^{\ell}.X) \cap R((\ell_2,\ell))\right] & (1) \\
\mathbf{Syn}_{R,\Phi}[\![\lambda^{\ell}.X]\!] & = & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell,\ell_2) \mapsto \mathbf{fn}_{\Phi}(\lambda^{\ell}.X) \cap R((\ell,\ell_2))\right] \sqcap & \\
& & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell_2,\ell) \mapsto \mathbf{fn}_{\Phi}(\lambda^{\ell}.X) \cap R((\ell_2,\ell))\right] & (2) \\
\mathbf{Syn}_{R,\Phi}[\![\mathsf{a}^{\ell}.P]\!] & = & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell,\ell_2) \mapsto \mathbf{fn}_{\Phi}(\mathsf{a}^{\ell}.P) \cap R((\ell,\ell_2))\right] \sqcap & \\
& & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell_2,\ell) \mapsto \mathbf{fn}_{\Phi}(\mathsf{a}^{\ell}.P) \cap R((\ell_2,\ell))\right] \sqcap & \\
& & \mathbf{Syn}_{R,\Phi}[\![P]\!] & (3) \\
\mathbf{Syn}_{R,\Phi}[\![\lambda^{\ell}.P]\!] & = & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell,\ell_2) \mapsto \mathbf{fn}_{\Phi}(\lambda^{\ell}.P) \cap R((\ell,\ell_2))\right] \sqcap & \\
& & \displaystyle\bigsqcap_{\ell_2 \in \mathbf{Lab}} R\left[(\ell_2,\ell) \mapsto \mathbf{fn}_{\Phi}(\lambda^{\ell}.P) \cap R((\ell_2,\ell))\right] \sqcap & \\
& & \mathbf{Syn}_{R,\Phi}[\![P]\!] & (4) \\
\mathbf{Syn}_{R,\Phi}[\![P_1 + P_2]\!] & = & \mathbf{Syn}_{R,\Phi}[\![P_1]\!] \sqcap \mathbf{Syn}_{R,\Phi}[\![P_2]\!] & (5) \\
\mathbf{Syn}_{R,\Phi}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathbf{Syn}_{R,\Phi}[\![P]\!] & (6) \\
\mathbf{Syn}_{R,\Phi}[\![P_1 \shuffle{A} P_2]\!] & = & \mathbf{Syn}_{R,\Phi}[\![P_1]\!] \sqcap \mathbf{Syn}_{R,\Phi}[\![P_2]\!] & (7) \\
\mathbf{Syn}_{R,\Phi}[\![X := P]\!] & = & \mathbf{Syn}_{R,\Phi}[\![P]\!] & (8) \\
\mathbf{Syn}_{R,\Phi}[\![\mathbf{0}]\!] & = & R & (9)
\end{array}
$$

Table 4.9 – Inductive definition of the operator $\mathbf{Syn} : \mathrm{IMC}^{\mathrm{G}} \to \mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$. The environments $R \in \mathbf{Lab} \times \mathbf{Lab} \to 2^{\mathbf{Act}}$ and $\Phi \in \mathbf{Var} \to 2^{\mathbf{Act}}$ provided as parameters.

1. *for all* $\mathsf{a} \in S_F((\ell_1,\ell_2))$
   (a) *there exists a label path* $\ell_1 \xrightarrow{*} \ell_3$ *with* $Name(\ell_3) = \mathsf{a}$*;*
   (b) *there exists a label path* $\ell_2 \xrightarrow{*} \ell_4$ *with* $Name(\ell_4) = \mathsf{a}$*;*
   (c) *for all label paths* $\ell_1 \xrightarrow{*} \ell_3$ *and* $\ell_2 \xrightarrow{*} \ell_4$*, both* ℓ_3 *and* ℓ_4 *active,* $Name(\ell_3) = \mathsf{a}$ *and* $Name(\ell_4) = \mathsf{a}$ *holds:* $\bot_{\mathfrak{M}}[\ell_3 \mapsto 1][\ell_4 \mapsto 1] \leq C$ *for some* $C \in \mathfrak{T}$ *such that* $Names(C) = \mathsf{a}$*;*
2. *if conditions 1(a) and 1(b) are fulfilled for two labels* ℓ_1 *and* ℓ_2*, such that* $\ell_1 \in Labs(P_1)$ *and* $\ell_2 \in Labs(P_2)$ *for some* $P_1 \shuffle{A} P_2 \preceq F$ *and there exists a chain* $C \in \mathfrak{T}$ *such that* $\bot_{\mathfrak{M}}[\ell_3 \mapsto 1][\ell_4 \mapsto 1] \leq C$*, then also* $\mathsf{a} \in S_F((\ell_1,\ell_2))$ *holds.*

Proof. The statement 1 holds essentially due to the well-formedness condition for F, more precisely the condition (7) in Table 3.8. According to the latter, in any subexpression P_1 of F put in parallel with another subsexpression P_2 (and there exist such expressions with $\ell_1 \in Labs(P_1)$ and $\ell_2 \in Labs(P_2)$ according to the definition of the operator **psyn** in Table 4.8) there cannot be an action name which is both free and hidden. Moreover, from the same condition (7) follows that both P_1 and P_2 are closed. From this we can easily deduce that all label paths starting from labels in P_1 only involve labels in P_1. The same holds for P_2. The statement 1 follows then from the definition of the operators **psyn** and **Syn** and from the fact that both ℓ_3 and ℓ_4 are active, i.e. participate in at least one chain.

We will now prove the statement 1 more formally. We will first prove an instrumental statement that in case $Name(\ell) = Name(\ell') = \mathsf{a}$ for some active $\{\ell, \ell'\} \subseteq Labs(F)$ then $\mathsf{a} \in S_F((\ell, \ell'))$ if and only if there exists $C \in \mathfrak{T}$ such that $\bot_{\mathfrak{M}} [\ell \mapsto 1] [\ell' \mapsto 1] \leq C$. We prove the statement by induction on the structure of F. The statement is clear for the base cases – the rules (1)-(2) and (9) in Table 3.1 – as there are no chains and $S_F = \bot_{\mathbf{syn}}$ for them. Most of the other rules follow from the induction hypothesis except rule (7) for the parallel operator. Let us prove the instrumental statement for some $P_1 \shuffle A \shuffle P_2 \preceq F$ assuming that it holds both for P_1 and P_2.

If $\mathsf{a} \notin A$ and for example $\{\ell, \ell'\} \subseteq Labs(P_1)$ then the statement follows from the induction hypothesis: for such a pair (ℓ, ℓ') the output of $\mathbf{Syn}_{R,\Phi}[\![P_1]\!]$ is not changed according to the rule 8 in Table 4.9 while computing $\mathbf{Syn}_{R,\Phi}[\![P_1 \shuffle A \shuffle P_2]\!]$, taking into account that $Labs(P_1) \cap Labs(P_2) = \emptyset$, and the induction hypothesis is applicable. Otherwise assume that $\mathsf{a} \in A$ and for example $\{\ell, \ell'\} \subseteq Labs(P_1)$. Then from the induction hypothesis follows that $\mathsf{a} \in S_{P_1}((\ell, \ell'))$ iff there exists a chain $C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!]$ containing ℓ and ℓ'. As both labels are active, there also exists $C \in \mathfrak{T}_\Lambda[\![P_1 \shuffle A \shuffle P_2]\!]$ with $C_1 < C$ and therefore containing ℓ and ℓ'. Proving the other direction, from the existence of a chain $C \in \mathfrak{T}_\Lambda[\![P_1 \shuffle A \shuffle P_2]\!]$ follows the existence of some $C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!]$ containing ℓ and ℓ' and the induction hypothesis is applicable. The case with $\{\ell, \ell'\} \subseteq Labs(P_2)$ is symmetric.

Assume that $\mathsf{a} \in A$, $\ell \in Labs(P_1)$ and $\ell' \in Labs(P_2)$. Then if both ℓ and ℓ' are active this means that there exist chains $C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!]$ with $\ell \in dom(C_1)$ and $C_2 \in \mathfrak{T}_\Lambda[\![P_2]\!]$ with $\ell' \in dom(C_2)$, therefore there exists a chain $C = C_1 + C_2 \in \mathfrak{T}_\Lambda[\![P_1 \shuffle A \shuffle P_2]\!]$ with both ℓ and ℓ'. On the other hand, if there exists such a chain $C \in \mathfrak{T}_\Lambda[\![P_1 \shuffle A \shuffle P_2]\!]$ containing both ℓ and ℓ' then there exist chains $C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!]$ with $\ell \in dom(C_1)$ and $C_2 \in \mathfrak{T}_\Lambda[\![P_2]\!]$ with $\ell' \in dom(C_2)$ and the induction hypothesis is applicable. If $\mathsf{a} \notin A$, $\ell \in Labs(P_1)$ and $\ell' \in Labs(P_2)$ then according to rule (7) in Table 4.8 holds $\mathsf{a} \notin S_{P_1 \shuffle A \shuffle P_2}((\ell, \ell'))$ and there is also no common chains with ℓ and ℓ' according to the rules for chains calculation.

We have herewith proved the instrumental statement. We need to additionally show that from $\mathsf{a} \in S_F((\ell_1, \ell_2))$ follow 1(a) and 1(b). This can be deduced from the rules (1)-(4) for the operator **Syn** in Table 4.9: there exist some labels corresponding to the action name a either in the definition of X (rules (1)-(2)) or in the process P (rules (3)-(4)). Another statement that we should prove using the rules for the operator **Syn** is that for all ℓ_1' and ℓ_2' such that $\ell_1 \xrightarrow{*} \ell_1' \xrightarrow{*} \ell_3$ and $\ell_2 \xrightarrow{*} \ell_2' \xrightarrow{*} \ell_4$ holds $\mathsf{a} \in S_F((\ell_1', \ell_2'))$ – but this can also be proved using the rules (1)-(4) of the operator **Syn** in Table 4.9. Altogether this is enough in order to show the correctness of the statement 1 of the lemma.

The statement 2 can be proved by the following reasoning: choose two label paths $\ell_1 \xrightarrow{*} \ell_3$ and $\ell_2 \xrightarrow{*} \ell_4$ satisfying conditions 1(a) and 1(b). From the well-formedness rule (7) in Table 3.8, in particular process identifier closeness of P_1 and P_2, follows that also $\ell_3 \in Labs(P_1)$ and $\ell_4 \in Labs(P_2)$. From the rules for the chains operator in Table 3.17 follows that a is free both in P_1 and P_2. Consequently $\mathsf{a} \in \mathbf{psyn}[\![F]\!]((\ell_1, \ell_2))$ and this pair has not been deleted while computing the results of **Syn** afterwards according to the rules (1)-(4) in Table 4.9. □

Remark 4.23. *Note that from the proof of Lemma 4.22 it is clear that in fact either for all label paths $\ell_1 \xrightarrow{*} \ell_3$ and $\ell_2 \xrightarrow{*} \ell_4$ with $\ell_1 \neq \ell_2$, $Name(\ell_3) = \mathsf{a}$ and $Name(\ell_4) = \mathsf{a}$ there exists a chain $C \in \mathfrak{T}$ such that $\bot_{\mathfrak{M}}[\ell_3 \mapsto 1][\ell_4 \mapsto 1] \leq C$ or for none of such paths. In the first case we will have $\mathsf{a} \in S_F((\ell_1, \ell_2))$ and in the second case – $\mathsf{a} \notin S_F((\ell_1, \ell_2))$. We cannot have two pairs of paths with the same "origins" such that for the first pair there is a chain (i.e. synchronisation on the action a) and for the second there is no such synchronisation.*

Also note that in order to compute $\mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]}[\![F]\!]$ we need to traverse the syntactic description of F accordingly three times: to compute $\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]$, then $\mathbf{psyn}[\![F]\!]$ and then use the both results as environments to compute $\mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]}[\![F]\!]$. In practice we could actually combine the operators in order to limit ourselves to only one traversal. We have used several operators in order to make the exposition and the proofs easier. Especially in the proofs we would need to additionally argue that while traversing the syntax of F the environments are being updated with the information before the time point when the information is actually utilised.

Altogether we have proved that the computed $\mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\bot_{\mathbf{fnvar}}}[\![F]\!]}[\![F]\!]$ is a reasonable over-approximation of what we have called a synchronisation structure of F: it takes into account all cases of generated labels that can jointly

participate in chains, but does not take into account that some labels may be blocked by the environment and therefore will never be executed – therefore the computed result is an over-approximation.

4.4.2 Label and multiset synchronisation-bisimulations

We will introduce relations on labels and multisets of a special kind which we will call *label and multiset synchronisation-bisimulations*. We will have a fixed $\mathrm{IMC}^{\mathrm{G}}$ program F without rates for which the Pathway Analysis has been conducted and its the synchronising structure, i.e. $\mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\perp_{\mathbf{fnvar}}[\![F]\!]}[\![F]\!]}[\![F]\!]$, has been computed. Based on the results of the Pathway Analysis and the synchronising structure we then will be able to determine if two arbitrary interdependent binary relations on accordingly labels and multisets can be called synchronisation-bisimulations.

As usual we will call the coarsest label/multiset synchronisation-bisimulations *synchronisation-bisimilarities*. We will prove that each equivalence relation which is a synchronisation-bisimulation is a bisimulation relation on $\mathfrak{M}$. Moreover, we will prove that synchronisation-bisimilarities on multisets are strictly bigger than chain-bisimilarities on multisets for the same initial $\mathrm{IMC}^{\mathrm{G}}$ program.

We start by formally defining label and multiset synchronisation-bisimulations. We have divided the conditions for better readability into the conditions referring to the "label part" and the conditions referring to the "multiset part", even though they are interconnected.

Definitions of label/multiset synchronisation-bisimulations

Definition 4.24. *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9,* $S_F = \mathbf{Syn}_{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\perp_{\mathbf{fnvar}}[\![F]\!]}[\![F]\!]}[\![F]\!]$*, we call a binary relation* R *on labels and a binary relation* R' *on multisets accordingly* label synchronisation-bisimulation *and* multiset synchronisation-bisimulation *if the conditions (A1)-(A5) and (B1)-(B4) hold.*

If $\ell_1 R \ell_2$ *then:*

(A1) both labels are active, i.e. there exist $C_1 \in \mathfrak{T}$ *and* $C_2 \in \mathfrak{T}$ *such that* $\perp_{\mathfrak{M}} [\ell_1 \mapsto 1] \leq C_1$ *and* $\perp_{\mathfrak{M}} [\ell_2 \mapsto 1] \leq C_2$*;*

(A2) $Name^h(\ell_1) = Name^h(\ell_2)$ *and additionally* $Name(\ell_1) = Name(\ell_2)$ *if* $\exists C \in \mathfrak{T}$ *s.t.* $\ell_1 \in dom(C)$ *or* $\ell_2 \in dom(C)$ *and* $|C| > 1$*;*

(A3) $\{|C| \mid (C \in \mathfrak{T}) \wedge (\ell_1 \in dom(C))\} = \{|C| \mid (C \in \mathfrak{T}) \wedge (\ell_2 \in dom(C))\}$*;*

(A4) for all $M_1 R' M_2$ *such that* $\ell_1 \in dom(M_1)$ *and* $\ell_2 \in dom(M_2)$ *holds:* $M_1 \in \mathfrak{T}$ *iff* $M_2 \in \mathfrak{T}$*;*

(A5) $G(\ell_1) R' G(\ell_2)$*.*

If $M_1 R' M_2$ *then there exist sums* $M_1 = \sum_{i=1..n} M_1^i + M_1'$ *and* $M_2 = \sum_{i=1..n} M_2^i + M_2'$ *such that:*

(B1) all labels in M_1' *and* M_2' *are not active;*

(B2) for all $1 \leq i \leq n$*,* $\ell_1 \in M_1^i$*,* $\ell_2 \in M_2^i$ *holds:* $\ell_1 R \ell_2$*;*

(B3) for all $1 \leq i \leq n$*,* $1 \leq k \leq 2$ *holds: if* $|M_k^i| > 1$ *then* $\forall \ell \in M_k^i$ *holds* $M_k^i = \sum_{(\ell' \in dom(K(\ell))) \wedge (\ell' \in dom(M_k))} \perp_{\mathfrak{M}} [\ell' \mapsto 1]$*;*

(B4) for all $1 \leq i, j \leq n$*, all* $\ell_1 \in dom(M_1^i)$*,* $\ell_2 \in dom(M_1^j)$*,* $\ell_3 \in dom(M_2^i)$ *and* $\ell_4 \in dom(M_2^j)$ *holds* $S((\ell_1, \ell_2)) = S((\ell_3, \ell_4))$*.*

We call the coarsest relations R *and* R' *fulfilling the conditions above* label synchronisation-bisimilarity *and* multiset synchronisation-bisimilarity *accordingly. We denote* $\ell_1 \sim^{syn}_{\mathbf{Lab}} \ell_2$ *(label synchronisation-bisimilar) and* $M_1 \sim^{syn}_{\mathfrak{M}} M_2$ *(multiset synchronisation-bisimilar) if* (ℓ_1, ℓ_2) *is contained in some label synchronisation-bisimulation and* (M_1, M_2) *is contained in a corresponding to it multiset synchronisation-bisimulation.*

Computing the coarsest label synchronisation-bisimulation requires the computation of $S_F = \mathbf{Syn}_{\mathbf{psyn}[\![F]\!], \mathbf{fnvar}_{\perp_{\mathbf{fnvar}}[\![F]\!]}}[\![F]\!]$ beforehand. From Tables 4.7, 4.8 and 4.9 we can see that the time complexity of the computation of $\mathbf{fnvar}_{\mathbf{\Phi}}[\![F]\!]$ is linear in the syntax of F for any environment while the computation of $\mathbf{psyn}[\![F]\!]$ and $\mathbf{Syn}_{R,\Phi}[\![F]\!]$ (given the fixed environments) are quadratic in the syntax of F, therefore as a whole the time complexity of computing S_F is quadratic in the syntax of F.

We can assess the time and space complexities of computing the coarsest label synchronisation-bisimulation in a similar way as it has been done for computing

the coarsest label chain-bisimulation in Section 4.3.2. The initial partition of labels can be done based on the conditions (A1)-(A3). The further refinement of partition classes should take into account the conditions (A4)-(A5). The time complexity of computing the coarsest label chain-bisimulation will be then $O(m * log(n))$ and the space complexity will be $O(m)$, where $n = |Labs(F)|$ and m is the number of generate-relations plus the number of chains in $\mathfrak{T}$, i.e. $m = |\{\ell_1 \in Labs(F), \ell_2 \in Labs(F) | \ell_1 \longrightarrow \ell_2\}| + |\mathfrak{T}|$. Deciding whether two multisets are synchronisation-bisimilar can be then done in the linear time in the number of labels in the domains of two multisets.

Lemma 4.25 (Synchronisation-bisimilarities)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9, the label and multiset synchronisation-bisimilarities for* F *are equal accordingly to the unions of all label and multiset synchronisation-bisimulation relations for* F*.*

Proof. The proof is a usual proof of bisimilarities being the union of all bisimulations, see for example [BK08]. We have to show that the unions of all label/multiset synchronisation-bisimulation relations are accordingly label/multiset synchronisation-bisimulations itself. This can be done by going through Definition 4.24 and showing that all the conditions are fulfilled also for unions of relations. The unions are on the other hand the coarsest synchronisation-bisimulations because there cannot exist any strictly coarser relations fulfilling all the conditions in Definition 4.24 as they would be included in the unions. □

Properties of multiset synchronisation-bisimilarity

We will show the properties of synchronisation-bisimilarities relation on multisets (given a fixed $\mathrm{IMC}^{\mathrm{G}}$ program F). These are relations that are the most interesting for us: as the coarsest synchronisation-bisimulations they captures the most cases of the similar behaviour of states in the labelled transition system representing the semantics of F. We will prove the following statements:

1. the relation $\sim^{syn}_{\mathfrak{M}}$ is an equivalence relation on multisets;
2. any multiset synchronisation-bisimulation which is an equivalence, therefore also $\sim^{syn}_{\mathfrak{M}}$, is a bisimulation relation on $\mathfrak{M}$;

Lemma 4.26 (Synchronisation-bisimilarities are equivalences)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9, the relations* $\sim^{syn}_{\boldsymbol{Lab}}$ *and* $\sim^{syn}_{\mathfrak{M}}$ *are equivalence relations on accordingly the active labels from* $\boldsymbol{Lab}$ *and on* $\mathfrak{M}$*.*

Proof. We need to prove that the relations $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$ on active labels and on multisets are reflexive, symmetric and transitive.

Let the relations $\mathcal{I}_{\mathbf{Lab}} \subseteq \mathbf{Lab} \times \mathbf{Lab}$ and $\mathcal{I}_{\mathfrak{M}} \subseteq \mathfrak{M} \times \mathfrak{M}$ be the identity relations on the active elements of $\mathbf{Lab}$ and on $\mathfrak{M}$ accordingly. Then all conditions from Definition 4.24 are fulfilled for both $\mathcal{I}_{\mathbf{Lab}}$ and $\mathcal{I}_{\mathfrak{M}}$, therefore $\mathcal{I}_{\mathbf{Lab}} \subseteq \sim^{syn}_{\mathbf{Lab}}$ and $\mathcal{I}_{\mathfrak{M}} \subseteq \sim^{syn}_{\mathfrak{M}}$ (according to Lemma 4.25 the relations $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$ are unions of all label/multiset synchronisation-bisimulations) and the relations $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$ are therefore reflexive.

Assume that there are some label/multiset synchronisation-bisimulations $\mathcal{R} \subseteq \mathbf{Lab} \times \mathbf{Lab}$ and $\mathcal{R}' \subseteq \mathfrak{M} \times \mathfrak{M}$. We will define the relations $\mathcal{R}^{-1}$ and $\mathcal{R}'^{-1}$ in the following way: $(\ell_2, \ell_1) \in \mathcal{R}^{-1}$ iff $(\ell_1, \ell_2) \in \mathcal{R}$ and $(M_2, M_1) \in \mathcal{R}'^{-1}$ iff $(M_1, M_2) \in \mathcal{R}'$. Then the relations $\mathcal{R}^{-1}$ and $\mathcal{R}'^{-1}$ satisfy all the conditions in Definition 4.24. This holds for all label/multiset synchronisation-bisimulations – therefore $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$ are symmetric as they are the unions of all label/multiset synchronisation-bisimulations.

Similar arguments hold for transitivity: assume that there is a pair of label/multiset synchronisation-bisimulations $\mathcal{R}_1 \subseteq \mathbf{Lab} \times \mathbf{Lab}$, $\mathcal{R}'_1 \subseteq \mathfrak{M} \times \mathfrak{M}$, $\mathcal{R}_2 \subseteq \mathbf{Lab} \times \mathbf{Lab}$, and $\mathcal{R}'_2 \subseteq \mathfrak{M} \times \mathfrak{M}$. Then the relation $\mathcal{R}_1 \circ \mathcal{R}_2$ contains a pair (ℓ_1, ℓ_2) iff there exist some pairs $(\ell_1, \ell_3) \in \mathcal{R}_1$ and $(\ell_3, \ell_2) \in \mathcal{R}_2$. The relation $\mathcal{R}'_1 \circ \mathcal{R}'_2$ contains a pair (M_1, M_2) iff there exist some pairs $(M_1, M_3) \in \mathcal{R}'_1$ and $(M_3, M_2) \in \mathcal{R}'_2$. The relations $\mathcal{R}_1 \circ \mathcal{R}_2$ and $\mathcal{R}'_1 \circ \mathcal{R}'_2$ satisfy all conditions in Definition 4.24. As this holds for all label/multiset synchronisation-bisimulations and $\sim^{syn}_{\mathbf{Lab}}$ and $\sim^{syn}_{\mathfrak{M}}$ are the unions thereof, they are transitive as well. □

Another important fact that we will need in order to prove that synchronisation-bisimulations are bisimulations on $\mathfrak{M}$ is proved in Lemma 4.27.

Lemma 4.27 (**Syn** operator)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *without rates and* $\mathcal{F}$ *as in Definition 4.9,* $S := \mathbf{Syn}^{\mathbf{psyn}[\![F]\!],\mathbf{fnvar}_{\perp}}_{\mathbf{fnvar}[\![F]\!]}[\![F]\!]$ *and a pair of relations* (R, R') *that are accordingly a label and a multiset synchronisation-bisimulations, it holds that in case*

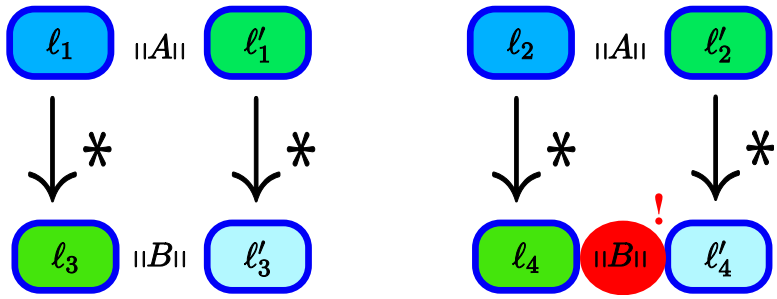

Figure 4.5 – Graphical representation of Lemma 4.27.

1. $\ell_1 R \ell_2$ *and* $\ell'_1 R \ell'_2$*;*
2. $S((\ell_1, \ell'_1)) = S((\ell_2, \ell'_2))$*;*
3. *there exist label paths* $\ell_1 \xrightarrow{*} \ell_3$, $\ell'_1 \xrightarrow{*} \ell'_3$, $\ell_2 \xrightarrow{*} \ell_4$ *and* $\ell'_2 \xrightarrow{*} \ell'_4$ *such that* $\ell_3 R \ell_4$ *and* $\ell'_3 R \ell'_4$

then $S((\ell_3, \ell'_3)) = S((\ell_4, \ell'_4))$.

Proof. The statement of the lemma is schematically represented in Figure 4.5. Synchronisation-bisimilar labels are denoted by the same colours; the statement to prove is brought out by the red circle. Note that, for example, $\ell_1 \text{ ||}A\text{|| } \ell'_1$ in Figure 4.5 actually means that $S((\ell_1, \ell'_1)) = A$.

In the proof of Lemma 4.22 it was mentioned that from $\mathsf{a} \in S_F((\ell_2, \ell'_2))$, $\ell_2 \xrightarrow{*} \ell_4 \xrightarrow{*} \ell_5$ and $\ell'_2 \xrightarrow{*} \ell'_4 \xrightarrow{*} \ell'_5$ with $Name(\ell_5) = Name(\ell'_5) = \mathsf{a}$ follows $\mathsf{a} \in S_F((\ell_4, \ell'_4))$. It is therefore enough to show that we have exactly this situation in the conditions of the lemma.

We can easily prove (from the definition of the operator **psyn**) that $B \subseteq A$, therefore for each $\mathsf{a} \in B$ holds $\mathsf{a} \in A$. Besides, if labels corresponding to a are reachable from ℓ_3 and ℓ'_3 (we have proved this in Lemma 4.22), for example, $\ell_3 \xrightarrow{*} \ell_6$ and $\ell'_3 \xrightarrow{*} \ell'_6$, then some labels bisimilar to them are reachable from ℓ_4 and ℓ'_4, i.e. there exist the labels ℓ_5 and ℓ'_5 reachable from ℓ_4 and ℓ'_4 such that $Name^h(\ell_5) = Name^h(\ell_6)$ and $Name^h(\ell'_5) = Name^h(\ell'_6)$. The last statement follows from the condition (A2) for bisimilar labels. It is however also the case that ℓ_5 and ℓ'_5 do not constitute chains on their own, because

ℓ_6 and ℓ'_6 do not constitute chains on their own (we have proved in Lemma 4.22 that they participate in a chain together). Therefore it also holds that $Name(\ell_5) = Name(\ell_6) = \mathsf{a}$ and $Name(\ell'_5) = Name(\ell'_6) = \mathsf{a}$. Altogether this is exactly the case described at the beginning of the proof and therefore $\mathsf{a} \in S_F((\ell_4, \ell'_4))$ holds.

□

We will prove now the main result of this section – that the synchronisation-bisimulation is a bisimulation itself.

Lemma 4.28 (Synchronisation-bisimulation is a bisimulation)**.** *Given an* IMC$^{\mathrm{G}}$ *program F without rates, $\mathcal{F}$ as in Definition 4.9 and a pair of equivalence relations (R, R') that are accordingly a label and a multiset synchronisation-bisimulations, then from $M_1 R' M_2$ follows $M_1 \sim M_2$ for any $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$. In particular, from $M_1 \sim^{syn}_{\mathfrak{M}} M_2$ follows $M_1 \sim M_2$ for any $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$.*

Proof. We need to show that the relation R' is a bisimulation on $\mathfrak{M}$ in the sense of Definition 4.11 if the syntax of the initial program F does not contain rates. This will be enough to show the statement of the lemma because the relation $\sim^{syn}_{\mathfrak{M}}$ is an equivalence relation according to Lemma 4.26 and it is a bisimulation according to Lemma 4.25.

As there are no rates in the syntax of F, it is enough to prove the condition 1 from Definition 4.11. From the condition A4 in Definition 4.24 follows that for any enabled chain $C_1 \leq M_1$ there exists an enabled chain $C_2 \leq M_2$: for $C_1 \leq M_1$ and $C_1 \in \mathfrak{T}$ follows that there exist a multiset $M \leq M_2$ such that $C_1 \sim^{syn}_{\mathbf{Lab}} M$ and $M \in \mathfrak{T}$. Both chains correspond either to τ or to the same external action name according to the condition A2 in Definition 4.24.

Further we can use the condition B3 in Definition 4.24 in order to prove that $(M_1 - K_{\mathfrak{M}}(C_1))R'(M_2 - K_{\mathfrak{M}}(C_2))$ as all components in blocks will be killed by the execution of the chains. The condition A5 in Definition 4.24 proves $G_{\mathfrak{M}}(C_1)R'G_{\mathfrak{M}}(C_2)$ and, similarly to the discussion in the proof of Lemma 4.20, we can determine that all newly generated labels can neither be killed or kill already exposed labels – this can be shown by induction on the syntactic structure of F. Together with Lemma 4.27, from which follows (speaking in a simplified way) that all synchronising relations between "newly generated" active

labels and the already present ones in M_1 and M_2 are the same, this proves $(M_1 - K_{\mathfrak{M}}(C_1) + G_{\mathfrak{M}}(C_1))R'(M_2 - K_{\mathfrak{M}}(C_2) + G_{\mathfrak{M}}(C_2))$ and this fulfilles the condition 1 in Definition 4.11. □

4.4.3 Synchronisation-, chain- and "ordinary" bisimulations

We will prove that the relation $\sim^{ch}_{\mathfrak{M}}$ defined in Section 4.3.2 is contained in $\sim^{syn}_{\mathfrak{M}}$ and is in general strictly smaller than the latter. This will demonstrate that we are not "losing" any information, i.e. any pairs of bisimilar states, while moving from the chain-bisimilarity to the synchronisation-bisimilarity.

Lemma 4.29 (Chain- and synchronisation-bisimilarities)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program F without rates and $\mathcal{F}$ as in Definition 4.9, then from $M_1 \sim^{ch}_{\mathfrak{M}} M_2$ follows $M_1 \sim^{syn}_{\mathfrak{M}} M_2$ for any $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$.*

Proof. It is enough to show that the relations $\sim^{ch}_{\mathbf{Lab}}$ and $\sim^{ch}_{\mathfrak{M}}$ are accordingly a label synchronisation-bisimulation and a multiset synchronisation-bisimulation. This will obviously prove the statement of the lemma as any label/multiset synchronisation-bisimulation is included in the label/multiset synchronisation-bisimilarities.

Most of the conditions in the Definition 4.24 clearly hold. For example, A3 holds because either chain-bisimilar labels participate in the same chains or are interchangeable, therefore the sizes of their chains are the same. The condition A4 for label synchronisation-bisimulation follows from Lemma 4.19: from this lemma follows that if all labels in two multisets are pairwise connected by the relation $\sim^{ch}_{\mathbf{Lab}}$ and one of the multisets is a chain, then the other one is also a chain. The condition B3 is true because we can assign all M^i_ks to consist of only one label such that for all $M^i_1 = \{\ell\}$ and $M^i_2 = \{\ell'\}$ it holds that $\ell \sim^{ch}_{\mathbf{Lab}} \ell'$.

It is left to prove the condition B4 in the definition of multiset synchronisation-bisimulation. We show it schematically in Figure 4.6, where $\shortparallel A \shortparallel$ denotes the fact that $S((\ell_1, \ell'_1)) = A$ and the red circle emphasises the fact to be proved.

Due to the symmetry it is enough to prove that for any $\mathsf{a} \in S((\ell_1, \ell'_1))$ also holds $\mathsf{a} \in S((\ell_2, \ell'_2))$. We can prove the statement by contradiction. Assume that we have a situation where $\ell_1 \sim^{ch}_{\mathbf{Lab}} \ell_2$, $\ell'_1 \sim^{ch}_{\mathbf{Lab}} \ell'_2$, $\mathsf{a} \in S((\ell_1, \ell'_1))$ but

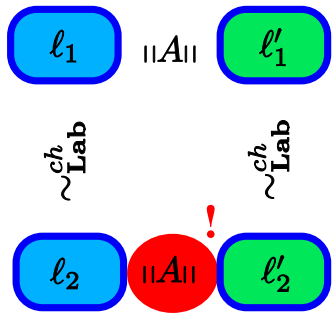

Figure 4.6 – Graphical representation of the proof for condition B4 in the definition of multiset synchronisation-bisimulation (Definition 4.24) in Lemma 4.29.

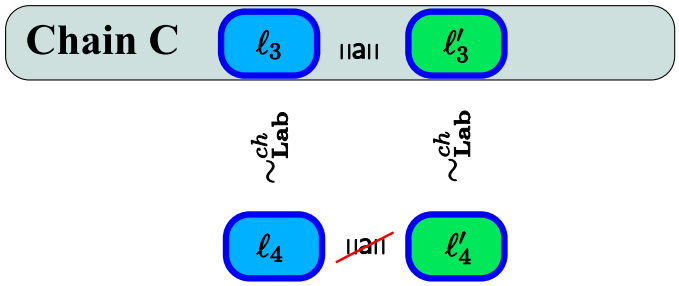

Figure 4.7 – Illustration of the proof by contradiction of Lemma 4.29.

$\mathsf{a} \notin S((\ell_2, \ell'_2))$. Then according to Lemma 4.22 there exist label paths $\ell_1 \xrightarrow{*} \ell_3$ and $\ell'_1 \xrightarrow{*} \ell'_3$ such that $\perp_{\mathfrak{M}} [\ell_3 \mapsto 1, \ell'_3 \mapsto 1] \leq C$ for some $C \in \mathfrak{T}$. From the conditions of chain-bisimilarity it will follow that there exist corresponding label paths $\ell_2 \xrightarrow{*} \ell_4$ and $\ell'_2 \xrightarrow{*} \ell'_4$, with $\ell_3 \sim^{ch}_{\mathbf{Lab}} \ell_4$ and $\ell'_3 \sim^{ch}_{\mathbf{Lab}} \ell'_4$, however there does not exist any chain $C' \in \mathfrak{T}$ such that $\perp_{\mathfrak{M}} [\ell_4 \mapsto 1, \ell'_4 \mapsto 1] \leq C'$ (otherwise from Lemma 4.22 would follow $\mathsf{a} \in S((\ell_2, \ell'_2))$). Schematically the situation is represented in Figure 4.7.

We show that this is impossible. Assume that both $\ell_3 \sim^{ch}_{\mathbf{Lab}} \ell_4$ and $\ell'_3 \sim^{ch}_{\mathbf{Lab}} \ell'_4$ are caused by ℓ_3 being interchangeable with ℓ_4 and by ℓ'_3 being interchangeable with ℓ'_4. We can make a conclusion that a chain $C' := C - \perp_{\mathfrak{M}} [\ell_3 \mapsto 1, \ell'_3 \mapsto 1] + \perp_{\mathfrak{M}} [\ell_4 \mapsto 1, \ell'_4 \mapsto 1]$ exists in which both ℓ_4 and ℓ'_4 participate and this will contradict our previous assumptions. If on the other hand ℓ_3 is interchangeable with ℓ_4, but ℓ'_3 participates together with ℓ'_4 in all the chains then $C' := C - \perp_{\mathfrak{M}} [\ell_3 \mapsto 1] + \perp_{\mathfrak{M}} [\ell_4 \mapsto 1]$ is a chain and $\perp_{\mathfrak{M}} [\ell_4 \mapsto 1, \ell'_3 \mapsto 1, \ell'_4 \mapsto 1] \leq C'$ which is again a contradiction. The last case is symmetrical. We can deduce

that there necessary exists a chain in which both ℓ_4 and ℓ_4' participate and therefore our initial assumption was wrong. □

Note that $\sim_M^{ch} \neq \sim_M^{syn}$ in general. This is first of all due to more permissive bisimulation relation on labels: it can be the case that $\sim_{\mathbf{Lab}}^{ch} \neq \sim_{\mathbf{Lab}}^{syn}$. For example, in the IMC$^{\mathrm{G}}$ expression $(\mathsf{b}^{\ell_5}.\mathsf{a}^{\ell_1}.\mathbf{0} \parallel \mathsf{a} \parallel \mathsf{c}^{\ell_6}.\mathsf{a}^{\ell_2}.\mathbf{0}) \parallel \emptyset \parallel (\mathsf{d}^{\ell_7}.\mathsf{a}^{\ell_3}.\mathbf{0} \parallel \mathsf{a} \parallel \mathsf{e}^{\ell_8}.\mathsf{a}^{\ell_4}.\mathbf{0})$ it holds $\mathsf{a}^{\ell_i} \sim_{\mathbf{Lab}}^{syn} \mathsf{a}^{\ell_i}$ for any $1 \leq i, j \leq 4$ while it only holds $\mathsf{a}^{\ell_1} \sim_{\mathbf{Lab}}^{ch} \mathsf{a}^{\ell_2}$ and $\mathsf{a}^{\ell_3} \sim_{\mathbf{Lab}}^{ch} \mathsf{a}^{\ell_4}$. We have added the additional actions b^{ℓ_5}, c^{ℓ_6}, d^{ℓ_7} and e^{ℓ_8} in order to illustrate that all a^{ℓ_i}s may be exposed independently of each other.

Also note that the relation $\sim_{\mathbf{Lab}}^{syn}$ in contrast to the relation $\sim_{\mathbf{Lab}}^{ch}$ is "not enough" anymore to determine bisimilar multisets. For instance, in the example above we would consider $\bot_{\mathfrak{M}} [\ell_1 \mapsto 1] [\ell_2 \mapsto 1]$ and $\bot_{\mathfrak{M}} [\ell_1 \mapsto 1] [\ell_3 \mapsto 1]$ bisimilar based only on the relation $\sim_{\mathbf{Lab}}^{syn}$ but in fact they are not. Therefore we additionally need the results of the operator **Syn** in order to to "narrow down" the equivalence relation on the multisets to a bisimulation relation.

We could further extend the bisimulation relation on multisets in several ways. We could identify labels with different external action names that have been internalised. We do this for the synchronisation bisimulation only for labels that constitute chains on their own. We cannot extend this identification to the labels that do not constitute chains on their own directly because the operator **Syn** should be adapted. For example, the processes $\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \underline{Y := \mathsf{a}^{\ell_2}.Y}$ and $\underline{X := \mathsf{b}^{\ell_3}.X} \parallel \mathsf{b} \parallel \underline{Y := \mathsf{b}^{\ell_4}.Y}$ could only be identified if the operator **Syn** can determine that ℓ_1 and ℓ_2 synchronising on a are equivalent to ℓ_3 and ℓ_4 synchronising on b.

Another possible improvement is identifying sets of parallel labels that have an equivalent behaviour. For example, in the IMC$^{\mathrm{G}}$ process $(\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \underline{Y := \mathsf{a}^{\ell_2}.Y}) + \underline{Z := \mathsf{a}^{\ell_3}.Z}$ we could realise that $\{\ell_1, \ell_2\}$ and $\{\ell_3\}$ behave in the same way. Note that our synchronisation-bisimilarity will not relate different numbers of parallel labels to each other. The question remains however how to construct a bisimulation relation which will stay both sound and scalable while checking all n-to-m label correspondences.

It is also very desirable to extend our compositional approach of constructing bisimulation relations to IMC$^{\mathrm{G}}$ programs with rates. It will not be easy because while determining whether two states are bisimilar their joint rates of transitions to classes of bisimilar states should be compared. This means that all rates in a state should be known – which is not necessary for the actions.

An interesting topic for future work is the question concerning subclasses of $\mathrm{IMC}^{\mathrm{G}}$ for which we can construct bisimilarities with our methods, i.e. in which cases our methods will have the highest "gain" compared to the usual methods of identifying bisimilar states. We conjecture that synchronisation-bisimilarities coincide with the bisimilarity on $\mathfrak{M}$ first for the same subclasses of $\mathrm{IMC}^{\mathrm{G}}$ as chain-bisimilarities – i.e. for the linear fragment of $\mathrm{IMC}^{\mathrm{G}}$ without rates, bisimilar states in a symmetrical composition in the sense of [Her02] with linear fragments not necessary syntactically equal but with the same semantics. Moreover, we can recognise the bisimilarity of two processes with several different (but bisimilar) subprocesses in parallel and the same synchronisation structure – for example, if none of the actions is blocked by the environment. We leave therefore the identification of subclasses of $\mathrm{IMC}^{\mathrm{G}}$ for which the synchronisation-bisimilarity coincides with the bisimilarity on $\mathfrak{M}$ as a possible direction for future work.

Chapter 5

Reachability and Pathway Analysis

In this chapter we will discuss how the results of the Pathway Analysis of IMC$^{\mathrm{G}}$ can be used in order to answer some questions which usually fall into the model checking domain. At the beginning of the chapter in Section 5.1 we will discuss how the so-called *valid* states can be determined. The idea is to be able to determine that some configurations of exposed labels are impossible ("not valid", not reachable) and therefore can be excluded from the further analysis. This can simplify matters a lot, because instead of the naive approach of considering $2^{|Labs(F)|}$ states of the labelled transition system representing the semantics of some IMC$^{\mathrm{G}}$ program F we will not consider those states with exposed label from $2^{|Labs(F)|}$ that definitely cannot be reached from the initial state representing the exposed labels of F. Note that we are identifying in our discussion IMC$^{\mathrm{G}}$ programs and their derivatives on the one hand and their exposed labels on the other hand: we can do this in case the initial IMC$^{\mathrm{G}}$ program F and its Pathway Analysis results in $\mathcal{F}$ as Definition 4.9 are fixed, because the Pathway Analysis on IMC$^{\mathrm{G}}$ programs is correct and precise (see Section 3.5).

The idea is to exploit the fact that the presence of some labels in the domain of a label multiset which represents exposed labels of a derivative of F requires the presence of some other labels and implies the absence of some other labels. We determine these kinds of implications by using Static Analysis, i.e. by applying

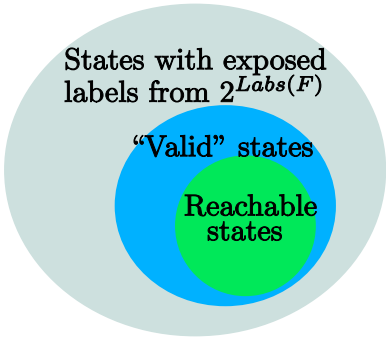

Figure 5.1 – Illustration of the relation between "valid" and reachable states, given an IMC$^{\mathrm{G}}$ program F.

several operators defined by us to the syntactic description of F and combining the results returned by them into the function *Valid* which returns a boolean value. We get an over-approximation of valid states, i.e. in general some states for which the function *Valid* returns *true* are in fact unreachable from $\mathcal{E}_\Gamma[\![F]\!]$. It is however always the case that states for which *Valid* returns *false* cannot be reached from $\mathcal{E}_\Gamma[\![F]\!]$. See Figure 5.1 for an illustration.

Further developments in this chapter include an illustration of how the well-known model-checking algorithms can be transferred to the Pathway Analysis setting. We consider as an example so-called *Zeno* states and an algorithm for computing a set of all *Zeno* states in Section 5.2. An IMC$^{\mathrm{G}}$ expression E is called Zeno if there exists an infinite path starting from E such that all transitions are decorated by τ, i.e. are internal transitions. Such behaviour is often considered unrealistic in concrete systems (no time passes and the action transitions also cannot be blocked by the environment as all the actions are internal) – therefore checking a model for the absence of Zeno states represents an important task. We transfer the definition of Zeno states and the standard algorithm for their computation to our setting of label multisets, with some small adjustments.

In the algorithm for computing label multisets that are Zeno, i.e. correspond to exposed labels of Zeno IMC$^{\mathrm{G}}$ expressions derivable from an IMC$^{\mathrm{G}}$ program F for which the Pathway Analysis has been conducted, we are utilising the information on valid states returned by the described above function *Valid*. As the function *Valid* returns an over-approximation of states reachable from an initial IMC$^{\mathrm{G}}$ program F, it could in fact be the case that some of the Zeno states computed by us are simply unreachable – an additional check is required in order to determine whether they are in fact reachable or not. On the other

hand, if we do not find any Zeno states then we can be sure that the labelled transition system representing the semantics of F does not contain any Zeno states.

Consequently we discuss similar problems – for example, model-checking a system for deadlock states – that are even simpler than computing Zeno states. Finding deadlock states requires only direct inspection of enabled chains in all valid states. If no deadlock states have been found, then we can be sure that the system is deadlock-free, otherwise the reachability of the found deadlock states should be additionally checked.

In the following sections we discuss the questions of reachability, both within an unlimited time (Section 5.3) and "timed" reachability (Section 5.4). We determine which labels are (in an over-approximation of) potentially reachable labels from an initial set of exposed labels (Section 5.3.1) and show how to update our approximation after one semantic step without recomputing everything from scratch (Section 5.3.2). We also touch upon the question of how to determine which labels are not just reachable a finite number of times but can be generated or executed an infinite number of times – they are in some sense "renewable resources" (Section 5.3.3).

Finally, we will compute the minimum expected time to reach one set of labels from another set in Section 5.4. We can only compute the *expected* time for $\mathrm{IMC}^{\mathrm{G}}$ processes because the durations of transitions are not fixed but are exponentially distributed instead. "Minimum" refers to the fact that we compute the expected time not for one fixed scheduler but for a class of schedulers – see Section 2.3 for a discussion on schedulers. We consider only so-called *simple*, also called *positional* schedulers (see, for example, [Put94] or [NSK09]) which are both history-independent and time-abstract, and therefore are especially simple to analyse – hence the name. We also do not do the computations for all $\mathrm{IMC}^{\mathrm{G}}$ programs but only for linear ones, with an additional syntactic restriction. We are however confident that our method is extendable to a broader subclass of $\mathrm{IMC}^{\mathrm{G}}$.

Note that even though we are working with finite labelled transition systems, and therefore computing the exact results for such systems is possible, the algorithms that we have developed in this chapter are mostly computing over-approximations of actual system properties. For example, in the reachability algorithm in Section 5.3.1 we only determine for each separate label whether it is reachable or not and do no take into account the "interplay" between the labels, when some labels are blocked and some other labels have, for example, to be executed several times in order to some other labels to become exposed. The positive side of the loss in precision is the gain in efficiency of computation and in memory usage. For example, we do not need to build the whole labelled tran-

sition system of an initial IMC$^{\mathrm{G}}$ program F but only to compute the Pathway Analysis and several additional operators on F and to post-process the results. For reachability we do the reachability analysis on labels and not on states – remember, that there can be maximal $2^{|Labs(F)|}$ states but only $|Labs(F)|$ labels.

5.1 Characterisation of valid states

We will define two operators – *excl* and *choice* – which will help us to determine which labels exclude the presence of each other (the operator *excl*) and which labels presume the presence of each other. The operator *excl* simply uses the fact that there is a natural order in labels' appearance: for example, in the IMC$^{\mathrm{G}}$ program $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.X}$ the presence of the label ℓ_1 excludes the label ℓ_2 and the other way round. The situation is more complex with the operator *choice*: here we are using the fact that the presence of some labels can indicate that two or more potential subprocesses did not start their execution yet, so all their initial labels are exposed. Consequently we combine the results of *excl* and *choice* in the function *Valid*.

5.1.1 Mutually exclusive labels

We define the operator $excl : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times\mathbf{Lab}}$ in Table 5.1. It returns pairs of labels such that if one of the labels is exposed then the other one is necessarily not exposed (the operator is symmetric). We will prove this fact in the Lemma 5.1. We can subsequently rule out any $M \in \mathfrak{M}$ such that $\ell_1 \in dom(M)$, $\ell_2 \in dom(M)$ and $(\ell_1, \ell_2) \in excl[\![F]\!]$ as not describing exposed labels of any E derivable from an initial IMC$^{\mathrm{G}}$ program F. The rules in Table 5.1 are mostly clear except perhaps rule (5): labels in two choice branches are mutually exclusive with the exception of labels on the highest syntactic level. Thus in the IMC$^{\mathrm{G}}$ program $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.X + \mathsf{c}^{\ell_3}.\mathsf{d}^{\ell_4}.X}$ the labels ℓ_2 and ℓ_4 are mutually exclusive while the labels ℓ_1 and ℓ_3 are not mutually exclusive. We can use an empty environment for computing exposed labels in rule (5) because summands in F are guarded according to the syntax of IMC$^{\mathrm{G}}$ (see Table 3.1).

Lemma 5.1 (*Excl* operator)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F*,* $\mathcal{F}$ *as in Definition 4.9, then for any* $F \xrightarrow{*} E$ *holds:*

$$((\ell_1, \ell_2) \in excl[\![F]\!]) \wedge (\ell_1 \in dom(\mathcal{E}_\Gamma[\![E]\!])) \Rightarrow (\ell_2 \notin dom(\mathcal{E}_\Gamma[\![E]\!])).$$

$$
\begin{array}{rcll}
excl[\![\mathsf{a}^{\ell}.X]\!] & = & \emptyset & (1)\\
excl[\![\lambda^{\ell}.X]\!] & = & \emptyset & (2)\\
excl[\![\mathsf{a}^{\ell}.P]\!] & = & \bigcup_{\ell' \in Labs(P)} \{(\ell,\ell'),(\ell',\ell)\} \cup excl[\![P]\!] & (3)\\
excl[\![\lambda^{\ell}.P]\!] & = & \bigcup_{\ell' \in Labs(P)} \{(\ell,\ell'),(\ell',\ell)\} \cup excl[\![P]\!] & (4)\\
excl[\![P_1 + P_2]\!] & = & (\bigcup_{\ell_1 \in Labs(P_1) \setminus dom(\mathcal{E}_\emptyset[\![P_1]\!])} \bigcup_{\ell_2 \in Labs(P_2) \setminus dom(\mathcal{E}_\emptyset[\![P_2]\!])} & \\
 & & \{(\ell_1,\ell_2),(\ell_2,\ell_1)\}) \cup excl[\![P_1]\!] \cup excl[\![P_2]\!] & (5)\\
excl[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & excl[\![P]\!] & (6)\\
excl[\![P_1 \ {}_{\shortparallel}A_{\shortparallel}\ P_2]\!] & = & excl[\![P_1]\!] \cup excl[\![P_2]\!] & (7)\\
excl[\![\underline{X := P}]\!] & = & excl[\![P]\!] & (8)\\
excl[\![\mathbf{0}]\!] & = & \emptyset & (9)
\end{array}
$$

Table 5.1 – Definition of the operator $excl : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab} \times \mathbf{Lab}}$.

Proof. We prove the lemma by proving the more general statement that it holds $((\ell_1,\ell_2) \in excl[\![F]\!]) \wedge (\ell_1 \in dom(\mathcal{E}_\Gamma[\![E'']\!])) \Rightarrow (\ell_2 \notin dom(\mathcal{E}_\Gamma[\![E'']\!]))$ also for any subexpression E'' of E, i.e. for any $E'' \preceq E$. This will establish the lemma as it obviously holds $E \preceq E$. We make our proof by induction on the number of steps in the derivation $F \xrightarrow{*} E$.

The statement about subexpressions holds for F: we can show by induction on the structure of F using the rules of the operator *excl* in the Table 5.1 that for all $E'' \preceq F$, $\ell_1 \in dom(\mathcal{E}_\Gamma[\![E'']\!])$ and $\ell_2 \in dom(\mathcal{E}_\Gamma[\![E'']\!])$ it holds $(\ell_1,\ell_2) \notin excl[\![F]\!]$ and this is logically equivalent to the statement that we are proving.

We have to prove now that if the statement holds for some E such that $F \xrightarrow{*} E$ then it also holds for E' such that $E \longrightarrow E'$. We prove this by induction on the transition derivation according to the rules in Table 3.5.

The statement is clear for the rules (1) and (10) in Table 3.5 because in these cases $E' \preceq E$. The statement easily follows from the induction hypothesis for the majority of other rules (remember that $Labs(P_1) \cap Labs(P_2) = \emptyset$ holds for $P_1 \ {}_{\shortparallel}A_{\shortparallel}\ P_2 \preceq E$). For the rules (9) and (16) we need however to additionally show that the statement of the lemma holds for $P\{\underline{X := P}/X\}$ if it holds for $\underline{X := P}$ – this is necessary for the induction hypothesis to become applicable.

We can deduce the statement for all subexpressions $P'' \preceq P$ directly from the induction hypothesis for $\underline{X := P}$. For subexpressions of the type $P''\{\underline{X := P}/X\}$ with $P'' \preceq P$ we additionally apply Lemma 3.6. This lemma states that $\mathcal{E}_\Gamma[\![E'']\!] = \mathcal{E}_\Gamma[\![E''\{\underline{X := E'}/X\}]\!]$ holds for all $\mathrm{IMC}^{\mathrm{G}}$ expressions E'' and $\underline{X := E'}$. We can deduce therefore $\mathcal{E}_\Gamma[\![P'']\!] = \mathcal{E}_\Gamma[\![P''\{\underline{X := P/X}\}]\!]$ and in this way deter-

mine that the induction hypothesis for subexpressions of $\underline{X := P}$ is applicable also for $P''\{\underline{X := P}/X\}$ with $P'' \preceq P$. □

5.1.2 Labels that appear together

In the following we will define a second operator – the operator *choice*. It returns on an IMC$^{\mathrm{G}}$ program a set of pairs of sets of labels. This is different from the operator *excl* which we have defined above and which returns pairs of just labels and not sets of labels. This is because we would like to determine the situation where there is a choice not only between individual labels but between sets. For example, for the IMC$^{\mathrm{G}}$ program $\mathsf{a}^{\ell_1}.\mathbf{0} + (\mathsf{b}^{\ell_2}.\mathbf{0} \parallel A \parallel \mathsf{c}^{\ell_3}.\mathbf{0})$ we would obtain $\{(\{\ell_1\}, \{\ell_2, \ell_3\})\}$ as the result of the application of *choice*. Intuitively, we can choose between an execution of the label ℓ_1 on the one hand and the execution of both the labels ℓ_2 and ℓ_3 on the other hand.

See the rules for the operator *choice* in Table 5.2. The most important rule is rule (5). The idea is to "cross" the choice sets of P_1 and P_2 which are exposed (saved accordingly in S_1 and S_2) with each other. In case P_1 or P_2 have some exposed labels which are not yet returned as the choice sets of accordingly P_1 or P_2, then new choice sets are created and added to the set S. The choice sets returned by *choice* on separately P_1 and P_2 are also included. The operator *choice* is defined to be symmetric, therefore for a pair of choice sets L_1 and L_2 it is enough to add (L_1, L_2) to the result of $\mathit{choice}[\![F]\!]$ – this would automatically mean that (L_2, L_1) is in $\mathit{choice}[\![F]\!]$ as well.

We will show in Lemma 5.2 how to use the results of the operator *choice* on an IMC$^{\mathrm{G}}$ program F in order to rule out sets of exposed labels which do not characterise any derivation of F. Intuitively, any set of labels L_1, such that $|L_1| > 1$ (i.e. L_1 contains more than one element), and $(L_1, L_2) \in \mathit{choice}[\![F]\!]$ for some L_2, represents the exposed labels of a synchronisation construct (see rule (5) in Table 5.2). If L_2 contains more than one element as well then it represents exposed labels of another synchronisation construct, otherwise it represents simply an exposed prefix. It is easy to notice that if at least one of the labels in a synchronisation construct has been executed, then only labels from that synchronisation construct can become exposed anew. This is due to the process identifier closeness of synchronisation constructs in well-formed IMC$^{\mathrm{G}}$ expressions, see rule (7) in Table 3.8. Therefore if both labels from L_1 and L_2 are exposed, this means that no label from L_1 has ever been executed, from which we can derive that all the labels in L_1 have to be exposed. The same holds for L_2 if it also contains more than one element, i.e. also represents

$$
\begin{array}{rcll}
choice[\![\mathsf{a}^{\ell}.X]\!] & = & \emptyset & (1)\\
choice[\![\lambda^{\ell}.X]\!] & = & \emptyset & (2)\\
choice[\![\mathsf{a}^{\ell}.P]\!] & = & choice[\![P]\!] & (3)\\
choice[\![\lambda^{\ell}.P]\!] & = & choice[\![P]\!] & (4)\\
choice[\![P_1 + P_2]\!] & = & choice[\![P_1]\!] \cup choice[\![P_2]\!] \cup S \cup & \\
 & & \bigcup_{(L_1,L_2)\in S_1} \bigcup_{(L_1',L_2')\in S_2} & \\
 & & \{(L_1,L_1'),(L_1,L_2'),(L_2,L_1'),(L_2,L_2')\} & \\
 & & \{(L_1',L_1),(L_2',L_1),(L_1',L_2),(L_2',L_2)\} & (5)
\end{array}
$$

$$
\begin{array}{rcl}
\text{where } S_1 & = & \bigcup_{\{(L_1,L_2)\in choice[\![P_1]\!] \mid L_1\cup L_2 \subseteq dom(\mathcal{E}_{\emptyset}[\![P_1]\!])\}} \{(L_1,L_2)\}\\
S_2 & = & \bigcup_{\{(L_1',L_2')\in choice[\![P_2]\!] \mid L_1'\cup L_2' \subseteq dom(\mathcal{E}_{\emptyset}[\![P_2]\!])\}} \{(L_1',L_2')\}\\
S & = & \begin{cases}
\bigcup_{(L_1,L_2)\in S_1} \{(L_1, dom(\mathcal{E}_{\emptyset}[\![P_2]\!])), & \\
(dom(\mathcal{E}_{\emptyset}[\![P_2]\!]), L_1)\} & \\
\quad if\ (S_1 \neq \emptyset) \wedge (S_2 = \emptyset) \wedge (\mathcal{E}_{\emptyset}[\![P_2]\!] \neq \bot_{\mathfrak{M}}) & \\
\bigcup_{(L_1,L_2)\in S_2} \{(L_1, dom(\mathcal{E}_{\emptyset}[\![P_1]\!])), & \\
(dom(\mathcal{E}_{\emptyset}[\![P_1]\!]), L_1)\} & \\
\quad \text{if } (S_2 \neq \emptyset) \wedge (S_1 = \emptyset) \wedge (\mathcal{E}_{\emptyset}[\![P_1]\!] \neq \bot_{\mathfrak{M}}) & \\
\{(dom(\mathcal{E}_{\emptyset}[\![P_1]\!]), dom(\mathcal{E}_{\emptyset}[\![P_2]\!])), & \\
(dom(\mathcal{E}_{\emptyset}[\![P_2]\!]), dom(\mathcal{E}_{\emptyset}[\![P_1]\!]))\} & \\
\quad \text{if } (S_1 = \emptyset) \wedge (S_2 = \emptyset) \wedge & \\
\quad\quad (\mathcal{E}_{\emptyset}[\![P_1]\!] \neq \bot_{\mathfrak{M}}) \wedge (\mathcal{E}_{\emptyset}[\![P_2]\!] \neq \bot_{\mathfrak{M}}) &
\end{cases}
\end{array}
$$

$$
\begin{array}{rcll}
choice[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & choice[\![P]\!] & (6)\\
choice[\![P_1 \ {}_{|\!|}A_{|\!|}\ P_2]\!] & = & choice[\![P_1]\!] \cup choice[\![P_2]\!] & (7)\\
choice[\![X := P]\!] & = & choice[\![P]\!] & (8)\\
choice[\![\mathbf{0}]\!] & = & \emptyset & (9)
\end{array}
$$

Table 5.2 – Definition of the symmetric operator $choice : \mathrm{IMC}^{\mathrm{G}} \to 2^{2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}}}$ given an $\mathrm{IMC}^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9.

exposed labels of a synchronisation construct. Otherwise L_2 represents only one exposed prefix and the statement of the lemma is trivially true for it.

Lemma 5.2 (*Choice* operator)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F*,* $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E$*, then from* $(L_1, L_2) \in choice[\![F]\!]$*,* $\ell_1 \in L_1$*,* $\ell_2 \in L_2$*,* $\ell_1 \in dom(\mathcal{E}_{\Gamma}[\![E]\!])$ *and* $\ell_2 \in dom(\mathcal{E}_{\Gamma}[\![E]\!])$ *follows:*

1. *for all* $\ell \in L_1$ *holds* $\ell \in dom(\mathcal{E}_{\Gamma}[\![E]\!])$*;*

2. for all $\ell \in L_2$ holds $\ell \in dom(\mathcal{E}_\Gamma[\![E]\!])$.

Proof. We can prove the statement of the lemma by induction on the syntax of F. In order to be able to prove the statement of the lemma by induction, we need to prove a more general statement – that the statement of the lemma holds for any $E'' \preceq E$, i.e. that from $\ell_1 \in dom(\mathcal{E}_\Gamma[\![E'']\!])$ and $\ell_2 \in dom(\mathcal{E}_\Gamma[\![E'']\!])$ follows that for any $\ell \in L_1$ or $\ell \in L_2$ such that $(L_1, L_2) \in choice[\![F]\!]$ holds $\ell \in dom(\mathcal{E}_\Gamma[\![E'']\!])$.

We prove the statement by induction on the number of transitions in $F \stackrel{*}{\longrightarrow} E$. We can prove that the statement holds for $F = E$ by induction on F, using the rules for the *choice* operator in the Table 5.2 and the fact that F is uniquely labelled. We will prove now that if $F \stackrel{*}{\longrightarrow} E$, $E \longrightarrow E'$ and the statement holds for E then it also holds for E'. We prove this by induction on the transition derivation using the rules in the Table 3.5.

The statement holds for the rules (1) and (10) because it holds for any $E'' \preceq E$ and we have $E' \preceq E$. Most of the other rules easily follow from the induction hypothesis. Note that the induction hypothesis is applicable for the rules (4)-(6) and (13)-(14) because according to the rules of the *choice* operator in Table 5.2 for each $(L_1, L_2) \in choice[\![E_1 \ {}_{\shortparallel}A_{\shortparallel} \ E_2]\!]$ holds either $(L_1, L_2) \in choice[\![E_1]\!]$ or $(L_1, L_2) \in choice[\![E_2]\!]$.

It is therefore left to show that if the statement of the lemma holds for any $E'' \preceq \underline{X := E}$ then it also holds for any $E'' \preceq E\{\underline{X := E}/X\}$. It is obviously true if $E'' \preceq E$. Otherwise $E'' = E'''\{\underline{X := E}/X\}$ with some $E''' \preceq E$ and the statement of the lemma follows from Lemma 3.6. According to it, $\mathcal{E}_\Gamma[\![E'''\{\underline{X := E}/X\}]\!] = \mathcal{E}_\Gamma[\![E''']\!]$ and we can derive the statement from the induction hypothesis for E. □

We can explain the above lemma on an example. Take as an example the process $\underline{X := \mathsf{a}^{\ell_1}.X + Y := \mathsf{b}^{\ell_2}.Y} \ {}_{\shortparallel}A_{\shortparallel} \ \underline{Z := \mathsf{c}^{\ell_3}.Z}$ that can execute the action a and return to the initial configuration with the exposed labels ℓ_1, ℓ_2 and ℓ_3. Clearly if both ℓ_1 and ℓ_2 are exposed then also ℓ_3 is exposed, also if both ℓ_1 and ℓ_3 are exposed then ℓ_2 is exposed. This is also what Lemma 5.2 predicts as the *choice* operator returns the symmetric closure of $\{(\{\ell_1\}, \{\ell_2, \ell_3\})\}$.

However we will "lose" a part of the behaviour after choosing the second choice alternative, i.e. either action b or action c. Therefore even if both ℓ_2 and ℓ_3 are

exposed, it does not mean that ℓ_1 is exposed – and this corresponds to what Lemma 5.2 is predicting. We cannot "redo" the choice of the second alternative process later on – ℓ_1 will never become exposed again.

We saw in the above example that we are mostly interested in "choice sets" which contain more than one label and are returned by the operator *choice* in one of the pairs. The operator *choice* returns however also pairs with both sets containing only one element. There are two reasons for this. First, it would be hard to keep track of choice sets if we would only return those containing more than one element on subexpressions: for example, we would like $\mathit{choice}[\![\underline{X := \mathsf{a}^{\ell_1}.X + Y := \mathsf{b}^{\ell_2}.Y + (Z := \mathsf{c}^{\ell_3}.Z \between A \between \mathsf{d}^{\ell_4}.\mathbf{0})}]\!]$ to be the symmetric closure of $\{(\{\ell_1\},\{\ell_3,\ell_4\}),(\{\ell_2\},\{\ell_3,\ell_4\})\}$, but in this case we would need to record in some way that $\{\ell_1\}$ and $\{\ell_2\}$ are alternative choice sets. Second, we could use the results of *choice* for deciding some additional properties in the future, where we will need to know which labels are in a "choice relation" with each other.

The next Lemma 5.3 is easily explainable: it states namely that there cannot be more choice alternatives in the derivative expressions than the ones already present in the initial IMC$^{\mathrm{G}}$ expression. We can however have a strict inclusion as we have seen in the example $\underline{X := \mathsf{a}^{\ell_1}.X + Y := \mathsf{b}^{\ell_2}.Y} \between A \between \underline{Z := \mathsf{c}^{\ell_3}.Z} \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}}$ $\underline{Y := \mathsf{b}^{\ell_2}.Y} \between A \between \underline{Z := \mathsf{c}^{\ell_3}.Z}$.

Lemma 5.3 (*Choice* operator: results inclusion)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F*,* $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E$*, then* $\mathit{choice}[\![E]\!] \subseteq \mathit{choice}[\![F]\!]$ *holds.*

Proof. It is enough to prove that from $E \longrightarrow E'$ follows $\mathit{choice}[\![E']\!] \subseteq \mathit{choice}[\![E]\!]$ for any IMC$^{\mathrm{G}}$ expressions E and E'. We prove this by induction on the transition derivation.

This clearly follows for the base cases – the rules (1) and (10) in the Table 3.5 – from the rules of the *choice* operator in the Table 5.2. The statement follows from the induction hypothesis for the rest of the rules from the Table 3.5 besides the rules (9) and (16). In order to use the induction hypothesis also for them we need to show that $\mathit{choice}[\![\underline{X := E}]\!] \supseteq \mathit{choice}[\![E\{\underline{X := E}/X\}]\!]$.

We prove this by showing that $\mathit{choice}[\![\underline{X := E}]\!] \supseteq \mathit{choice}[\![E''\{\underline{X := E}/X\}]\!]$ for any $E'' \preceq E$. We can prove the latter by induction on the structure of E''. This is clear for the base syntax rules – (1)-(4) and (9) in Table 3.1. For rule (5) we can use the Lemma 3.6 from which follows that $\mathcal{E}_\Gamma[\![P_1]\!] = \mathcal{E}_\Gamma[\![P_1\{\underline{X := E}/X\}]\!]$

$$Valid(M) = \begin{cases} false & \text{if } dom(M) \not\subseteq Labs(F) \\ false & \text{if } \exists \ell_1, \ell_2\ ((\ell_1, \ell_2) \in excl[\![F]\!]) \wedge (\{\ell_1, \ell_2\} \subseteq dom(M)) \\ false & \text{if } \exists \ell_1, \ell_2, \ell, L_1, L_2\ ((L_1, L_2) \in choice[\![F]\!]) \wedge (\ell_1 \in L_1) \\ & \wedge (\ell_2 \in L_2) \wedge (\ell \in L_1 \cup L_2) \wedge (\{\ell_1, \ell_2\} \subseteq dom(M)) \\ & \wedge (\ell \notin dom(M)) \\ true & \text{otherwise.} \end{cases}$$

Table 5.3 – Definition of the function $Valid : \mathfrak{M} \to \{false, true\}$, given an IMC$^{\mathrm{G}}$ program F and the results of $choice[\![F]\!]$ and $excl[\![F]\!]$.

and $\mathcal{E}_\Gamma[\![P_2]\!] = \mathcal{E}_\Gamma[\![P_2\{\underline{X := E}/X\}]\!]$ for any IMC$^{\mathrm{G}}$ expressions P_1 and P_2. For the rest of the syntactic rules we can apply the induction hypothesis. □

5.1.3 Determining valid states

We can use now the results of the operators **excl** and **choice** on some IMC$^{\mathrm{G}}$ program F in order to predict which derivative states are possible and which are not. This would be useful in constructing model-checking algorithms – we could exclude some impossible states and in this way decrease the complexity of the algorithms. In further developments (for example, in Table 5.5) we will use the function *Valid* which will return for a label multiset a boolean value signalling whether it is a possible multiset according to the functions **excl** and **choice** or not.

We will assess now the complexity of computing the function *Valid* for a fixed IMC$^{\mathrm{G}}$ program F and the complexity of applying it to any multiset M with $dom(M) \subseteq Labs(F)$. The complexity of computing *Valid* is determined by the complexities of computing the results of applying the operators *excl* and *choice* to F. These complexities are expressed as functions either of the size of the syntax of F (counting all the operators, action names and delay rates that appear in the syntax) or of the number of labels in F, i.e. of $|Labs(F)|$. For uniquely labelled IMC$^{\mathrm{G}}$ expressions there is however not a big difference between the two, as the number of labels occurring in them and the size of their syntactic description differ only by a linear parameter.

As we can see from Table 5.1, the time complexity of the computation of $excl[\![F]\!]$ is quadratic in the syntax of F and the space complexity is quadratic in the

number of labels in F: we would need a matrix of the size $|Labs(F)|^2$ for saving the computed exclusion relation. The application to an arbitrary multiset M with $dom(M) \subseteq Labs(F)$ has also the quadratic time complexity in the number of labels in F: for each label in $dom(M)$ we should check all the labels it excludes for their presence in $dom(M)$.

The computation of $choice[\![F]\!]$ can be implemented with the time complexity linear in the size of the syntax of F and the space complexity linear in the number of labels in F: for each biggest sum in the syntactic description of F we could have a separate partition and for each summand inside the sum we would have its exposed labels forming a "subpartition". The application of the results of $choice[\![F]\!]$ to some multiset M with $dom(M) \subseteq Labs(F)$ can be implemented as linear in the number of labels in the syntax of F as well: we should check whether there exist two subpartitions in one partition that both contain labels from $dom(M)$ but are not fully contained in $dom(M)$.

5.2 Computing Zeno states

In this section we will demonstrate how the usual model-checking algorithms for labelled transition systems can transferred to our setting of the Pathway Analysis of IMC$^{\mathrm{G}}$ systems. The idea is that we can first perform the Pathway Analysis on an IMC$^{\mathrm{G}}$ program computing the tuple $\mathcal{F}$ as in Definition 4.9, compute the function *Valid* as in Table 5.3 and then based on the gathered information do the model-checking of formulas from, for example, CTL [CES86] or ACTL [NV90] logics.

Even though the algorithms that we intend to use are well-known (see, for example, [BK08]), the benefit of our method is that we do not need to build a labelled transition system induced by the semantics of an IMC$^{\mathrm{G}}$ program before the actual model-checking procedure. In the standard setting a labelled transition system is constructed for an IMC$^{\mathrm{G}}$ program by the application of the Structural Operational Semantics rules (some abstraction can take place at this stage though) before the actual model checking can take place. We can on the other hand do the computations on the "valid" states (according to the function *Valid* defined in Table 5.3) – this may lead to a lower space complexity as we have a more compact way of representing the semantics of the system by saving only valid states and not the transitions between them.

In case some valid states will be found that satisfy the logical formula it would be necessary to check whether the found states are in fact reachable from the initial state. Also in this case we could use the methods for computing an over-

approximation of reachable labels (see Section 5.3) and in this way avoid the building of the whole labelled transition system. There are also other potential benefits of our method due to the fact that states have an internal structure which is known to us – we can, for example, ignore some of the labels that are not important for us, thus merging together a number of states that only differ in their "unimportant" labels. This could considerably reduce the state space.

We will discuss the CTL/ACTL model checking of IMC^{G} on the example of computing a set of so-called *Zeno* states. As usual, we start with some initial IMC^{G} program and understand under "states" all the IMC^{G} expressions that can be derived from the initial program by the application of the Structural Operational Semantics rules. States are Zeno if they are in the coarsest relation such that they can make an internal move, i.e. a transition with the τ-action, which will lead to a Zeno state as well (see Definition 5.4). For any Zeno state there exists therefore an infinite path starting from it with only internal actions. This behaviour can be easily captured by a formula in the ACTL logic. Computing Zeno states can also be regarded as a example of computing sets of states with properties that can be formulated co-inductively, i.e. of states that exhibit some behaviour and can do a transition into states having the same properties.

Definition 5.4 (Zeno states)**.** *An* IMC^{G} *expression is called* Zeno *if it is contained in the coarsest relation* R *on* IMC^{G} *such that for each* $E \in R$ *there exists a semantic transition* $E \xrightarrow{\tau} E'$ *to an* IMC^{G} *expression* $E' \in R$*.*

Definition 5.4 concerns IMC^{G} expressions but it can be easily converted into the definition for the multisets of labels. The property of states characterised by multisets of labels to be Zeno can be expressed as the greatest fixed point of an order-preserving endofunction. See Table 5.4 for the definition of the endofunction *FZeno* which operates on sets of label multisets and returns a set of valid multisets such that each label multiset in it has a derivative in the input. The derivation should be through a τ-chain. Note that we are identifying IMC^{G} expressions in Definition 5.4 with their exposed labels in Table 5.3. It is clear that we need an initial IMC^{G} program F to be fixed, its Pathway Analysis to be conducted and all the functions from the tuple $\mathcal{F}$ as in Definition 4.9 to be computed before defining the function *FZeno*.

In Lemma 5.5 we will prove that multisets of exposed labels of Zeno IMC^{G} expressions indeed constitute the greatest fixed point (GFP) of the function *FZeno* defined above. In this way we will prove that we got a transition from IMC^{G} expressions to multisets "right" in our definition of the function *FZeno*.

$$\begin{aligned} FZeno(S) \quad = \quad & \{M \in \mathfrak{M} | (\mathit{Valid}(M)) \wedge (\exists C \in \mathfrak{T}\ (\mathit{Name}^h_{ch}(C) = \tau) \wedge \\ & (C \leq M) \wedge (M - K_{ch}(C) + G_{ch}(C) \in S)\} \end{aligned}$$

Table 5.4 – Definition of the endofunction $FZeno : 2^{\mathfrak{M}} \to 2^{\mathfrak{M}}$, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9.

Lemma 5.5 (Zeno processes)**.** *Given an* IMC$^{\mathrm{G}}$ *program F, $\mathcal{F}$ as in Definition 4.9 and $F \xrightarrow{*} E$, then E is Zeno iff $\mathcal{E}_\Gamma[\![E]\!]$ is contained in the greatest fixed point of FZeno.*

Proof. First of all, there exists the greatest fixed point of the function *FZeno*: we can regard $2^{\mathfrak{M}}$ as a complete lattice with $\subseteq$ as a partial order (set inclusion relation). In this case *FZeno* is an order preserving function: it is easy to see that from $S_1 \subseteq S_2$ follows $FZeno(S_1) \subseteq FZeno(S_2)$. Knaster-Tarski theorem is therefore applicable and the function *FZeno* has its greatest fixed point on $2^{\mathfrak{M}}$.

We will prove now the statement of the lemma. If E is not Zeno then each sequence of states starting from it has the following form: $E_1 \xrightarrow{\tau} E_2 ... E_{n-1} \xrightarrow{\tau} E_n$ and $E_n \xrightarrow{\not\tau}$, with $E = E_1$ and $n \geq 1$. Let *fp* be a fixed point of the function *FZeno*. Then $\mathcal{E}_\Gamma[\![E]\!] \notin fp$ due to the following reasoning: assume $\mathcal{E}_\Gamma[\![E]\!] \in fp$. Then there exists a transition such that $E \longrightarrow E_2$ and $\mathcal{E}_\Gamma[\![E_2]\!] \in fp$, from the latter would follow the existence of the transition $E_2 \longrightarrow E_3$ such that $\mathcal{E}_\Gamma[\![E_3]\!] \in fp$ and so on until $\mathcal{E}_\Gamma[\![E_n]\!] \in fp$. The last however is impossible because $\mathcal{E}_\Gamma[\![E_n]\!]$ is not in a co-domain of *FZeno*: there is no chain C representing an internal action such that $C \leq E_n$ holds.

If on the other hand the state E is Zeno then there exists a transition $E \xrightarrow[C_1]{\tau} E_2$ such that the IMC$^{\mathrm{G}}$ expression E_2 is Zeno as well. Further there exists a transition $E_2 \xrightarrow[C_2]{\tau} E_3$ such that the IMC$^{\mathrm{G}}$ expression E_3 is Zeno and so on. Therefore, with $E_1 = E$, holds $\bigcup_{i=1..\infty} \mathcal{E}_\Gamma[\![E_i]\!] = FZeno(\bigcup_{i=1..\infty} \mathcal{E}_\Gamma[\![E_i]\!])$ (all $\mathcal{E}_\Gamma[\![E_i]\!]$s are "valid", there exists an exposed chain C_i representing an internal action such that $\mathcal{E}_\Gamma[\![E_i]\!] - K_{ch}(C_i) + G_{ch}(C_i) = \mathcal{E}_\Gamma[\![E_{i+1}]\!] \in \bigcup_{i=1..\infty} \mathcal{E}_\Gamma[\![E_i]\!]$), i.e. $\bigcup_{i=1..\infty} \mathcal{E}_\Gamma[\![E_i]\!]$ is a fixed point of *FZeno*. Note that we have in fact a finite union because the domain and co-domain of *FZeno* are finite.

We can make a conclusion that $\bigcup_{i=1..\infty} \mathcal{E}_\Gamma[\![E_i]\!] \subseteq GFP(FZeno)$ (*GFP* denotes the greatest fixed point), as any other fixed point of *FZeno* is smaller and there-

$$parents_{\mathfrak{M}}(M) \triangleq \{M' \in \mathfrak{M} | \exists C \in \mathfrak{T}\ (Valid(M')) \wedge (G_{ch}(C) \leq M) \wedge (M - G_{ch}(C) + C \leq M' \leq M - G_{ch}(C) + K_{ch}(C))\}$$

$$S := \{M \in \mathfrak{M} | \exists C \in \mathfrak{T}\ (Valid(M)) \wedge (Name^{h}_{ch}(C) = \tau) \wedge (C \leq M)\};$$

$$S_{nz} := \{M \in S | \nexists C \in \mathfrak{T}\ (Name^{h}_{ch}(C) = \tau) \wedge (C \leq M) \wedge (M - K_{ch}(C) + G_{ch}(C) \in S)\};$$

while $S_{nz} \neq \emptyset$ *do*

$$S := S \backslash S_{nz};$$

$$S_{nz} := \{M' \in S | (M' \in \bigcup_{M \in S_{nz}} parents_{\mathfrak{M}}(M)) \wedge (\nexists C \in \mathfrak{T}\ (Name^{h}_{ch}(C) = \tau) \wedge (C \leq M') \wedge (M' - K_{ch}(C) + G_{ch}(C) \in S))\};$$

Table 5.5 – Definition of the algorithm computing $S = GFP(FZeno)$, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9.

fore included in the $GFP(FZeno)$, which also means that $\mathcal{E}_{\Gamma}[\![E]\!] \in GFP(FZeno)$. We have thus proved that the greatest fixed point of $FZeno$ contains all multisets corresponding to Zeno states and no other multisets. □

We will present now an algorithm for computing the greatest fixed point of the function $FZeno$. As already mentioned, the algorithm itself is not new – it is essentially a usual model-checking algorithm for computing a CTL formula $\exists\Box\Phi$ for a CTL-state formula Φ, see, for example, [BK08]. With our algorithm in Table 5.5 we demonstrate besides the applicability of model-checking algorithms in our setting also the use of the function $parents_{\mathfrak{M}}$ which returns for a multiset of labels all the valid multisets that could have generated the input multiset in one step.

We start with computing a set S containing all "valid" states that additionally have an enabled chain corresponding to the τ-action. Consequently the multisets in S will be examined one by one in order to find those that do not have a τ-transition into the set S – these are saved in the set S_{nz} (non-Zeno) and deleted from S. If S_{nz} is not empty then the "parents" of label multisets in S_{nz} are examined and those of the parents that do not have any enabled τ-chain leading to S are saved in the new S_{nz} which is deleted from S in its turn. This examining of "parents" continues as long as S_{nz} is not empty. In Lemma 5.6 we will prove that the just described algorithm (put more formally in Table 5.5) computes exactly $GFP(FZeno)$ in S.

Lemma 5.6 (Computing $GFP(FZeno)$)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *and* $\mathcal{F}$ *as in Definition 4.9, the above algorithm terminates computing* $S = GFP(FZeno)$.

Proof. First of all, we can easily see that for any $F \xrightarrow{*} E \xrightarrow[C]{} E'$ and $M = \mathcal{E}_\Gamma[\![E']\!]$ holds $\mathcal{E}_\Gamma[\![E]\!] \in parents_{\mathfrak{M}}(M)$: this is true because it holds that $\mathcal{E}_\Gamma[\![E']\!] = \mathcal{E}_\Gamma[\![E]\!] - K_{ch}(C) + G_{ch}(C)$ and due to the fact that $K_{ch}(C) \geq C$ for any $C \in \mathfrak{T}$ (the latter can be easily proved by induction on the structure of F). Moreover, $\{M \in \mathfrak{M} | \exists C \in \mathfrak{T}\ (Valid(M)) \wedge (Name^h_{ch}(C) = \tau) \wedge (C \leq M)\}$ contains $GFP(FZeno)$, because it clearly holds that any Zeno state can do a τ-transition.

On the other hand, any $M \triangleq \mathcal{E}_\Gamma[\![E]\!]$ such that $M \notin GFP(FZeno)$ will be deleted from S in one of the loop runs because for any such M holds that any path starting from E reaches a state E' such that $E' \overset{\tau}{\not\rightarrow}$. Assume that n is the length of the longest path such that $E \xrightarrow{*} E'$, where all the transitions are internal, and $E' \overset{\tau}{\not\rightarrow}$. We can prove that $M \in S_{nz}$ in the n loop run – we can prove this by induction on n. The base case is $n = 0$, i.e. $E \overset{\tau}{\not\rightarrow}$ and M will be deleted from S without any loop run. Otherwise we have $E \xrightarrow{\tau} E'' \xrightarrow{*} E'$. As E'' has been deleted from S in the $n-1$ loop run according to the induction hypothesis, it's "parent" E will be checked. It does not have however any other derivative in S after executing a τ-chain (because all the other paths are shorter and therefore all derivatives have been deleted from S before), therefore E will be deleted from S in the n loop run.

Algorithm terminates, because S is finite and at least one element of S is deleted in each loop run. □

Some state properties that do not depend on the properties of the following states (in contrast to Zeno) can be checked even more easily – for example, the existence of deadlock states or of the states that have only transitions decorated by external actions. In the latter case the execution of the external actions can be blocked by the environment and the states will become deadlock states, therefore the computation of such states can be of interest. For computing deadlocks we could construct a set of multisets which are both valid according to the function *Valid* and do not have any enabled chain. If the computed set is empty then there are no deadlock states, otherwise we would need to check whether the computed potential deadlock states are actually reachable from the initial state. For computing states blockable by the environment we would

compute a set of valid multisets such that all the chains enabled in the multisets correspond to external action names.

In general the transfer of usual model-checking methods to the Pathway Analysis setting is quite straightforward, does not require the building of a labelled transition system representing the semantics of an IMC^{G} program beforehand (but possibly at least a part thereof needs to be constructed afterwards, for conducting reachability analysis for found states). We conjecture that the considerable improvements are possible if we group states together with similar properties before conducting the model checking – the similarities of the states can be determined by comparing their corresponding exposed labels.

5.3 Over-approximation of reachability

In the previous sections we have often referred to the reachability problem which consists in determining whether one state is reachable from another state in some labelled transition system. Reachability is often regarded as *the* problem in model checking, as it is the basic problem in the model-checking algorithms. In this section we will discuss reachability of labels instead of reachability of states: we assume that we know that some labels are/could be exposed and we would like to assess which other labels could also become exposed. We compute an over-approximation of reachable labels: all the labels that we assess to be reachable are in fact reachable but we might asses as reachable some labels that can in fact never become exposed. Consequently we discuss the question of how to recompute our assessment after one transition and also how to determine whether labels could be generated/executed only finite number of times or unlimited number of times.

We can further combine the computation of reachable labels with the function *Valid* from Table 5.3 in order to filter out not only invalid states according to the function *Valid* but also states that have at least one label exposed that is unreachable according to our assessment.

5.3.1 Computing reachable and executable labels

The schematic presentation of our algorithm is given in Figure 5.2. The basic idea is quite simple: we start with a set of reachable labels equal to the set of exposed labels in some initial state (this is a parameter M of the function *reach* in Table 5.6), check which chains have all their labels in the set of reachable

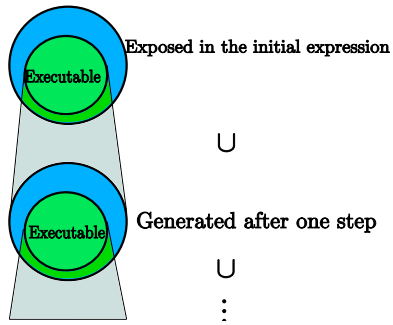

Figure 5.2 – Illustration for the computation of labels' reachability.

$$\begin{array}{rcl} reach_M(L_1, L_2) & = & (dom(M) \cup L_1 \cup L_1', L_2 \cup L_2') \\ L_1' & := & \bigcup_{\ell \in L_2} dom(G(\ell)) \\ L_2' & := & \bigcup_{\{C \in \mathfrak{T} \mid dom(C) \subseteq L_1\}} dom(C) \end{array}$$

Table 5.6 – Definition of the function $reach_M : 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \to 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}}$ given an IMC$^{\mathrm{G}}$ program F, $\mathcal{F}$ as in Definition 4.9, $M \in \mathfrak{M}$.

labels, add labels generated by those chains to the set of reachable labels and so on until the saturation. The algorithm presumes the foregoing Pathway Analysis of an initial IMC$^{\mathrm{G}}$ program F. It is clear why the algorithm will terminate: the set of occurring labels is finite for any IMC$^{\mathrm{G}}$ program.

The function $reach_M$ is defined in Table 5.6. The function is parameterised on a label multiset M that will usually represent the exposed labels of an initial IMC$^{\mathrm{G}}$ program F or its derivative. The idea is that first of all we consider all exposed labels in M as reachable (this is what $dom(M)$ refers to) and moreover those labels are considered reachable for which there exist labels which generate them and are executable. Executable are considered those labels for which there exists a chain such that all labels participating in the chain are reachable. Two input parameters of $reach_M$ keep track of labels which are currently considered to be accordingly reachable and executable and two output parameters return the recomputed sets of reachable and executable labels.

Let us denote $(REACH_M, EXE_M) \triangleq LFP(reach_M)$, with LFP standing for the *least fixed point*. The least fixed point exists because $reach_M$ is an order-preserving function on the complete and finite lattice $2^{Labs(F)} \times 2^{Labs(F)}$ for a fixed IMC$^{\mathrm{G}}$ program F (for example, consider the order $(A, B) \leq (C, D)$ iff $A \leq B$ and $C \leq D$). We will prove the facts about $REACH_M$ and EXE_M in

the next lemma.

Lemma 5.7 ($REACH_M$ and EXE_M fixed points)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F, $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E$, $M = \mathcal{E}_\Gamma[\![E]\!]$, *then from* $E \xrightarrow{*} E'$ *and* $\ell \in dom(\mathcal{E}_\Gamma[\![E']\!])$ *follows* $\ell \in REACH_M$*; from* $E' \xrightarrow[C]{\alpha} E'''$ *for some* $\alpha \in \boldsymbol{Act} \cup \{\tau\} \cup \boldsymbol{Rate}$ *and* $\ell' \in dom(C)$ *follows* $\ell' \in EXE_M$.

Proof. We prove this by induction on the number of derivation steps in $E \xrightarrow{*} E'$. We will namely show that if $E \xrightarrow{\mathsf{n}} E'$ for some $n \in \mathbb{N}_0$ then it holds that $\ell \in [\bigcup_{i=1..2n-1}(reach^i_M((\emptyset,\emptyset)))]^1$ and $\ell' \in [\bigcup_{i=1..2n}(reach^i_M((\emptyset,\emptyset)))]^2$. This will prove the statement of the lemma because the following equality is true: $(REACH_M, EXE_M) = \bigcup_{i=0..\infty} reach^i_M((\emptyset,\emptyset))$ (it is true because $(\emptyset,\emptyset)$ is the least element of the finite complete lattice $2^{Labs(F)} \times 2^{Labs(F)}$).

The base case is $E = E'$ and the statement holds because on the one hand $\ell \in [reach^1_M((\emptyset,\emptyset))]^1 = M$ follows from $M = \mathcal{E}_\Gamma[\![E]\!]$, on the other hand $\ell \in [reach^2_M((\emptyset,\emptyset))]^2$ is true because according to the rules for the $reach_M$ operator $[reach^2_M((\emptyset,\emptyset))]^2 = \bigcup_{\{C \in \mathfrak{T}_\Lambda[\![F]\!] | dom(C) \subseteq M\}} dom(C)$ and these are exactly labels from the executable chains – this follows from the proof of the correctness of the Pathway Analysis.

Assume now that the statement of the lemma is true for some $n \geq 1$, that means for some $E \xrightarrow{*} E''$, and we need to show that it is also true for E' such that $E'' \xrightarrow[C]{\alpha} E'$ for some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$. Any exposed label ℓ of E' is either also an exposed label of E'' and therefore already in $[\bigcup_{i=1..2n-1}(reach^i_M((\emptyset,\emptyset)))]^1$ or has been generated after the execution of the chain C. As any label in C is by induction hypothesis in $[\bigcup_{i=1..2n}(reach^i_M((\emptyset,\emptyset)))]^2$, we can deduce that $\ell \in [\bigcup_{i=1..2(n+1)-1}(reach^i_M((\emptyset,\emptyset)))]^1$ holds. On the other hand, if $E' \xrightarrow{\mathsf{C}'}$ then as we have already shown $\ell \in [\bigcup_{i=1..2(n+1)-1}(reach^i_M((\emptyset,\emptyset)))]^1$ holds for any $\ell \in dom(C')$, therefore $\ell \in [\bigcup_{i=1..2(n+1)}(reach^i_M((\emptyset,\emptyset)))]^2$ follows from the rules of the function $reach_M$. □

The algorithm for computing the least fixed point of $reach_M$ can be implemented with the time and space complexities of $O(|Labs(F)| * |\mathfrak{T}|)$: for each label from $Labs(F)$ added to the set of reachable labels we should check all the chains in which it participates in order to add further reachable labels that the chains

generate. The number of added labels can be maximal $|Labs(F)|$ and the number of chains in which a label participates is at most $|\mathfrak{T}|$.

5.3.2 Updating reachability

In this section we will show how we can recompute the fixed points $REACH_M$ and EXE_M, for $M = \mathcal{E}_\Gamma[\![E]\!]$ such that $F \xrightarrow{*} E$ for some IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9, after one semantic step $E \longrightarrow E'$, i.e. how to compute $(REACH_{M'}, EXE_{M'}) = LFP(reach_{M'})$ for $M' = \mathcal{E}_\Gamma[\![E']\!]$ without recomputing them "from scratch". We define the function $unreach_{R,EXE,K_{set},M}$ for this purpose. It is namely the case that the sets of reachable and executable labels can only become smaller after one semantic step, because some choices may lead to "less behaviour". We will assess this "loss" without reassessing those reachable and executable labels which have not been influenced by the transition.

Computing over-approximations of reachable/executable labels in Section 5.3.1 and updating the sets of reachable/executable labels after one semantic step in this section have some similarities to accordingly widening and narrowing operators in the Abstract Interpretation (see, for example, [NNH99]). Similar to widening, our function *reach* computes upper approximations of the least fixed points of transfer functions which operate on configurations of reachable/executable labels and return the updated configurations of accordingly reachable/executable labels after possible semantic transitions from these configurations. Similar to narrowing, we are improving the approximations in the function *unreach* by utilising the information on the semantic step that has actually been taken. Like in the narrowing procedure, there is a descending chain created by the iterative application of *unreach* which eventually stabilises.

The function $unreach_{R,EXE,K_{set},M}$ is defined in Table 5.7. Assume that we are given an IMC$^{\mathrm{G}}$ program F, $\mathcal{F}$ as in Definition 4.9, $F \xrightarrow{*} E$, and we have already computed $LFP(reach_{M'})$ for $M' = \mathcal{E}_\Gamma[\![E]\!]$. Then in case $E \xrightarrow[C]{} E'$ for some $C \in \mathfrak{T}$ we will be able to compute with the function *unreach* an over-approximation of $LFP(reach_M)$ for $M = \mathcal{E}_\Gamma[\![E']\!]$ which is in general more precise than $LFP(reach_{M'})$ (it is clear from Lemma 5.7 that $LFP(reach_{M'})$ is an over-approximation of $LFP(reach_M)$).

The idea is to initialise the parameters R and EXE of *unreach* to $REACH_{M'}$ and $EXE = EXE_{M'}$ ($(REACH_{M'}, EXE_{M'}) = LFP(reach_{M'})$ as in Section 5.3.1), K_{set} and M to accordingly $dom(K_{ch}(C))$ and $\mathcal{E}_\Gamma[\![E']\!]$. The sets R and EXE are sets of accordingly reachable and executable labels that might get smaller. The set

$$
\begin{array}{l}
unreach_{R,EXE,K_{set},M}(L_1, L_2, L_3) = (R, EXE, K_{set}) \\
\quad if\ (L_1 \supset R) \vee (L_2 \supset EXE) \\
unreach_{R,EXE,K_{set},M}(L_1, L_2, L_3) = (L_1 - K', L_2 - K'', K''') \\
\quad if\ (L_1 \subseteq R) \wedge (L_2 \subseteq EXE)
\end{array}
$$

$$
\begin{array}{l}
with\ K' := \{\ell \in L_3 | (parents(\ell) \cap L_2 = \emptyset) \wedge (\ell \notin dom(M))\} \\
\qquad K'' := \{\ell \in L_1 | \nexists C \in \mathfrak{T}\ (\ell \in dom(C)) \wedge (dom(C) \subseteq L_1 - K')\} \\
\qquad K''' := \bigcup_{\ell \in K''} dom(G(\ell))
\end{array}
$$

Table 5.7 – Definition of the function $unreach_{R,EXE,K_{set},M} : 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \rightarrow 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}}$ given an IMC$^{\mathrm{G}}$ program F, $\mathcal{F}$ as in Definition 4.9, $R \in 2^{\mathbf{Lab}}$, $EXE \in 2^{\mathbf{Lab}}$, $K_{set} \in 2^{\mathbf{Lab}}$, $M \in \mathfrak{M}$.

K_{set} includes labels which need to be examined. The labels from the domain of the multiset M are undoubtedly reachable.

The computation of *unreach* proceeds in the following way: when we first start with an input tuple from the complete lattice $2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}}$ in which either the first element is bigger than (i.e. contains) the R parameter of the function *unreach* or the second element is bigger than the EXE parameter then we just return the first three parameters of *unreach*. This is in a sense an initialisation phase where we start with R and EXE as possibly reachable/executable labels; the possibly unreachable labels that need to be examined further are passed through the third argument. Otherwise we are doing the following: from the set of reachable labels (the first argument of *unreach*) we delete those labels which are not exposed in E' and none of whose "parents" (see Table 4.1 for the definition of the function *parents*) are executable; from the set of executable labels (the second argument of *unreach*) we delete those labels for which there is no chain such that all the labels constituting the chain are reachable; labels which are generated by those labels that have been determined to be "non-executable" need to be examined further in the next step and are returned therefore as the third element of the output of *unreach*.

The function *unreach* is monotone in the first and the second elements: they can obviously only become smaller in the output. The function *unreach* is not necessary monotone in the third element but it will eventually stabilise because it is equal to the labels generated by reachable but non-executable labels and the reachable labels (i.e. the first argument of *unreach*) will eventually stabilise. We will prove in Lemma 5.8 that under this setting the first element of $GFP(unreach_{R,EXE,K_{set},M})$ will be greater or equal equal to the first element of $LFP(reach_M)$ and the second element of $GFP(unreach_{R,EXE,K_{set},M})$ will be greater or equal equal to the second element of $LFP(reach_M)$, assuming a fixed

IMC^{G} program F and $\mathcal{F}$ as in Definition 4.9. Informally speaking, we do not "delete too much", i.e. we still get a safe over-approximation of reachable and executable labels. We conjecture that it is also the case that we delete all the labels that are not contained in the least fixed point of $reach_M$ but proving this would be a bit more involved.

Lemma 5.8 (*UNREACH* fixed point)**.** *Given an* IMC^{G} *program* F*,* $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E \underset{C'}{\longrightarrow} E''$*,* $M' = \mathcal{E}_\Gamma[\![E]\!]$*,* $R = REACH_{M'}$*,* $EXE = EXE_{M'}$*,* $K_{set} := dom(K_{ch}(C') - C')$ *and* $M = \mathcal{E}_\Gamma[\![E'']\!]$*, then in case also* $E'' \xrightarrow{*} E'$ *and* $\ell \in dom(\mathcal{E}_\Gamma[\![E']\!])$*, it follows* $\ell \in [GFP(unreach_{R,EXE,K_{set},M})]^1$*; from* $E' \underset{C}{\longrightarrow} E''''$ *with* $\ell' \in dom(C)$ *follows* $\ell' \in [GFP(unreach_{R,EXE,K_{set},M})]^2$*.*

Proof. We will prove the statements by showing that on any stage of the algorithm execution holds $\ell \notin K'$ and $\ell' \notin K''$. This will be enough as from the Lemma 5.7 follows $\ell \in REACH_M$ and $\ell' \in EXE_M$.

We will prove the lemma by induction on the number of steps in the transition sequence $E'' \xrightarrow{*} E'$. If $E'' = E'$ then the statement is clear: from the rules for K' follows that $\ell \notin K'$ always holds. We can deduce the fact that $\ell' \notin K''$ always holds from the fact that $\ell \notin K'$ always holds and there exists a chain $C \leq \mathcal{E}_\Gamma[\![E'']\!]$ such that no $\ell'' \in dom(C)$ is in K' on any stage (as they are exposed labels of E''). Therefore no label from C will be deleted from EXE and $\ell' \notin K''$ always holds.

We need now to prove the induction step. Assume that $E \longrightarrow E''$, $E'' \xrightarrow{*} E'''$, $E''' \longrightarrow E'$, the statement has been proved for E''' and we need to prove it now for E'. For any $\ell \in dom(\mathcal{E}_\Gamma[\![E']\!])$ follows that either $\ell \in dom(\mathcal{E}_\Gamma[\![E''']\!])$ or there exists a label $\ell'' \in dom(\mathcal{E}_\Gamma[\![E''']\!])$ such that $\ell \in dom(G(\ell''))$. In the first case $\ell \notin K'$ follows from the induction hypothesis. In the second case $\ell'' \notin K''$ follows from the induction hypothesis and therefore $\ell \notin K'$ holds as well because there exists a "parent" of ℓ in EXE which will not be deleted. For any executable ℓ' in E' there exists a chain $C \leq \mathcal{E}_\Gamma[\![E']\!]$ such that all the labels from C are never deleted from R according to the induction hypothesis. We can make a conclusion that $\ell' \notin K''$ always holds. □

Similarly to the algorithm for computing the least fixed point of $reach_M$, computing the greatest fixed point of $unreach_{R,EXE,K_{set},M}$ can be implemented with the time and space complexity of $O(|Labs(F)| * |\mathfrak{T}|)$: for each label which is

$$
\begin{array}{rcl}
rgen(L) & = & \{\ell \in L | \exists C \in \mathfrak{T}\ (dom(C) \subseteq L) \wedge (\ell \in dom(G_{ch}(C)))\} \\
rexe(L) & = & \{\ell \in L | \exists C_1, C_2 \in \mathfrak{T}\ (dom(C_1) \cup dom(C_2) \subseteq L) \wedge \\
 & & (\ell \in dom(G(C_1))) \wedge (\ell \in dom(C_2))\}
\end{array}
$$

Table 5.8 – Definition of the endofunctions $rgen : 2^{\mathbf{Lab}} \to 2^{\mathbf{Lab}}$ and $rexe : 2^{\mathbf{Lab}} \to 2^{\mathbf{Lab}}$, given an IMC$^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9.

potentially deletable from the set of reachable labels we should check in the worst case whether none of its "parents" are executable, i.e. whether there is no chain containing one of the "parents" which is fully reachable. All the labels from $Labs(F)$ can become potentially deletable from the set of reachable labels and each label can participate in maximum $|\mathfrak{T}|$ chains. The complexity will be however much lower in the average case.

5.3.3 Repeatable reachability

In some cases we would like to determine whether some labels are not "simply" reachable, but reachable unlimited number of times, i.e. whether they represent "renewable resources". We also would like to compute (an over-approximation of) the set of labels which can be executed unlimited number of times. This can be useful for determining liveness properties. In the following we will determine which labels can be generated/executed infinite number of times along at least one existing infinite transition sequence and not necessary along every infinite transition sequence.

We have defined in Table 5.8 two functions *rgen* and *rexe* whose output is a subset of their input. They have the greatest fixed points on every finite lattice 2^S with $S \subseteq \mathbf{Lab}$.

We will prove in Lemma 5.9 that, given the Pathway Analysis results in $\mathcal{F}$ as in Definition 4.9 of an IMC$^{\mathrm{G}}$ program F, any label that can be repeatedly generated along some transition sequence is in $GFP(rgen)$ on $2^{Labs(F)}$ and any label that can be repeatedly executed along some transition sequence is in $GFP(rexe)$ on $2^{Labs(F)}$. The basic ideas of the proof for these facts are schematically presented in Figure 5.3 and Figure 5.4. In the first case $GFP(rgen)$ is a union of labels from which a sequence of chains can be formed that reproduces itself and all the labels that are generated by these chains. Therefore $\ell \in GFP(rgen)$ if it is just generated by one of the chains: $\ell_1 \in dom(G_{ch}(C_1))$ in Figure 5.3. In the second case $GFP(rexe)$ is a union of labels from the domains of the chains itself – therefore $\ell_1 \in dom(C_1)$ in Figure 5.4.

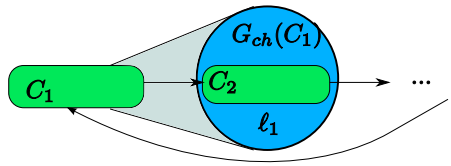

Figure 5.3 – Illustration for the computation of labels' repeatable reachability.

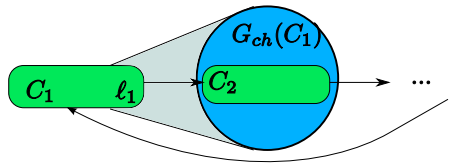

Figure 5.4 – Illustration for the computation of labels' repeatable executability.

Lemma 5.9 (GFP of *rgen* and *rexe*)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F *and* $\mathcal{F}$ *as in Definition 4.9, if there exists a transition sequence starting from* F *such that* ℓ *is generated infinitely often then* $\ell \in GFP(rgen)$ *on* $2^{Labs(F)}$*. Moreover, if there exists a transition sequence starting from* F *such that* ℓ *is executed infinitely often then* $\ell \in GFP(rexe)$ *on* $2^{Labs(F)}$*.*

Proof. The proof is similar to the proof of Lemma 5.5. There exist the greatest fixed points of both *rgen* and *rexe* because $2^{Labs(F)}$ is a finite complete lattice with set inclusion as a partial order and the functions *rgen* and *rexe* are order-reversing.

It is enough to show that for each $\ell \in Lab(F)$ that is generated unlimited number of times along some transition sequence starting from F holds $\ell \in fp$ for some fixed point $fp \in 2^{Lab(F)}$ such that $fp = rgen(fp)$. From this will follow $\ell \in GFP(rgen)$, as $fp \leq GFP(rgen)$ and the partial order in this case is the set inclusion. The same is relevant for any $\ell \in Lab(F)$ that is executed unlimited number of times along some transition sequence starting from F and the greatest fixed point of *rexe*.

If $\ell \in Lab(F)$ is generated infinitely often along some transition sequence starting from F then there exists a loop (as the number of states derivable from F is finite) $E_1 \xrightarrow[C_1]{} E_2 \xrightarrow[C_2]{} \ldots \xrightarrow[C_{n-1}]{} E_n \xrightarrow[C_n]{} E_1$ such that $\ell \in dom(G_{ch}(C_1))$ and $F \xrightarrow{*} E_1$. Assign $L := \bigcup_{i=1..n}(dom(C_i) \cup dom(G_{ch}(C_i)))$. Then L is a fixed

point of *rgen* because for each $\ell' \in L$ holds that ℓ' is generated by some chain C for which $dom(C) \subseteq L$ holds. The last is obvious if ℓ' has been added to L inside some $dom(G_{ch}(C_i))$. Otherwise, if ℓ' has been added to L inside some $dom(C_i)$, we can argue that each label inside C_i is also generated by some other chains from the set $\{C_j | 1 \leq j \leq n\}$. This is due to the fact that $K_{ch}(C_i) \geq C_i$ holds for any chain from $\mathfrak{T}$, i.e. all the labels from the chain C_i will be killed by execution of the chain and therefore will be generated consequently in order for the chain C_i to become executable again in the next loop. Altogether this proves that L is a fixed point of *rgen* and $\ell \in L$.

If $\ell \in Labs(F)$ can be executed infinitely often along some transition sequence starting from F then there exists a loop $E_1 \xrightarrow[C_1]{} E_2 \xrightarrow[C_2]{} ... \xrightarrow[C_{n-1}]{} E_n \xrightarrow[C_n]{} E_1$ such that $\ell \in dom(C_1)$ and $F \xrightarrow{*} E_1$. Assign $L := \bigcup_{i=1..n} dom(C_i)$. Then $\ell \in L$ clearly holds. Moreover L is a fixed point of *rexe* because for each $\ell' \in L$ holds $\ell' \in C_i$ for some $1 \leq i \leq n$, but also $\ell' \in C_j$ for some $1 \leq j \leq n$: as ℓ' has been "consumed" after the execution of the chain C_1 is has to be generated by some consequent chain so that it will become executable again in the next loop. We have thus proved that L is a fixed point of *rexe* and $\ell \in L$. □

We could perform the computation of the greatest fixed points of *rgen* and *rexe* on the domain of reachable labels of an initial IMC^{G} program F or one of its derivations E – we can use the least fixed point of the defined in Table 5.6 function *reach*, i.e. $reach_{\mathcal{E}_\Gamma[\![F]\!]}$ or $reach_{\mathcal{E}_\Gamma[\![E]\!]}$, in order to determine which labels are reachable. We will obtain in this case a correct over-approximation of the labels that can be generated/executed infinite number of times along some transition sequences starting either from F or from E.

Similarly to the algorithms in Sections 5.3.1 and 5.3.2, the computation of the greatest fixed points of *rgen* and *rexe* can be implemented with the time and space complexities of $O(|Labs(F)| * |\mathfrak{T}|)$: each label ℓ in $Labs(F)$ can be potentially repeatedly reached/executed and at most all the chains from $\mathfrak{T}$ should be checked in order to determine whether there exists a chain all the labels in the domain of which can be repeatedly reached/executed and which generates/contains ℓ.

There are many other properties of labels that could be defined as the greatest/least fixed points of functions on label sets. Our developments represent an exploration of possibilities. We are confident that we can develop our methods further in order to check a broader class of system properties by Static Analysis methods.

5.4 Expected time reachability

In the previous section we have considered the question how to compute a set of labels that contains all the labels reachable from some initial configuration regardless of the time it takes. In this section we will discuss how the lower bound for the expected time to reach a set of (in principle reachable) labels can be computed. Only expected time can be computed for IMC^{G} systems because their delay transitions do not have fixed durations but their durations are instead exponentially distributed with corresponding rates. We will compute the minimum expected time to reach one state in an IMC^{G} system from another state, both represented by sets of their exposed labels. We will often call a set of exposed labels in a state from which we start the computation the *initial* label set and a set of exposed labels in a state to reach – the *goal* label set.

As the non-determinism is inherent for IMC^{G} systems we should first of all decide on a scheduler or a class of schedulers that will be used for resolving non-deterministic choices. We consider a class of particularly simple schedulers whose decisions are dependent only on states in which a decision has to be made but not on a history before reaching the state or the time that have passed until the current state has been reached from the initial state. These schedulers are called *Static Markovian Deterministic* schedulers [Put94], *positional* schedulers [NSK09] or just *simple* schedulers. After all the decisions of a simple scheduler are fixed, an IMC^{G} system can be in fact reduced to a Continuous-Time Markov Chain. We will be looking for a simple scheduler which makes such decisions that the expected time to simultaneously reach all the labels from the goal set is minimal.

We will do computations on a subclass of IMC^{G}, introducing some serious syntactic constraints. First of all, we consider only linear IMC^{G} processes; the second constraint is that we may have a choice only between external actions but not between delays – see Table 5.9 for the derivation rules for suitable IMC^{G} programs. The exact reason for these constraints will be explained below in Section 5.4.2 but we want to make a remark at this point that the introduced constraints allow us to confine ourself to a simple algorithm on the one hand and to guarantee that the computation is exact (in particular, because no blocking due to the failed synchronisation can occur in linear IMC^{G} processes) on the other hand.

The expected time reachability has been considered for probabilistic timed automata in [KNPS06]. The standard algorithm for computing minimal and maximal (for goal states that are reachable with probability 1) expected reachability time for MDPs with fixed time associated with action transitions can be found in [Alf99] and is easily adaptable to many similar systems. It has been pointed

out that computing expected reachability time corresponds in fact to computing the expected cost or reward accumulated until reaching a goal state, with the cost/reward function equal to zero for action-transitions and equal to transition durations for delay transitions [KNPS06]. Our algorithm that will be presented in Sections 5.4.3-5.4.4 can be adapted to computing other cost/reward functions then expected reachability time – we see this as a possible topic for future work.

Calculating minimal expected time reachability is a subclass of an intensively studied stochastic shortest path problem [BT91] but in our setting, with the introduced syntactic constraints and simple schedulers, the first stage of the algorithm (see Section 5.4.3) becomes in fact just an instance for the well-known Floyd's algorithm (also called Floyd-Warshall algorithm) for finding a shortest path in a directed graph for all state pairs [Flo62]. The difference is that we compute "shortest distances" between labels, not between states as in [Flo62]. We need consequently the second stage in the algorithm for post-processing the results of the first stage – see Section 5.4.4.

In the first stage of the algorithm we calculate thus the minimal expected durations (that we sometimes call *distances*) of label paths between the labels. Calculating distances between the labels has the complexity of Floyd's algorithm which is cubic for time and quadratic for space, in our case it is accordingly $|Labs(F)|^3$ and $|Labs(F)|^2$ with F an IMC$^{\mathrm{G}}$ program to be analysed. We can sum expected durations of transitions constituting a path between two labels because the expected value of a sum of random variables is equal to the sum of expected values of random variables [GS97]. Random variables are in our case durations of delay-transitions in an IMC$^{\mathrm{G}}$ system and their expected values are equal to one divided by their rates.

The post-processing can be implemented as linear in the number of labels in the initial label set plus the number of labels in the goal label set. These are clearly lower time/space complexities than those that would be caused by applying Floyd's algorithm to a labelled transition system representing the semantics of F. In the last case the complexity will dependent on the number of states in the labelled transition system which can be up to $2^{|Labs(F)|}$. We have currently rather strict syntactic constraints but we are confident that our method can be extended at least to the full linear fragment of IMC$^{\mathrm{G}}$ and may be also to some larger subclasses of IMC$^{\mathrm{G}}$.

Our algorithm can be useful, for example, in the verification related to reliability of systems. Reliability is often understood as a system parameter measuring the time interval in which a system will operate faultlessly with a sufficiently high probability. It can be expressed, for example, as a *mean time to failure* (MTTF) for irreparable systems or a *mean time between failures* (MTBF) for systems that can be repaired [Abe00]. In this setting our algorithm computes MTTF (if the

goal set of labels represents a failure state) in the worst case, i.e. with the most unfavourable scheduler, for systems representable in a linear fragment of $\mathrm{IMC}^{\mathrm{G}}$ with an additional restriction on the choice operator (see Table 5.10).

5.4.1 General description of the algorithm

We will first describe our algorithm for a simple case when the initial and the goal set each contain only one label, i.e. we will compute the minimum expected time to reach one label ℓ (speaking more precisely, to reach any multiset M such that $\ell \in dom(M)$) from another label ℓ' (from any multiset M' such that $\ell' \in dom(M')$). By "reaching one label from another label" we mean that it should hold $\ell' \xrightarrow{*} \ell$, i.e. there should exist a label path from ℓ' to ℓ. We will show below that such label paths have a direct relation to paths between the multisets M' and M.

Take as an example the following $\mathrm{IMC}^{\mathrm{G}}$ program, for which we would like to determine what is the minimum (over all possible simple schedulers) expected time to reach ℓ_1 from ℓ_2: $X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.(\mathsf{c}^{\ell_3}.\lambda_1^{\ell_4}.\lambda_2^{\ell_5}.X + \mathsf{d}^{\ell_6}.\lambda_3^{\ell_7}.\lambda_4^{\ell_8}.X)$. From the syntax of the program it is clear that there are two possible label paths: $\ell_2 \longrightarrow \ell_3 \longrightarrow \ell_4 \longrightarrow \ell_5 \longrightarrow \ell_1$ and $\ell_2 \longrightarrow \ell_6 \longrightarrow \ell_7 \longrightarrow \ell_8 \longrightarrow \ell_1$. The expected time for the first of the paths is $\frac{1}{\lambda_1} + \frac{1}{\lambda_2}$ and the expected time for the second path is $\frac{1}{\lambda_3} + \frac{1}{\lambda_4}$. This follows from the properties of the exponential distribution and of the expected value operator, see Section 2.1: the expected transition duration is 1 divided by the transition rate and the expected value of the sum is the sum of expected values. In the example we can therefore simply compute the minimum expected time to reach ℓ_1 from ℓ_2 as $min(\frac{1}{\lambda_1} + \frac{1}{\lambda_2}, \frac{1}{\lambda_3} + \frac{1}{\lambda_4})$.

The syntactic constraints that will be introduced in Section 5.4.2 guarantee that there cannot occur situations where the choice is between actions with the same name (that a scheduler cannot distinguish) or between several delay transitions (where the choice is done probabilistically), therefore a simple scheduler has a full freedom in the choice of a label path. In the example above the choice between the execution of ℓ_3 and ℓ_6 is fully deterministic and depends on a concrete scheduler that is used.

We will compute the expected time to reach one label from another one by first computing expected time to reach labels directly, i.e. by generating them according to the generate operator from the Pathway Analysis. In the above example the defined by us in Section 5.4.3 operator tr will return $tr(\ell_1, \ell_2) = 0$, $tr(\ell_2, \ell_3) = 0$, $tr(\ell_3, \ell_4) = 0$, $tr(\ell_4, \ell_5) = \frac{1}{\lambda_1}$, $tr(\ell_5, \ell_1) = \frac{1}{\lambda_2}$ etc. It is clear that for actions the expected time is zero and for delay rates the expected time is 1

divided by the rate. It is also clear that in case $\ell \nrightarrow \ell'$ we cannot reach ℓ' from ℓ directly and therefore assign $tr(\ell, \ell') = \infty$: in our example $tr(\ell_1, \ell_3) = \infty$ etc.

As a next phase we will compute minimum expected durations of label paths between not directly "connected" labels, i.e. between all ℓ and ℓ' such that $\ell \nrightarrow \ell'$ but $\ell \xrightarrow{*} \ell'$. As there exists one shortest path that can be chosen by a scheduler (or several paths with the same expected duration), we can use a well-known Floyd's algorithm [Flo62] for finding a path with the shortest expected duration based on the results of the *tr* operator. The computed shortest path durations between all pairs of labels will be saved in the mapping *et*. In our example $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.(\mathsf{c}^{\ell_3}.\lambda_1^{\ell_4}.\lambda_2^{\ell_5}.X + \mathsf{d}^{\ell_6}.\lambda_3^{\ell_7}.\lambda_4^{\ell_8}.X)}$ we will have $et(\ell_2, \ell_3) = tr(\ell_2, \ell_3) = 0$ and $et(\ell_3, \ell_4) = tr(\ell_3, \ell_4) = 0$, while the distance between ℓ_2 and ℓ_4 will be computed as $min(tr(\ell_2, \ell_3) + tr(\ell_3, \ell_4), tr(\ell_2, \ell_4)) = min(0, \infty) = 0$.

Until now we have talked about reaching one label from another label. We will however in general discuss reaching a set of labels from another set of labels. First of all note that in order to make a meaningful statement by using the computed mapping *et* we need to be sure that both sets represent (the initial state is equal and the goal set is a subsets of) exposed labels of two $\mathrm{IMC}^{\mathrm{G}}$ expressions reachable from an initial $\mathrm{IMC}^{\mathrm{G}}$ program on which *et* has been computed. Otherwise if in the example above we would try to compute time to reach both ℓ_2 and ℓ_3 from ℓ_1 by taking, for example, a sum, minimum or some other function of $et(\ell_1, \ell_2)$ and $et(\ell_1, \ell_3)$, we would not get any satisfying answer: ℓ_2 and ℓ_3 just cannot be exposed together. We could use the function *Valid* defined in Section 5.1 in order to rule out impossible combinations of labels: it would be able to determine that ℓ_2 and ℓ_3 exclude each other.

The next aspect is that if the initial set of labels contains several labels then only one of them can be executed and the rest will be killed. This is because we are only considering a linear fragment of $\mathrm{IMC}^{\mathrm{G}}$, so if two labels are exposed at the same time then they are connected by the choice construct. We need therefore to take the minimum over all the labels in the initial set of their minimal expected time to reach the labels from the goal set. Concretely, in our example $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.(\mathsf{c}^{\ell_3}.\lambda_1^{\ell_4}.\lambda_2^{\ell_5}.X + \mathsf{d}^{\ell_6}.\lambda_3^{\ell_7}.\lambda_4^{\ell_8}.X)}$ with the initial set $\{\ell_3, \ell_6\}$ and the goal set $\{\ell_1\}$ if $\frac{1}{\lambda_1} + \frac{1}{\lambda_2} < \frac{1}{\lambda_3} + \frac{1}{\lambda_4}$ then we choose ℓ_3 otherwise we will choose ℓ_6.

As for the goal set, all the labels in it are also connected by the choice construct, but this time we need to take the maximum over all the labels in the set. If one of the labels in the goal set is not reachable from all the labels in the initial set, then the goal set is considered to be not reachable in general and our algorithm will return infinity as the minimum expected time. But also in case all the labels from the goal set are reachable, if we take a minimum also over the labels in

the goal set, we might be computing the minimum expected time to reach only a subset of the goal set. We will discuss this question in Section 5.4.4 in more detail.

This was basically the full description of our algorithm. We will additionally discuss the relation between the "goal set" and the exposed labels of concrete $\mathrm{IMC}^{\mathrm{G}}$ expressions that the goal set is a subset of. With our method we do not guarantee that we can reach an expression with only labels from the goal set exposed in the expected time that our algorithm computes, even if our algorithm computes a value different from infinity – because some additional labels may be exposed as well. In the example $\mathsf{a}^{\ell_1}.\underline{Y := \mathsf{b}^{\ell_2}.\lambda^{\ell_3}.\mathsf{c}^{\ell_4}.Y} + \mathsf{d}^{\ell_5}.\lambda^{\ell_6}.\mathsf{e}^{\ell_7}.\mathbf{0}$ if we compute the minimum expected time to reach $\{\ell_2\}$ from $\{\ell_1\}$ then we compute in fact the minimum expected time to reach $\{\ell_2, \ell_5\}$ from $\{\ell_1\}$. Note that there is in fact a configuration where $\{\ell_2\}$ is reachable from $\{\ell_1\}$ without ℓ_5 being simultaneously exposed (after one unfolding of the process definition for Y). We will not however take into account such cases where we may need several executions of the *same* labels in order to reach the goal configuration. We will only take into account the *first* time labels in the goal set are reached.

We will prove in Section 5.4.4 that, accepting a possibility that the goal set is just a subset of exposed labels, our computations deliver the exact result for the chosen subclass of $\mathrm{IMC}^{\mathrm{G}}$ and the chosen class of schedulers. The extension of our method to the full linear fragment of $\mathrm{IMC}^{\mathrm{G}}$ is relatively easy, but the extension for the parallel operator is much harder. If there are two parallel processes that both arrive at delay transitions then those delay transition that completes first will continue, and this leads to the situation where we cannot simply add the expected durations of label paths. For example, for the $\mathrm{IMC}^{\mathrm{G}}$ program $\lambda_1^{\ell_1}.\mathsf{a}^{\ell_2}.\mathbf{0} \;|||_{\emptyset}|||\; \lambda_2^{\ell_3}.\mathsf{b}^{\ell_4}.\mathbf{0}$ the expected time to reach $\{\ell_2, \ell_4\}$ from $\{\ell_1, \ell_3\}$ is strictly smaller than the expected time to reach $\{\ell_2\}$ from $\{\ell_1\}$ plus the expected time to reach $\{\ell_4\}$ from $\{\ell_3\}$. It is $\frac{\lambda_2}{\lambda_1+\lambda_2} * (\frac{1}{\lambda_1+\lambda_2} + \frac{1}{\lambda_1}) + \frac{\lambda_1}{\lambda_1+\lambda_2} * (\frac{1}{\lambda_1+\lambda_2} + \frac{1}{\lambda_2}) = \frac{\lambda_1+\lambda_2}{\lambda_1*\lambda_2} - \frac{1}{\lambda_1+\lambda_2}$ in the first case and $\frac{1}{\lambda_1} + \frac{1}{\lambda_2} = \frac{\lambda_1+\lambda_2}{\lambda_1*\lambda_2}$ in the second case, see Section 2.1 for the properties of exponential distributions.

Besides, as we allow the creation of a finite number of new subprocesses in the full $\mathrm{IMC}^{\mathrm{G}}$, it should be taken into account that a label can give rise to several labels that are in parallel subprocesses which additionally complicates the computations. We would namely need to additionally remember the "forking points" and which labels can be generated only after passing these forking points. Therefore we have decided to first present our ideas for the linear fragment of $\mathrm{IMC}^{\mathrm{G}}$ where such situations do not arise. We leave the question of extendibility of our method to other subclasses of $\mathrm{IMC}^{\mathrm{G}}$ as a topic for future research.

$$\vdash^{tr} \mathsf{a}^{\ell}.X \tag{1}$$

$$\vdash^{tr} \lambda^{\ell}.X \tag{2}$$

$$\frac{\vdash^{tr} P}{\vdash^{tr} \mathsf{a}^{\ell}.P} \tag{3}$$

$$\frac{\vdash^{tr} P}{\vdash^{tr} \lambda^{\ell}.P} \tag{4}$$

$$\frac{(\vdash^{tr} P_1) \wedge (\vdash^{tr} P_2)}{\vdash^{tr} P_1 + P_2}$$

$$if \quad (dom(\mathcal{E}_\Gamma[\![P_1]\!]) \cap dom(\mathcal{E}_\Gamma[\![P_2]\!]) = \emptyset) \wedge (dom(\mathcal{E}_\Gamma[\![P_1]\!]) \cup dom(\mathcal{E}_\Gamma[\![P_2]\!]) \subseteq \mathbf{Act}) \tag{5}$$

$$\frac{\vdash^{tr} P}{\vdash^{tr} \underline{X := P}} \tag{6}$$

$$\vdash^{tr} X \tag{7}$$

$$\vdash^{tr} \mathbf{0} \tag{8}$$

Table 5.9 – Derivation rules for $\mathrm{IMC}^{\mathrm{G}}$ programs suitable for our expected time reachability analysis. $X \in \mathbf{Var}$, $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $\lambda \in \mathbf{Rate}$, $\ell \in \mathbf{Lab}$.

5.4.2 Syntactic constraints

In this section we will introduce a number of syntactic constraints that we will additionally put on $\mathrm{IMC}^{\mathrm{G}}$ programs so that our method for computing the minimum expected reachability time becomes applicable. We will require that for an $\mathrm{IMC}^{\mathrm{G}}$ program F the fact $\vdash^{tr} F$ can be proved. The derivation rules for $\vdash^{tr}$ are given in Table 5.9. We could have integrated the well-formedness rules in Table 3.8 and the rules in Table 5.9 into one set of rules but we are not doing this for the sake of simplicity – i.e. for the rules in Table 5.9 to represent only "new" restrictions on $\mathrm{IMC}^{\mathrm{G}}$ expressions.

Let us now discuss the rules for $\vdash^{tr}$ in detail. First, note that there are no parallel operator and no hide-operator rules. We have given in Section 5.4.1 an example illustrating the difficulties with the parallel operator. Concerning the hide-operator, in case some of the actions will become internalised then a simple scheduler could not differentiate between them and the computed by our algorithm result will not be tight (but will be still a lower bound). Second, there are only external actions and no delays in the choice construct (see the

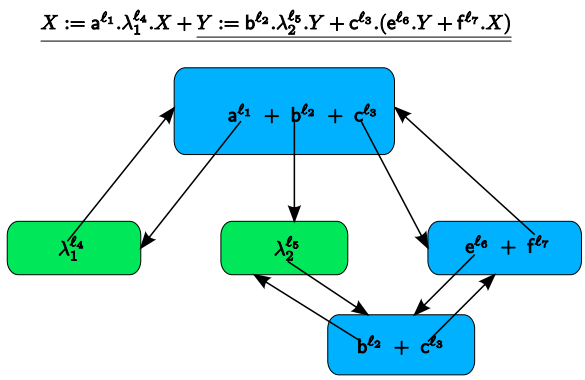

Figure 5.5 – Example of an IMC$^{\mathrm{G}}$ program F for which $\vdash^{tr} F$ holds. Arrows go from labels to sets of labels that the first generate after their execution.

additional condition in rule (5) in Table 5.9). This means that there are no probabilistic alternatives in case we have a deterministic scheduler. Expressions similar to $\underline{X := \mathsf{a}^{\ell_1}.(\lambda_1^{\ell_2}.X + \lambda_2^{\ell_3}.X)}$ are therefore not allowed: a scheduler cannot deliberately choose between ℓ_2 and ℓ_3, as the probability of executing ℓ_2 is always $\frac{\lambda_1}{\lambda_1+\lambda_2}$ and the probability of executing ℓ_3 is always $\frac{\lambda_2}{\lambda_1+\lambda_2}$ irrespective of a scheduler. Third, all choice alternatives, which are external actions as just mentioned, are different (see the additional condition in rule (5) in Table 5.9). A scheduler can therefore always choose any of choice alternatives and thus always determine "which way to go", i.e. which path to take.

See Figure 5.5 for an example of an IMC$^{\mathrm{G}}$ program whose syntax satisfies the conditions of $\vdash^{tr}$. States in the semantics of the example IMC$^{\mathrm{G}}$ program are denoted by their corresponding exposed actions/rates and labels. Note that the "green" (Markovian) states have only one exposed rate while there can be several exposed actions in the "blue" (interactive) states. There are no hybrid (i.e. with both actions and rates exposed) states in the example and there cannot be any hybrid states in the semantics of any IMC$^{\mathrm{G}}$ program F for which $\vdash^{tr} F$ holds.

The reason for the introduction of syntactic constraints via $\vdash^{tr}$ is that we are aiming at constructing an algorithm with a low complexity. With the restrictions computations on IMC$^{\mathrm{G}}$ programs become much simpler due to the absence of parallel delay transitions etc. Moreover, simple schedulers obtain the full control over the paths taken. There is also a lower memory consumption because we do not need to construct beforehand a labelled transition system representing the semantics of an IMC$^{\mathrm{G}}$ program F that we are analysing. Instead of that we only save a matrix of the size $|Labs(F)|^2$: each pair of labels

$(\ell, \ell') \in Labs(F) \times Labs(F)$ has the minimum expected time to reach ℓ' from ℓ assigned to it. Remember that a number of states in the labelled transition system for F can be up to $2^{|Labs(F)|}$, so we would need to save a matrix of the size $2^{|Labs(F)|*2}$ if we would compute distances between states in the labelled transition system.

In Lemma 5.14 below we will formally show that syntactic restrictions in Table 5.10 are enough to guarantee that the expected duration of one "shortest" path is equal to the minimum expected duration over all paths between the initial and the goal set of labels under the class of simple schedulers. It is therefore relatively easy to compute – it is enough to find one of the "shortest" paths (there can be more than one) with the least expected duration. We will additionally need to ensure that there exists a simple scheduler such that all labels in the goal set are reachable from some of the labels in the initial set with the probability 1. If this is not the case then we return infinity. In the last case the probability is always 0 as we do not have probabilistic choices in an IMC$^{\mathrm{G}}$ program F and its derivatives if $\vdash^{tr} F$ holds – see Lemma 5.14.

5.4.3 Expected durations of label paths

In this section we will formally present an algorithm for computing the minimum expected time to reach one label from another one. We will compute the minimum expected time over all possible label paths between the labels. As usual, at the beginning we fix an IMC$^{\mathrm{G}}$ program F for which we will do all the computations. In fact all the definitions and algorithms from this section are applicable to an arbitrary IMC$^{\mathrm{G}}$ program F but putting them together in Section 5.4.4 gives a meaningful result only for such programs for which $\vdash^{tr} F$ holds.

We start by defining the operator tr in Table 5.10 which computes the expected time to reach from labels in $Labs(F)$ their directly generated labels. Labels corresponding to actions reach their generated labels in zero time while labels corresponding to delay rates reach their generated labels in the expected time of delay. We have the least upper bound operator (see Table 5.11) in the definition of tr, but for uniquely labelled IMC$^{\mathrm{G}}$ expressions we do not have situations where there are two or more ways to generate one label from another one, therefore for an IMC$^{\mathrm{G}}$ program F we will just create a union of tr application results on the subexpressions of F.

We then compute a kind of transitive closure of the relation tr. The idea is to compute the "least expected time" to reach one label from another one by possibly transiting other labels in between. We compute the mapping $et \in$

$$
\begin{array}{rcll}
tr_\Gamma[\![\mathsf{a}^\ell.X]\!] & = & \bigsqcup_{\ell' \in dom(\mathcal{E}_\Gamma[\![X]\!])} \{(\ell, \ell', 0)\} & (1) \\
tr_\Gamma[\![\lambda^\ell.X]\!] & = & \bigsqcup_{\ell' \in dom(\mathcal{E}_\Gamma[\![X]\!])} \{(\ell, \ell', \frac{1}{\lambda})\} & (2) \\
tr_\Gamma[\![\mathsf{a}^\ell.P]\!] & = & \bigsqcup_{\ell' \in dom(\mathcal{E}_\Gamma[\![X]\!])} \{(\ell, \ell', 0)\} \sqcup tr_\Gamma[\![P]\!] & (3) \\
tr_\Gamma[\![\lambda^\ell.P]\!] & = & \bigsqcup_{\ell' \in dom(\mathcal{E}_\Gamma[\![X]\!])} \{(\ell, \ell', \frac{1}{\lambda})\} \sqcup tr_\Gamma[\![P]\!] & (4) \\
tr_\Gamma[\![P_1 + P_2]\!] & = & tr_\Gamma[\![P_1]\!] \sqcup tr_\Gamma[\![P_2]\!] & (5) \\
tr_\Gamma[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & tr_\Gamma[\![P]\!] & (6) \\
tr_\Gamma[\![P_1 \between A \between P_2]\!] & = & tr_\Gamma[\![P_1]\!] \sqcup tr_\Gamma[\![P_2]\!] & (7) \\
tr_\Gamma[\![\underline{X := P}]\!] & = & tr_\Gamma[\![P]\!] & (8) \\
tr_\Gamma[\![\mathbf{0}]\!] & = & \emptyset & (9)
\end{array}
$$

Table 5.10 – Operator $tr : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab} \times \mathbf{Lab} \times \mathbb{R}_0^+}$, environment $\Gamma \in 2^{X \times \mathfrak{M}}$, given an $\mathrm{IMC}^{\mathrm{G}}$ program F and $\mathcal{F}$ as in Definition 4.9.

$$
\begin{array}{rcl}
T_1 \sqcup T_2 & = & \bigcup_{(\ell_1, \ell_1', t_1) \in T_1} \\
& & \begin{cases} \{(\ell_1, \ell_1', t_1)\} & \text{if } \nexists t_2 \ st\ (\ell_1, \ell_1', t_2) \in T_2 \\ \{(\ell_1, \ell_1', t_1)\} & \text{if } (\exists t_2 \ st\ (\ell_1, \ell_1', t_2) \in T_2) \wedge (t_1 < t_2) \end{cases} \cup \\
& & \bigcup_{(\ell_2, \ell_2', t_2) \in T_2} \\
& & \begin{cases} \{(\ell_2, \ell_2', t_2)\} & \text{if } \nexists t_1 \ st\ (\ell_2, \ell_2', t_1) \in T_1 \\ \{(\ell_2, \ell_2', t_2)\} & \text{if } (\exists t_1 \ st\ (\ell_2, \ell_2', t_1) \in T_1) \wedge (t_2 < t_1) \end{cases}
\end{array}
$$

Table 5.11 – Definition of the least upper bound operator on the domain $2^{\mathbf{Lab} \times \mathbf{Lab} \times \mathbb{R}_0^+}$.

$Labs(F) \times Labs(F) \to \mathbb{R}_0^+ \cup \{\infty\}$ starting by the initialisation of all pairs of labels in et by infinity apart from the case that both labels are equal – we assign namely $et(\ell, \ell) = 0$ for all $\ell \in Labs(F)$. For pairs of labels where the first label can generate the second one directly we assign the distance in et according to the results of $tr_\Gamma[\![F]\!]$.

We proceed with recomputing expected times between labels, taking the minimum whenever there are several alternatives. Taking the minimum is done in a clever way, applying the Floyd's algorithm [Flo62]: it is in succession checked for all pairs of labels whether one of label paths through more and more labels is "shorter" (has a smaller expected duration) then the currently computed minimum expected duration of label paths from the first to the second label in the pair. So the infinite duration in $et(\ell_1, \ell_2)$ after the execution of the algorithm is finished means that there is no path between ℓ_1 and ℓ_2. See a more formal description of the algorithm in Table 5.12.

```
for all (ℓ1, ℓ2) ∈ Labs(F) × Labs(F)
 if (ℓ1 = ℓ2)  then et(ℓ1, ℓ2) := 0;
 else   et(ℓ1, ℓ2) := ∞;

for all (ℓ, ℓ', t) ∈ trΓ[[F]]
   et(ℓ, ℓ') := t;

for k := 1 to |Labs(F)|
   for i := 1 to |Labs(F)|
      for j := 1 to |Labs(F)|
         et(ℓi, ℓj) = min(et(ℓi, ℓj), et(ℓi, ℓk) + et(ℓk, ℓj));
```

Table 5.12 – Algorithm for computing the mapping $et : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Lab}\times\mathbf{Lab}} \to \mathbb{R}_0^+ \cup \{\infty\}$, given an $\mathrm{IMC}^{\mathrm{G}}$ program F, $\mathcal{F}$ as in Definition 4.9 and $tr_\Gamma[\![F]\!]$.

Note that we have made a choice that a label can reach itself in zero time. It seems like a natural decision but we could have made another decision that we are interested in the *next* appearance of the same label. For example, in the process $\underline{X} := \mathsf{a}^{\ell_1}.\lambda^{\ell_2}.X$ we could have chosen to have $et(\ell_1, \ell_1) = \frac{1}{\lambda}$ instead of $et(\ell_1, \ell_1) = 0$. We have chosen the latter possibility mainly for the reasons of simplicity.

As a next step we will prove that the mapping *et* computed by the algorithm in Table 5.12 corresponds to the expected time needed to generate one label from another label. We will define the so-called *expected duration of a label path* and prove its relation to *et*. The computations of the expected durations of label paths are straightforward – the expected durations of all rate-transitions on a path are just added together. We call the function that returns the expected duration of a label path *time*. It is defined in Definition 5.10.

Definition 5.10 (Label path's expected duration)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F, $\mathcal{F}$ *as in Definition 4.9 and a set of labels* $\ell_1 ... \ell_n$ *such that* $\ell_i \longrightarrow \ell_{i+1}$ *for all* $1 \leq i < n$*, then the expected duration of the label path* $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ *is returned by the function time and is computed as*

$$time(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = \sum_{1 \leq i < n} \begin{cases} \frac{1}{Name(\ell_i)} & \text{if } Name(\ell_i) \in \boldsymbol{Rate} \\ 0 & \text{otherwise.} \end{cases}$$

We set $time(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = 0$ *if* $n = 1$.

We show in Lemma 5.11 that the expected durations of label paths are closely related to the mapping *et*: we have namely computed in $et(\ell_1, \ell_2)$ the lower bound for the expected durations of all label paths between ℓ_1 and ℓ_2 for all $(\ell_1, \ell_2) \in \mathit{Labs}(F) \times \mathit{Labs}(F)$, and this bound is tight.

Lemma 5.11 (Label path's minimum expected duration)**.** *Given an* IMC$^{\mathrm{G}}$ *program F, $\mathcal{F}$ as in Definition 4.9 and a label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$, then $\mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell_n) \geq et(\ell_1, \ell_n)$ holds. Moreover if $et(\ell_1, \ell_n) \neq \infty$ then there exists a label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ such that $\mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = et(\ell_1, \ell_n)$ holds. If $et(\ell_1, \ell_n) = \infty$ then there does not exist any label path from ℓ_1 to ℓ_n.*

Proof. We will prove the first statement of the lemma by induction on the number of steps in a label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$. For $n = 1$ the statement follows from assigning $et(\ell, \ell) := 0$ for all $\ell \in \mathit{Labs}(F)$ in the algorithm for computing *et* in Table 5.12. It is also the case that $\mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = 0$ for $n = 1$ according to Definition 5.10.

For $\ell_1 \longrightarrow \ell_n$, i.e. $n = 2$, the statement follows from the definition of the operator *tr* in Table 5.10, the definition of the generate operator from $\mathcal{F}$ as in Definition 4.9 and the unique labelling of F. We can namely see from the definition of the operator *tr* that for any label ℓ_n generated by ℓ_1 the operator *tr* returns (ℓ_1, ℓ_n, t), with t equal to one divided by the rate associated with ℓ_1 or equal to 0 if ℓ_1 is associated with an action. This is equal to $\mathit{time}(\ell_1 \longrightarrow \ell_n)$ and also to $et(\ell_1, \ell_n)$ according to the rules for the computation of *et*. Altogether $et(\ell_1, \ell_n) = \mathit{time}(\ell_1 \longrightarrow \ell_n)$.

Any label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ with $n > 2$ can be subdivided into two label paths $\ell_1 \longrightarrow ... \longrightarrow \ell'_n$ and $\ell'_n \longrightarrow ... \longrightarrow \ell_n$, with lengths of both paths being strictly smaller than the length of the label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$. We can therefore apply our induction hypothesis to both smaller label paths and obtain $\mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell'_n) \leq et(\ell_1, \ell'_n)$ and $\mathit{time}(\ell'_n \longrightarrow ... \longrightarrow \ell_n) \leq et(\ell'_n, \ell_n)$. It is easy to see from the algorithm for computing *et* that $et(\ell_1, \ell_n) \leq et(\ell_1, \ell'_n) + et(\ell'_n, \ell_n)$: this is clear if either $et(\ell_1, \ell'_n) = \infty$ or $et(\ell'_n, \ell_n) = \infty$ and follows from for-loops conditions otherwise. Altogether $\mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = \mathit{time}(\ell_1 \longrightarrow ... \longrightarrow \ell'_n) + \mathit{time}(\ell'_n \longrightarrow ... \longrightarrow \ell_n) \geq et(\ell_1, \ell'_n) + et(\ell'_n, \ell_n) \geq et(\ell_1, \ell_n)$ (the first equality follows from the definition of *time* function in Definition 5.10).

We have to argue now that there is in fact a label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ whose expected duration is equal to $et(\ell_1, \ell_n)$ if $et(\ell_1, \ell_n) \neq \infty$. It is clear in case $\ell_1 = \ell_n$: we have $et(\ell_1, \ell_n) = 0$ and there is a label path consisting only of ℓ_1 for which the expected duration is also zero according to Definition 5.10.

Otherwise the statement follows from the fact that each $et(\ell_1, \ell_n)$ has been calculated as a sum $et(\ell_1, \ell_n) = \sum_{i=1..n-1} et(\ell_i, \ell_{i+1})$ such that $et(\ell_i, \ell_{i+1}) = tr(\ell_i, \ell_{i+1})$ for $1 \leq i < n$. It is easy to see from the definition of the operator tr in Table 5.10 and the definition of the generate operator from $\mathcal{F}$ as in Definition 4.9 that $\ell_i \longrightarrow \ell_{i+1}$ holds for $1 \leq i < n$. The label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ has therefore the required property: it holds $time(\ell_1 \longrightarrow ... \longrightarrow \ell_n) = et(\ell_1, \ell_n)$.

If $et(\ell_1, \ell_n) = \infty$ then there does not exist any label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ because otherwise we would have $time(\ell_1 \longrightarrow ... \longrightarrow \ell_n) < \infty$ according to the definition of the function *time* in Definition 5.10 and therefore $et(\ell_1, \ell_n) < \infty$ according to the already proved facts, which is a contradiction. □

5.4.4 Computing minimum expected reachability time

In this section we will use the mapping *et* from Section 5.4.3 in order to compute a lower bound for the expected time on paths between multisets. As already mentioned, we will have an initial set of labels and a goal set of labels as input parameters of our algorithm. We require that the initial set contains all the labels exposed in one of the states reachable from the analysed IMC^{G} program F – otherwise we can't make a meaningful prediction. Our algorithm returns infinity if at least one of the labels from the goal set cannot be reached from any of the labels in the initial set and we cannot know for sure whether this is the case if the initial set is just a subset of the actually exposed labels. The goal set is allowed to be smaller then the actual set of exposed labels of any derivative of F. We will compute a lower bound for the expected time until reaching a state with all the labels in the goal set being exposed and not necessary *exactly* the labels in the goal set being exposed.

Assume therefore that an initial set of labels contains exposed labels of one state and the goal set of labels contains a subset of exposed labels of at least one state in the labelled transition system corresponding to the semantics of F. Reaching the goal set of labels then implicitly means reaching one of the goal states – these are states that have all the labels from the goal set exposed, and there can be several such states. There can be several states having the same

1. check whether all the labels from L_2 are reachable from L_1, i.e. whether for all $\ell_2 \in L_2$ there exists $\ell_1 \in L_1$ with $et(\ell_1, \ell_2) \neq \infty$. If not, stop the execution of the algorithm and return $etsets(L_1, L_2) = \infty$;

2. return $etsets(L_1, L_2) = min_{\ell_1 \in L_1} max_{\ell_2 \in L_2} et(\ell_1, \ell_2)$.

Table 5.13 – Algorithm for computing the output of the operator $etsets : 2^{\mathbf{Lab}} \times 2^{\mathbf{Lab}} \to \mathbb{R}_0^+ \cup \{\infty\}$, given an IMC$^{\mathrm{G}}$ program F, $\mathcal{F}$ as in Definition 4.9 and a mapping $et : \mathbf{Lab} \times \mathbf{Lab} \to \mathbb{R}_0^+ \cup \{\infty\}$, $L_1 \in 2^{Labs(F)}$, $L_2 \in 2^{Labs(F)}$.

exposed labels from the initial label set in the labelled transition system as well, however, all of them will be bisimilar (see Section 4.2.2).

We will shortly describe the algorithm for reaching the goal set L_2 from the initial set L_1 before presenting it in a more formal way. The idea of the algorithm is first to decide whether all the labels in L_2 are reachable from L_1. If this is not the case then the defined by us function *etsets* returns infinity. If all the labels from L_2 are reachable from at least one label in L_1 then we return the result from the mapping *et*, taking the minimum over all the labels in L_1 and the maximum over all the labels in L_2.

Taking the minimum over the labels in L_1 should be intuitively clear: we are looking for the minimum expected time, hence a label in L_1 with the shortest distance to labels in L_2 is chosen. We will illustrate why we take the maximum over the labels in L_2 on the example of the IMC$^{\mathrm{G}}$ program $Y := \underline{X := \mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.(\mathsf{c}^{\ell_3}.X + \mathsf{d}^{\ell_4}.\lambda^{\ell_5}.Y)} + \mathsf{e}^{\ell_6}.Y$. Assume that $L_1 = \{\ell_2\}$ and $L_2 = \{\ell_1, \ell_6\}$. Then taking, for example, the minimum over L_2 would mean choosing the path through ℓ_3 which is however a path only to ℓ_1 exposed and not to ℓ_6 as well. Taking on the other hand the maximum over L_2 would mean determining the expected duration of the path through ℓ_4 which leads to both ℓ_1 and ℓ_6 exposed.

Schematically, if we want to compute the minimum expected time to reach a set of labels L_2 from another set of labels L_1, we need to do the following:

In the rest of the section we will prove that the algorithm in Table 5.13 computes what we have claimed above. We will first of all define the notion of the expected duration of a transition sequence. This notion will help us in further reasoning.

Definition 5.12 (Expected duration of transition sequences)**.** *Given a set of* IMC$^{\mathrm{G}}$ *expressions* $E_1 ... E_n$ *such that* $E_i \xrightarrow{\alpha_i} E_{i+1}$ *for all* $1 \leq i < n$*, then the expected duration of the transition sequence* $E_1 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_{n-1}} E_n$ *is returned by*

the function stime and is computed as

$$stime(E_1 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_{n-1}} E_n) = \sum_{1 \leq i < n} \begin{cases} \frac{1}{\alpha_i} & \text{if } \alpha_i \in \textbf{\textit{Rate}} \\ 0 & \text{otherwise.} \end{cases}$$

We set $stime(E_1 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_{n-1}} E_n) = 0$ *if* $n = 1$.

We will show in Lemma 5.13 how the expected duration of some transition sequence $E_1 \longrightarrow ... \longrightarrow E_n$ relates to $et(L, L')$ such that L is a set of exposed labels of E_1 and L' is contained in the set of exposed labels of E_n. Action names or delay rates decorating transitions between multisets are omitted in the statement and proof of Lemma 5.13 for simplicity but we implicitly assume that they are known for each $E_i \longrightarrow E_{i+1}$ for $1 \leq i < n$.

Lemma 5.13 (Expected durations of paths)**.** *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *such that* $\vdash^{tr} F$ *holds and* $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E_1$*, a transition sequence* $E_1 \longrightarrow ... \longrightarrow E_n$*,* L_1 *and* L_2 *sets of labels such that* $L_1 = dom(\mathcal{E}_\Gamma[\![E_1]\!])$ *and* $L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E_n]\!])$*, then* $stime(E_1 \longrightarrow ... \longrightarrow E_n) \geq etsets(L_1, L_2)$ *holds. Additionally* $E_1 \stackrel{*}{\not\to} E'$ *for all* $E' \in \{E'' | (F \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ *if and only if* $etsets(L_1, L_2) = \infty$*.*

Moreover, the estimation is tight in case $etsets(L_1, L_2) \neq \infty$*: there exists a transition sequence* $E_1 \longrightarrow ... \longrightarrow E_n$ *for some* $E_n \in \{E'' | (F \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ *such that* $stime(E_1 \longrightarrow ... \longrightarrow E_n) = etsets(L_1, L_2)$ *holds.*

Proof. It is clear that from $etsets(L_1, L_2) = \infty$ follows $E_1 \stackrel{*}{\not\to} E'$ for all $E' \in \{E'' | (F \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$: if $etsets(L_1, L_2) = \infty$ then at least one of the labels in L_2 is unreachable from none of the labels in L_1 according to the algorithm for *etsets* in Table 5.13. The other direction will follow from the yet to be proved consideration that the estimation in $etsets(L_1, L_2)$ is tight, i.e. there exists a corresponding transition sequence from E_1 to one of the elements in $\{E'' | (F \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$. Therefore it is not possible that $etsets(L_1, L_2) < \infty$ but $E_1 \stackrel{*}{\not\to} E'$ for all $E' \in \{E'' | (F \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$.

We will prove now that $stime(E_1 \longrightarrow ... \longrightarrow E_n) \geq etsets(L_1, L_2)$ holds for all transition sequences between E_1 and E_n with $L_1 = dom(\mathcal{E}_\Gamma[\![E_1]\!])$ and $L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E_n]\!])$. It is easy to see that for all $\ell_n \in dom(E_n)$ there exists a label $\ell_1 \in dom(E_1)$ and a label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ such that $stime(E_1 \longrightarrow ... \longrightarrow E_n) = time(\ell_1 \longrightarrow ... \longrightarrow \ell_n)$. Using the estimate from Lemma 5.11, we obtain $stime(E_1 \longrightarrow ... \longrightarrow E_n) = time(\ell_1 \longrightarrow ... \longrightarrow \ell_n) \geq et(\ell_1, \ell_n)$. As the first equality is applicable to all $\ell_n \in dom(\mathcal{E}_\Gamma[\![E_n]\!])$, we can maximise over all the labels in $dom(\mathcal{E}_\Gamma[\![E_n]\!])$. We can safely minimise over all those labels in $dom(\mathcal{E}_\Gamma[\![E_1]\!])$ from which all the labels in $dom(\mathcal{E}_\Gamma[\![E_n]\!])$ are reachable (but if some of the labels in $dom(\mathcal{E}_\Gamma[\![E_n]\!])$ are unreachable from some $\ell \in dom(\mathcal{E}_\Gamma[\![E_1]\!])$ then taking the maximum over the distances from ℓ to all the labels in $dom(\mathcal{E}_\Gamma[\![E_n]\!])$ will return infinity). Altogether $stime(E_1 \longrightarrow ... \longrightarrow E_n) \geq min_{\ell_1 \in dom(\mathcal{E}_\Gamma[\![E_1]\!])} max_{\ell_n \in dom(\mathcal{E}_\Gamma[\![E_n]\!])} et(\ell_1, \ell_n) = etsets(L_1, L_2)$ – the last equality uses the definition of *etsets* in Table 5.13.

It is now left to prove that the lower bound computed in $etsets(L_1, L_2)$ is tight. We know that according to Lemma 5.11 the bound computed in the mapping *et* is tight, i.e. there exists a label path whose expected duration is equal to $et(\ell, \ell')$ for all $\{\ell, \ell'\} \subseteq Labs(F)$. From the rules of $\vdash^{tr}$ in Table 5.10 we can see that there is no synchronisation, therefore no action blocking due to it, and there is also no reason to apply the internal progress rule for $\mathrm{IMC}^{\mathrm{G}}$ as there are no delays in the choice construct in F and hence in any of its derivative $\mathrm{IMC}^{\mathrm{G}}$ expressions. Therefore for each label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$ for which there exists an $\mathrm{IMC}^{\mathrm{G}}$ expression E_1 with $F \xrightarrow{*} E_1$ and $\ell_1 \in dom(E_1)$ there exists a transition sequence $E_1 \xrightarrow[C_1]{\alpha_1} ... \xrightarrow[C_{n-1}]{\alpha_{n-1}} E_n$ such that $\ell_n \in dom(E_n)$ where for each $E_i \xrightarrow[C_i]{\alpha_i} E_{i+1}$ it holds that $C_i = \bot_{\mathfrak{M}}[\ell_i \mapsto 1]$, $\alpha_i = Name(\ell_i)$ and $\ell_i \longrightarrow \ell_{i+1}$ is a corresponding transition in the label path $\ell_1 \longrightarrow ... \longrightarrow \ell_n$. From the given considerations and the definition of *etsets* in Table 5.13 follows a proof for $|L_1| = 1$ and $|L_2| = 1$.

Similarly to the discussion above about *etsets* being a lower bound, we can take the minimum over all the labels in L_1 and the maximum over all the labels in L_2, but this will only prove that our estimation in $etsets(L_1, L_2)$ is exact for reaching one of the labels in L_2 but not for reaching all of them simultaneously. It is therefore left to prove that all the labels from L_2 are simultaneously reached in case a label $\ell \in L_2$ with the maximum $et(\ell_1, \ell)$ is reached in a label path $\ell_1 \longrightarrow ... \longrightarrow \ell$, for $\ell_1 \in L_1$ arbitrary.

We will do an informal proof showing how the above statement follows from

the syntactic rules of IMC$^{\mathrm{G}}$ in Table 3.1. We start with a case $|L_2| = 2$ which can be consequently generalised. Remember that we assume that there exists E' such that $F \xrightarrow{*} E'$ and $L_2 \subseteq dom(E')$. From the rules for $\vdash^{tr}$ in Table 5.10 it easily follows that all the labels in L_2 are connected by the plus operator as in rule (5) in Table 5.10. We can therefore safely assume that $L_2 = \{\ell_2, \ell_3\}$ and $\underline{X_1 := ...\underline{X_n := \ell_2.P_1}} + \underline{Y_1 := ...\underline{Y_m := \ell_3.P_2}} \preceq F$ for some P_1 and P_2, $n \geq 0$, $m \geq 0$. Assume also that for some $\ell_1 \in L_1$ it holds $et(\ell_1, \ell_2) < et(\ell_1, \ell_3) < \infty$. We need to show that for any label path $\ell_1 \longrightarrow ... \longrightarrow \ell_3$, for any transition sequence $E \longrightarrow ... \longrightarrow E'$ that corresponds to the label path as in the discussion for $|L_1| = 1$ and $|L_2| = 1$, it holds $\ell_2 \in dom(E')$.

The situation described above ($et(\ell_1, \ell_2) \neq et(\ell_1, \ell_3)$) is only possible if $\ell_1 \in P_1$ or $\ell_1 \in P_2$. Assume without loss of generality that $\ell_1 \in P_1$. Moreover, it has to be the case that two different variables X and Y are reachable from ℓ_1 where one of the variables "generates" ℓ_2, i.e. has ℓ_2 in its definition, and the other variable "generates" ℓ_3. Otherwise, with only one variable, it would be the case that $et(\ell_1, \ell_2) = et(\ell_1, \ell_3)$. Schematically we have a subexpression of F of the type $\underline{X_1 := ...\underline{X_n := \ell_2....\ell_1...X...Y}} + \underline{Y_1 := ...\underline{Y_m := \ell_3.P_2}}$. We need to decide now on the scope of two variables X and Y. It is clear that ℓ_3 is not in the scope of one of the variables, let it be X. We have therefore something like $\underline{X_1 := ...\underline{X := ...\underline{X_n := \ell_2....\ell_1...X...Y}}} + \underline{Y_1 := ...\underline{Y_m := \ell_3.P_2}}$. It is clear that $\ell_3.P_2$ is in the scope of the variable Y. It is however not possible to have a situation $\underline{X := \ell_2....\ell_1...X...Y} + \underline{Y := \ell_3.P_2}$, for example, because Y will be free in the overall expression. It is therefore only possible to have an expression of the type $Y := ...(\underline{X_1 := ...\underline{X := ...\underline{X_n := \ell_2....\ell_1...X...Y}}} + \underline{Y_1 := ...\underline{Y_m := \ell_3.P_2}})$ as a subexpression of F. Altogether reaching ℓ_3 from ℓ_2 through the unfolding of the variable Y will lead to ℓ_2 being simultaneously exposed.

We can generalise the discussion above to $|L_2| = k > 2$ by arguing that assuming it has been inductively proved that there exists a variable X which "generates" $k - 1$ labels from L_2 then we can find another variable Y that generates both X and the remaining variable $\ell_k \in L_2$. □

In Lemma 5.14 we will show that there always exists a transition sequence $E_1 \longrightarrow ... \longrightarrow E_n$ whose expected duration is equal to the minimum expected time to reach E_n from E_1 under the class of simple schedulers.

Lemma 5.14 (Transition sequences and simple schedulers)**.** *Given an* IMC$^{\mathrm{G}}$ *program F such that $\vdash^{tr} F$ holds, $F \xrightarrow{*} E_1 \xrightarrow{*} E_n$, then there is a one-to-one correspondence between transition sequences from E_1 to E_n and simple*

schedulers such that the probability of taking a transition sequence $E_1 \longrightarrow ... \longrightarrow E_n$ *under a particular simple scheduler is equal to* 1.

Proof. It is enough to prove that for all IMC$^{\mathrm{G}}$ expressions E and E' such that $F \xrightarrow{*} E$ and $E \longrightarrow E'$ there exists on the one hand a simple scheduler that can choose a transition from E to E' with the probability 1, and on the other hand, any simple scheduler would choose one of the transitions from E with probability 1. It is therefore enough to show that all the transitions from E are decorated by different action names, i.e. if $E \xrightarrow[c_1]{\alpha_1} E'$ and $E \xrightarrow[c_2]{\alpha_2} E''$ such that $E' \neq E''$ then $\alpha_1 \in \mathbf{Act}$, $\alpha_2 \in \mathbf{Act}$ and $\alpha_1 \neq \alpha_2$ hold. We can prove this by induction on the number of steps in the derivation $F \xrightarrow{*} E$ while proving at the same time that this also holds for all subexpressions $E'' \preceq E$.

If $E = F$ then the statement holds due to the rules for $\vdash^{tr} F$ in Table 5.9, in particular, rule (5), and can be proved by structural induction on F. Otherwise we have to prove that if the statement is true for all subexpressions of some E such that $F \xrightarrow{*} E$ then it is also true for all subexpressions of E' such that $E \longrightarrow E'$. We prove the last by induction on the transition derivation $E \longrightarrow E'$. The applicable derivation rules from Table 3.5 are the rules (1)-(3), (10)-(12), (9) and (16). In the rules (1)-(3) and (10)-(12) the right side of the transition is a subexpression of the left side, and the statement follows from the induction hypothesis for all subexpressions of E. For the rules (9) and (16) we need to prove that the statement about different action names decorating enabled transitions holds for all subexpressions of $E'''\{\underline{X := E'''}/X\}$ with E''' being an IMC$^{\mathrm{G}}$ expression if it holds for all subexpressions of $\underline{X := E'''}$. This follows however from the induction hypothesis and from Lemma 3.6 which states that exposed labels before and after substitutions are the same. □

Theorem 5.15 below contains the last statement to prove in this section. It will assemble all the proof parts which have been shown in the lemmas above into the proof that our algorithm for *etsets* correctly computes the minimum expected reachability time over all simple schedulers.

Theorem 5.15 (Minimum expected time reachability)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F *such that* $\vdash^{tr} F$ *holds,* $\mathcal{F}$ *as in Definition 4.9,* $F \xrightarrow{*} E$, $L_1 = dom(\mathcal{E}_\Gamma[\![E]\!])$ *and* $L_2 \subseteq Labs(F)$*, then the minimum expected time to reach an element in* $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ *from* E *under the class of simple*

schedulers is equal to etsets(L_1, L_2)*. Thereby etsets*$(L_1, L_2) = \infty$ *if and only if* $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\} = \emptyset$.

Proof. The statement of this theorem is directly derivable from the previously proved Lemmas 5.13 and 5.14. From Lemma 5.13 follows that $etsets(L_1, L_2) = \infty$ if and only if $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\} = \emptyset$. Otherwise there exists a transition sequence $E \longrightarrow ... \longrightarrow E'$ with $E' \in \{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ with the expected duration of the transition sequence in the sense of Definition 5.12 equal to $etsets(L_1, L_2)$ and minimal out of the expected durations of all the other transition sequences from E to elements of $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$.

It follows from Lemma 5.14 that on the one hand, for each transition sequence from E to an element in $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ there exists a simple scheduler inducing it (i.e. it is taken with the probability 1), and on the other hand, each simple scheduler gives rise to exactly one transition sequence from E to a chosen element from $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$.

Altogether, according to Lemma 5.13 we calculate in the function *etsets* the tight lower bound for the expected duration of all transition sequences from E to E', and according to Lemma 5.14 these are exactly transition sequences that are executed with the probability 1 under simple schedulers. Therefore *etsets* computes the minimum expected time to reach any of the elements in $\{E''|(E \xrightarrow{*} E'') \wedge (L_2 \subseteq dom(\mathcal{E}_\Gamma[\![E'']\!]))\}$ from E. □

We will make a short conclusion for this section. We have presented an algorithm for computing the minimum expected reachability time under simple schedulers and have proved its correctness. As input parameters we expect two label sets representing exposed labels of IMC^{G} expressions, both derivable from an initial IMC^{G} program F for which $\vdash^{tr} F$ holds. The rules for $\vdash^{tr}$ allow only linear IMC^{G} programs and the choice only between external actions. The goal set of labels can be a strict subset of exposed labels of the states the minimum expected reachability time to which is computed.

Our algorithm can be potentially extended to a less restrictive syntax. We could for example allow delay rates in the choice operator. It is not a problem with maximal one delay in the choice operator, but in case of more than one delay we would need to sum the results of computations on several paths multiplied with

their probabilities and not only to determine one shortest path in the sense of its expected duration. Introducing internalisation operator and allowing several actions with the same name in the choice operator would lead to a situation where our algorithm will still compute a lower bound for the minimum expected reachability time but this bound will possibly be not tight due to internal non-determinism.

Chapter 6

Conclusions

In this dissertation we have investigated the applicability of Static Analysis methods to stochastic process calculi. Our main thesis was that

> Static Analysis techniques can be used for verification in a syntax-directed and compositional way of systems modelled by stochastic process calculi.

Stochastic process calculi are widely accepted formalisms for describing concurrent stochastic systems. The syntax of process calculi is suited to describe systems compositionally, building larger systems out of smaller ones, with explicit indication of interactions and communication channels. Stochastic process calculi allow to model not only functional aspects of behaviour, but also incorporate the notions of time and probabilities. In this dissertation we have studied the calculus of Interactive Markov Chains (IMC) [BH00].

Model-checking methods can be applied to concurrent stochastic systems in order to verify whether they have required properties, both qualitative (safety, liveness) and quantitative (performance) ones [HKMKS00], [BHHK03], [BHKH05], [NZ10]. Model checking is however mostly done on expanded system models, after explicit resolution of all interdependencies and all possible interactions between subsystems. This means that the advantages of compositional modelling

(concise system description, compositionality) are not sufficiently exploited at the verification stage.

In the dissertation we have been developing alternative methods, that can verify concurrent stochastic systems in a compositional way, working on their compositional syntactic specification. We have shown that system specifications can be analysed by means of Static Analysis methods and have explored which problems can be handled in this way. In particular, we have extended the Static Analysis of IMC in order to identify bisimilar states, reachable states and to compute the expected timed reachability – the last for a linear fragment of IMC. The methods deliver in general approximate results but the precision should be enough for many practical purposes. Moreover, we have developed an algorithm for refining computed reachability sets. Static Analysis methods can thus exploit the advantages of compositionality in the syntax and they are moreover adaptable to the required precision level.

Syntax-driven verification of concurrent stochastic systems has similarities with the methods of partial order reduction [GKPP95] and symmetry reduction [CJEF96] that are also exploiting the modularity in concurrent systems in order to reduce their state space. The difference is that in our case we can apply different post-processing methods after conducting Static Analysis and thus can exploit several ways of reducing the complexity of computations simultaneously.

6.1 Main results

We will shortly characterise the main results of this dissertation and the conclusions that can be made based on these results.

Pathway Analysis is applicable to stochastic process calculi.

In this dissertation we have transferred Static Analysis methods from functional to stochastic process calculi, in particular, we have applied Data Flow / Pathway Analysis methods to the calculus of IMC [BH00]. We have defined the calculus of IMC^{G} that on one hand has a slightly more permissive syntax then IMC and on the other hand has "guardedness" conditions built into the syntax. In order to deal only with those IMC^{G} systems that have finite semantic models and are easy to analyse with Static Analysis methods, additional well-formedness conditions have been devised that should be checked inductively on the syntax of an IMC^{G} expression in question. The well-formedness conditions have been defined with a similar purpose as the well-formedness conditions for BioAmbients in [Pil07] – in order to exclude syntactic expressions that are

hard to analyse with Static Analysis methods – but have been strongly revised.

A number of usual Data Flow operators previously applied to, for example, CCS in [NN07] and BioAmbients [Pil07], i.e. *exposed*, *generate* and *kill* operators, have been redefined so that they become applicable to IMC^{G}. A new operator (*chains* operator) has been introduced in order to analyse synchronisation constructs in IMC^{G}. Action names and delay rates in the syntax of IMC^{G} have been treated similarly on the level of transfer functions (see Section 3.4), with only slight differences. The differences between actions and delays in the semantics of IMC^{G} influence however the way in which Pathway Analysis results are post-processed: in many cases delay-transitions are much harder to deal with than action transitions while checking properties.

We do not make use of a granularity function as in the previous work (see [NN07], [Pil07]) because semantic models of well-formed IMC^{G} systems, as well as semantic models of IMC systems, are always finite. Therefore it is not necessary to merge states with a granularity function in the Worklist Algorithm which constructs a labelled transition system (see Section 3.6) in order to guarantee the termination of the Worklist Algorithm. It becomes possible to check properties "on the fly", during the construction of the labelled transition system, because states are not merged and not revisited by the Worklist Algorithm.

Pathway Analysis can be precise. Data Flow or its subtype Pathway Analysis have been successfully applied until now to a number of process calculi – CCS [NN07, NN09], BioAmbients [NNPR04, Pil07], bKlaim [NNN08]. All these calculi allow infinite syntactic models. Based on the analysis results, a finite automaton was constructed by the Worklist Automaton that could simulate the semantics of an analysed system. The merging of states was necessary at this final stage of the analysis in case the system semantics was infinite. A granularity function has been used for deciding which states should be merged. Therefore possible imprecision (in the sense of over-approximation) of transfer functions computed by Data Flow Analysis operators on the syntax of the analysed system was acceptable in view of the merging of states at the final stage. The question of whether the analysis is precise has not been explored for these calculi.

In this dissertation Pathway Analysis methods have been adopted to systems with finite semantic models, and we have proved that transfer functions are computed without loss of precision for uniquely labelled, closed and well-formed IMC^{G} expressions that we have named IMC^{G} *programs*. The computed transfer functions and several additional operators on the syntax of IMC^{G} "capture" the semantics of IMC^{G} programs in a precise way.

Pathway Analysis can be used for computing bisimulation and reachability relations. We have devised in Chapter 5 two bisimulation relations

that can be computed by post-processing of the Pathway Analysis results of IMC^{G} programs. They can be computed in the low polynomial time and space complexities in the size of the syntax specification of an analysed system. The complexity gain is due to the fact that part of the decision whether two states are bisimilar is taken on the level of labels. It is first determined which of the labels that occur in the syntactic specification are "bisimilar". Additionally a kind of "synchronisation structure" of two states to be compared is computed. Two states are contained in a defined by us bisimulation relation if their "active labels" (decorating outgoing transitions from that states) are pairwise bisimilar and they have the same "synchronisation structure". We apply calculated bisimulation relations to an IMC^{G} fragment without delays as dealing with delays in a compositional way is much harder. Also for this fragment we do not build the coarsest bisimulations, but we conjecture that the devised by us bisimulation relations are the coarsest bisimulations for important subclasses of IMC^{G}.

In Section 5.3 we have devise an algorithm for computing reachable states represented by their "active labels". Reachability on the level of labels is computed as a transitive closure of the generate operator which is one of the operators of Pathway Analysis. After one semantic step a set of reachable labels can be refined. We could also determine by examining beforehand the syntax of an analysed system that some combinations of active labels are impossible, i.e. are unreachable, and in this way reduce the state space of the analysed system (see Section 5.1). These relations can be computed in the low polynomial time and space complexities in the size of the syntax specification. On the other hand, we compute an over-approximation of reachable labels/states in the first case and an under-approximation of unreachable states in the second case.

Pathway Analysis can be used for computing timed reachability. We have developed in Section 5.4 an algorithm for computing the minimum expected time to reach a fixed set of goal states from an initial state. The algorithm is applicable to a linear fragment of IMC^{G} under the class of simple schedulers. It uses the results of the previously conducted Pathway Analysis on the syntactic specification of a linear IMC^{G} system and has the low polynomial time and space complexities in the size of the specification. The algorithm is exploiting the fact that the expected value of a sum of random variables is equal to the sum of expected values of the variables [GS97]. Therefore the expected duration of a sequence of transitions can be computed "piecewise" and then summed up. This result shows that Static Analysis methods can be also used for computing quantitative information.

The purpose of the methods developed in this dissertation was to check both qualitative and quantitative properties whenever possible directly on a system specification, i.e. first on subsystems specifications and then to combine the

results using information on interactions between them that could be deduced from the specification. As the work is based on Static Analysis methods, in general safe approximations of properties (under-approximations of bisimilarity relations, over-approximations of sets of reachable states) have been computed. The computations could be done however in the low polynomial time and space complexities in the size of syntactic specifications. We can therefore make a conclusion that the main thesis of this dissertation holds.

6.2 Future work

During the work on this dissertation we have identified a number of possible extensions of the developed techniques and a number of research directions that can continue this line of work. We will list most of these ideas below.

Expected time reachability for full linear IMC$^{\mathbf{G}}$ and for non-linear fragments of IMC$^{\mathbf{G}}$ The algorithm from Section 5.4 computes minimum expected time reachability for a linear fragment of IMC$^{\mathrm{G}}$. It can be extended to the full linear IMC$^{\mathrm{G}}$ and to eventually maximum expected time (for states reachable with the probability 1). We would also like to extend the algorithm from the linear case to several processes running in parallel. This is however much harder than dealing with the linear case due to complex race conditions that are possible in parallel processes. Due to the hardness of the problem, we will aim to compute upper and lower bounds rather than actual values in order to preserve low complexities of computations. For resolving nondeterminism we will consider first of all the class of memoryless schedulers as they are most suitable for compositional verification and are used in many real-life systems. This line of work can be seen in a broader perspective as developing syntax-based quantitative Static Analysis techniques.

Computing larger bisimulation relations The algorithms from Chapter 4 can be enhanced in order to be able to identify broader classes of strongly bisimilar IMC$^{\mathrm{G}}$ expressions. We would also like to clarify the question for which subclasses of IMC$^{\mathrm{G}}$ our existing algorithms from Chapter 4 can identify *all* strongly bisimilar states. Moreover, we could possibly adapt the algorithms to computing, besides strong bisimulations, also weak bisimulations or trace equivalences, eventually even probabilistic bisimulations.

Enhancing well-formedness conditions We would like to explore the possibilities of using syntactic conditions similar to the well-formedness conditions in Chapter 3 in order either to exclude beforehand, without conducting the analysis, IMC$^{\mathrm{G}}$ systems with undesirable properties or detect the ones with

especially favourable properties. We could, for example, exclude beforehand systems where deadlocks are possible or those with Zeno cycles.

It is clear that while computing complex properties for the whole IMC^{G} we can only "over-approximate" the undesirable IMC^{G}s and "under-approximate" the desirable ones. This means that in the first case we will exclude all IMC^{G}s that have undesirable properties but also some that do not have such properties; in the second case all IMC^{G}s that we will detect will have the desirable properties but also some of those that we would reject will also have these properties. We could either accept the fact that the results are imprecise or strive to identify subclasses of IMC^{G} on which the algorithms deliver more precise results than on IMC^{G} in general.

Broader class of properties We can broaden the class of problems that could be tackled by using methods similar to the ones developed in this dissertation. As the class of problems is usually identified by a temporal logic, we could explore which logics can be model-checked with our methods. We would use 3-valued logics as Static Analysis methods in general approximate the actual values, returning upper and lower bounds. We can base our research on the evaluation of formulas in the 3-valued ACTL logic over modal transition systems constructed with the Data Flow Analysis of bKlaim in [NNN08]. The temporal logics that we could consider are subclasses of the 3-valued PCTL logic [FLW] and the 3-valued CSL logic [KKLW07].

Merging of states Worklist Algorithm in Section 3.6 builds a labelled transition system that can be model-checked afterwards. However due to the previously conducted Static Analysis of the syntactic specification of a system we know the "internal structure" of states. We can therefore merge "on the fly" either bisimilar states or states whose differences in the behaviour are not particularly important for the properties that we plan to check on the constructed transition system. The first possibility is to abstract away from state properties which are not relevant for answering verification questions and thus to merge states which differ only in their "irrelevant" behaviour. The second possibility is to assess differences between states based on the Static Analysis results and then merge "similar" states together.

Partial construction of state space Another interesting question is how to increase the precision of verification results if necessary. In Section 5.3.2 we have discussed an algorithm for updating a set of reachable labels after one semantic transition. We could develop this approach further into an algorithm for gradual refinement of verification results. The idea is to start with an approximate algorithm applied to the initial state of a system and then, in case the precision of the verification results will be insufficient, to build all the states reachable from the initial state after maximum a fixed number of transitions, run the

verification algorithm on the newly created states and then combine the returned verification results together. We will need to explore the questions how to assess the precision level of the result and how to combine the verification answers in the optimal way. The verification will thus deliver in general approximate results (e.g. upper and lower bounds for quantitative properties), but with the possibility to increase the precision in the next verification round.

Other process calculi We can extend the Pathway Analysis from IMC^{G} to other process calculi. For example, to calculi with different from IMC synchronisation models such as PEPA [Hil96] and Bio-PEPA [CH09] or to calculi with delays different from exponential such as SPADES [HS00]. Recent advances in the field of stochastic process calculi include the creation of uniform frameworks – in particular, Rated Transition Systems [KS08] and Rate Transition Systems [NLLM09a] (the last have been recently generalised to Function Transition Systems). By working on a uniform framework algorithms can be developed that are applicable to all stochastic process calculi definable in that framework – both already studied ones and potentially derivable from that uniform framework.

Implementation We plan to build a tool based on the obtained theoretical results. The implementation can be based on the already implemented implementations of Data Flow / Pathway Analysis for CCS [NN07], BioAmbients [Pil07] and bKlaim [NNN07].

Infinite semantic models We can extend the Pathway Analysis of IMC^{G} to infinite semantic models if we adopt a more flexible approach in the definition of transfer functions. We have assumed in this dissertation that generate, kill and chains operators results should be applicable after any number of semantics steps, i.e. they should not change (see Section 3.4). In the future work we could allow their results to change in a predictable way – for example, after certain transitions, – and correctly model a larger class of systems with process duplication. It will be interesting to try different subclasses of infinite systems and to compare the achievable precision on them. It is clear that we would need a calculus with infinite semantic models on which we could try out this "changeable" version of Pathway Analysis.

We can make a conclusion that methods developed in this dissertation are extendable to a broader class of properties, broader class of systems, and their precision is to some extend adaptable to the nature of verification problems and the required precision level.

Appendix A

Proofs from Chapter 3

Lemma 3.1 (Preservation of IMC$^{\mathrm{G}}$ syntax). Given a closed IMC$^{\mathrm{G}}$ expression E and $E \longrightarrow E'$, then E' is also a closed IMC$^{\mathrm{G}}$ expression.

Proof. We can prove the statement by induction on the transition derivation according to the SOS rules in Table 3.5. The rules (1) and (10) are the base cases and the statement is clear for them because the right side of the transition cannot be a variable due to process identifier closeness of E. The rest of the rules follow from the induction hypothesis. For the rules (9) and (16) we have to additionally show that $E\{\underline{X := E}/X\}$ is a closed IMC$^{\mathrm{G}}$ expression if this holds for $\underline{X := E}$. Process identifier closeness for $E\{\underline{X := E}/X\}$ follows from the fact that the only free process variable in E can be X, therefore no free process identifier variables are present in $E\{\underline{X := E}/X\}$ after the substitution. The second statement follows from the fact that if $\underline{X := E}$ is an IMC$^{\mathrm{G}}$ expression then also E is an IMC$^{\mathrm{G}}$ expression. From the rules (1)-(4) in Table 3.1 follows that we can use an IMC$^{\mathrm{G}}$ expression instead of a variable – the result will still be a valid IMC$^{\mathrm{G}}$ expression. □

Lemma 3.2 (Transition of process definition). Given a closed $\mathrm{IMC}^{\mathrm{G}}$ expression $\underline{X := E}$, then $\underline{X := E} \xrightarrow[C]{\alpha} E'$ if and only if there exists E'' such that $E \xrightarrow[C]{\alpha} E''$ and $E' = E''\{\underline{X := E}/X\}$.

Proof. We prove this by induction on the transition derivation according to the SOS rules in Table 3.5. Note that the existence of the transition $\underline{X := E} \xrightarrow[C]{\alpha} E'$ is equivalent to the existence of the transition $E\{\underline{X := E}/X\} \xrightarrow[C]{\alpha} E'$ due to the rules (9) and (16) in Table 3.5. We will prove that for each $\mathrm{IMC}^{\mathrm{G}}$ expression E, for each transition $E\{G/X\} \xrightarrow[C]{\alpha} E'$, where every process identifier which is not free in E is also not free in G, there exists a transition $E \xrightarrow[C]{\alpha} E''$ for some E'' such that $E' = E''\{G/X\}$, and the other way round. This will prove the lemma's statement as a particular case.

Let us note that in case X is not free in E then the statement is obviously true because X is not free in E' as well and we can set $E' = E''$. If X is free in E then we will prove the statement of the lemma by induction on the transition derivation of $E\{G/X\} \xrightarrow[C]{\alpha} E'$.

The base cases are rules (1) and (10) from Table 3.5 and the statement is clear for them. The rest of the rules can be easily derived from the induction hypothesis besides the rules (9) and (16). Note that in case of the synchronisation operator we may have one of the synchronisation sides unchanged (rules (4)-(5) and (13)-(14)). For the unchanged synchronisation side we may simply use the expression without substitution on the right-hand side of the transition. For example, for the rule (4) we can derive from the induction hypothesis that $E\{G/X\} \xrightarrow[C]{\alpha} E'$ iff $E \xrightarrow[C]{\alpha} E''$ for some E'' such that $E' = E''\{G/X\}$ and therefore $E\{G/X\} \,{}_{\shortparallel}\, A \,{}_{\shortparallel}\, F\{G/X\} \xrightarrow[C]{\alpha} E' \,{}_{\shortparallel}\, A \,{}_{\shortparallel}\, F\{G/X\}$ iff $E \,{}_{\shortparallel}\, A \,{}_{\shortparallel}\, F \xrightarrow[C]{\alpha} E'' \,{}_{\shortparallel}\, A \,{}_{\shortparallel}\, F$ with $E''\{G/X\} \,{}_{\shortparallel A\shortparallel}\, F\{G/X\} = E' \,{}_{\shortparallel A\shortparallel}\, F\{G/X\}$.

For the rules (9) and (16), if X is free in the initial expression then there exists a transition $\underline{Y := E\{G/X\}} \xrightarrow[C]{\alpha} E'$. This is equivalent to the existence of the transition $E\{G/X\}\{E/Y\{G/X\}\} \xrightarrow[C]{\alpha} E'$. This is however the same as the transition $E\{E/Y\}\{G/X\} \xrightarrow[C]{\alpha} E'$, because Y is not free in G as it is not free in $\underline{Y := E}$. We can apply the induction hypothesis to this transition because the transition derivation tree is smaller.

We derive from the induction hypothesis that $E\{E/Y\}\{G/X\} \xrightarrow[C]{\alpha} E'$ iff there

exists E'' such that $E\{E/Y\} \xrightarrow[C]{\alpha} E''$ and $E' = E''\{G/X\}$. The last transition is however according to the rules (9) and (16) in Table 3.5 equivalent to the transition $\underline{Y := E} \xrightarrow[C]{\alpha} E''$ with $E' = E''\{G/X\}$ and this is exactly what we had to prove. □

Lemma 3.3 (Preservation of well-formedness). Given an $\mathrm{IMC}^{\mathrm{G}}$ expression E such that $\vdash_{\mathbf{fn}(E)} E$ holds and $Labs(E_1) \cap Labs(E_2) = \emptyset$ for all $E_1 \,{}_{\shortparallel}A_{\shortparallel}\, E_2 \preceq E$, then for all E' such that $E \xrightarrow[C]{\alpha} E'$ also $\vdash_{\mathbf{fn}(E')} E'$ holds and $Labs(E_1) \cap Labs(E_2) = \emptyset$ for all $E_1 \,{}_{\shortparallel}A_{\shortparallel}\, E_2 \preceq E'$.

Proof. Note that we can prove for simplicity that $\vdash_{\mathbf{fn}(E)} E'$ holds. This will also prove that $\vdash_{\mathbf{fn}(E')} E'$ holds because, as it is easy to deduce, $\mathbf{fn}(E') \subseteq \mathbf{fn}(E)$.

We prove the lemma by the induction on the transition derivation with the well-formedness parameter S being any considerable set of actions: we have to assume this because if E is well-formed relative to some action set S then any its subexpression is also well-formed relative to this set.

The base cases are rules (1) and (10) in Table 3.5: the statement about well-formedness follows from the well-formedness rules (1)-(4) in Table 3.8 and the statement about disjoint labeling is obvious. The rules concerning the choice and hide operators follow by induction. The rules about the parallel operator are a bit more complicated. For example, the statement concerning disjoint labeling of E' and F' in the rule (6) in Table 3.5 follows from the disjoint labeling of E and F and from the obvious fact that if $E \xrightarrow[C]{\alpha} E'$ then $Labels(E') \subseteq Labels(E)$. For the well-formedness of $E' \,{}_{\shortparallel}A_{\shortparallel}\, F'$ we can apply our induction hypothesis not to the fixed at the beginning set S but to the sets $S \cup \mathbf{fn}(E)$ and $S \cup \mathbf{fn}(F)$. We know that E is well-formed relative to the first set and F is well-formed relative to the second set. From the induction hypothesis follows that E' will be well-formed relative to $S \cup \mathbf{fn}(E)$ and F' will be well-formed relative to $S \cup \mathbf{fn}(F)$. From $\mathbf{fn}(E') \subseteq \mathbf{fn}(E)$ and $\mathbf{fn}(F') \subseteq \mathbf{fn}(F)$ follows that E' is well-formed relative to $S \cup \mathbf{fn}(E')$ and F' is well-formed relative to $S \cup \mathbf{fn}(F')$. From the rule (7) in Table 3.8 follows $\vdash_S E' \,{}_{\shortparallel}A_{\shortparallel}\, F'$.

For proving the lemma for the rules (9) and (16) in Table 3.5 we need to show that the lemma's conditions hold for $E\{\underline{X := E}/X\}$ if they hold both for E and $\underline{X := E}$. Labels of synchronising processes are disjoint in $E\{\underline{X := E}/X\}$ because this is the case for both E and $\underline{X := E}$ and no substitution is conducted

in the synchronising processes as E is well-formed and X cannot be free in a synchronising process due to its process identifier closeness, according to the rule (7) of well-formedness in Table 3.8. It is left to prove that $E\{\underline{X := E}/X\}$ is well-formed relative to some action set S if this is the case for both E and $\underline{X := E}$.

In order to prove this fact by induction on the syntactic structure of E we will show that for any $\mathrm{IMC}^{\mathrm{G}}$ expression E'' the expression $E''\{\underline{X := E}/X\}$ is well-formed relative to some action set S if this is the case for both E'' and $\underline{X := E}$ (no transition is derivable for $E'' \in \mathbf{Var}$ in case $E'' \neq X$, so we can omit this case). We prove this by induction on the structure of E''. Base cases are the rules (1)-(2) from Table 3.1 and the statement follows from the well-formedness rules (3)-(4) in Table 3.8. The rules (3)-(6) and (8) follow from the induction hypothesis and the rules for well-formedness in Table 3.8. The rule (7) follows as E'' is well-formed, therefore both synchronising sides are closed and no substitutions can be made in them. □

Lemma 3.5 (Exposed labels of $\mathrm{IMC}^{\mathrm{G}}$ expressions). Given an $\mathrm{IMC}^{\mathrm{G}}$ expression E, then $\mathcal{E}_{\emptyset}[\![E]\!] = \mathcal{E}_{\Gamma}[\![E]\!]$ holds for any environment $\Gamma \in 2^{\mathbf{Var}\times\mathfrak{M}}$.

Proof. We can prove the statement by induction on the structure of E following the rules from Table 3.1. Base cases are rules (1)-(4) and (9) from Table 3.1 and the statement easily follows for them from the rules (1)-(4) and (9) for the exposed operator in Table 3.9. The rest of the rules follows from the induction hypothesis and the rules for the exposed operator. □

Lemma 3.6 (Exposed labels under substitution). Given an $\mathrm{IMC}^{\mathrm{G}}$ expression E'', then $\mathcal{E}_{\Gamma}[\![E'']\!] = \mathcal{E}_{\Gamma}[\![E''\{\underline{X := E'}/X\}]\!]$ holds for all environments Γ and all $\mathrm{IMC}^{\mathrm{G}}$ expressions $\underline{X := E'}$. If $E'' = X$ then, provided that $(X, \mathcal{E}_{\emptyset}[\![E']\!])$ is a unique mapping for a variable X in Γ, i.e. for all $(X, M) \in \Gamma$ it holds that $M = \mathcal{E}_{\emptyset}[\![E']\!]$, then $\mathcal{E}_{\Gamma}[\![E'']\!] = \mathcal{E}_{\Gamma}[\![E''\{\underline{X := E'}/X\}]\!]$ holds as well.

Proof. We can prove the statement by induction on the structure of E''. If E'' is a variable then in case a substitution has taken place we use the fact that $(X, \mathcal{E}_{\emptyset}[\![E']\!])$ is the only definition of X in Γ. From Lemma 3.5 follows $\mathcal{E}_{\emptyset}[\![E']\!]{=}\mathcal{E}_{\Gamma}[\![E']\!]$ and the statement follows from the rule (8) for the exposed

operator in Table 3.9. The statement is obvious if no substitution has taken place.

If $E'' \notin \mathbf{Var}$ then E'' is an IMC$^{\mathrm{G}}$ expression and we can easily prove the lemma by induction on its syntactic structure following the rules from Table 3.1, using the rules for the exposed operator from Table 3.9. □

Lemma 3.7 (Variable definitions under transitions). Given a closed IMC$^{\mathrm{G}}$ expression E and a transition $E \xrightarrow[C]{\alpha} E'$, then $\Gamma_{\mathbf{Var}}[\![E']\!] \subseteq \Gamma_{\mathbf{Var}}[\![E]\!]$ holds.

Proof. We prove this lemma by induction on the transition derivation according to the SOS rules in Table 3.5. The base cases are the semantic rules (1) and (10) in Table 3.5 and the statement follows for them because $\Gamma_{\mathbf{Var}}[\![E']\!] = \Gamma_{\mathbf{Var}}[\![E]\!]$. The rest of the rules easily follows from the induction hypothesis and the rules of the operator $\Gamma_{\mathbf{Var}}$ in Table 3.10 besides the rules (9) and (16). We can however prove that $\Gamma_{\mathbf{Var}}[\![E\{\underline{X := E}/X\}]\!] \subseteq \Gamma_{\mathbf{Var}}[\![\underline{X := E}]\!]$: the reason is that for any $\underline{Y := F\{\underline{X := E}/X\}} \preceq E\{\underline{X := E}/X\}$ holds $\mathcal{E}_\emptyset[\![F]\!] = \mathcal{E}_\emptyset[\![F\{\underline{X := E}/X\}]\!]$ due to Lemma 3.6. □

Lemma 3.8 (Generate and kill operators under substitution). Given well-formed IMC$^{\mathrm{G}}$ expressions E' and $\underline{X := E}$, $\Gamma \in 2^{\mathbf{Var} \times \mathfrak{M}}$, $(X, \mathcal{E}_\emptyset[\![E]\!])$ a unique mapping for X in Γ, then $\mathcal{G}_\Gamma[\![E'\{\underline{X := E}/X\}]\!] \subseteq \mathcal{G}_\Gamma[\![E']\!] \cup \mathcal{G}_\Gamma[\![\underline{X := E}]\!]$ and $\mathcal{K}[\![E'\{\underline{X := E}/X\}]\!] \subseteq \mathcal{K}[\![E']\!] \cup \mathcal{K}[\![\underline{X := E}]\!]$ hold.

Proof. We prove this by induction on the structure of E'. The base cases are rules (1)-(2) from Table 3.1 and the statement is obvious in case no substitution has been made. If the substitution has taken place the statement follows from $(X, \mathcal{E}_\emptyset[\![E]\!])$ being a unique mapping for X in Γ, $\mathcal{E}_\Gamma[\![E]\!] = \mathcal{E}_\emptyset[\![E]\!]$ (see Lemma 3.5) for the generate operator and the rules (1)-(4) in Tables 3.11 for the generate operator, the rules (1)-(4) in Tables 3.13 and 3.14 for the kill operator.

For the rules (3)-(4) from Table 3.1 the statement follows from the induction hypothesis, the rules (3)-(4) in Tables 3.11, 3.13 and 3.14 and the consideration for the generate operator that $\mathcal{E}_\Gamma[\![P]\!] = \mathcal{E}_\Gamma[\![P\{\underline{X := E}/X\}]\!]$ for $P \notin \mathbf{Var}$ (see Lemma 3.6). For the rules (5)-(7) from Table 3.1 the statement follows from the induction hypothesis and the rules (5)-(7) in Tables 3.11, 3.13 and 3.14.

For the rule (8) from Table 3.1 in case we have a process definition for some process variable $Y \neq X$ then the statement of lemma follows from the induction hypothesis and the rule (8) in Tables 3.11, 3.13 and 3.14. If $Y = X$ then no substitution takes place and the statement obviously holds. □

Lemma 3.9 (Chain operator under substitution). Given well-formed IMC^{G} expressions E and E', then $\mathfrak{T}_\Lambda[\![E\{E'/X\}]\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!] \cup \mathfrak{T}_\Lambda[\![E']\!]$ holds for all $\Lambda \in 2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$.

Proof. We prove this by induction on the syntactic structure of E according to the rules in Table 3.1. The rules (1)-(2) in Table 3.1 are the base cases and follow from the rules (1)-(4) for the chains operator in Table 3.17. The rest of the rules follow by induction besides the rule (7): we need to use the well-formedness of E for it. The statement follows because any well-formed expression does not have free process variables in its synchronising processes, therefore no substitutions are made and $\mathfrak{T}_\Lambda[\![(E\{E'/X\}) \,{}_{\shortparallel}A_{\shortparallel}\, (F\{E'/X\})]\!] = \mathfrak{T}_\Lambda[\![E \,{}_{\shortparallel}A_{\shortparallel}\, F]\!]$. □

Lemma 3.10 (Generate operator under transitions). Given an IMC^{G} program F, $\Gamma = \Gamma_{\mathbf{Var}}[\![F]\!]$, $F \xrightarrow{*} E$ and $E \longrightarrow E'$, then $\mathcal{G}_\Gamma[\![E']\!] \subseteq \mathcal{G}_\Gamma[\![E]\!]$ holds.

Proof. We prove the lemma by induction on the transition derivation. The statement follows for the base cases – the rules (1) and (10) in Table 3.5 – from the rules (1)-(4) concerning prefixed expression for the generate operator in Table 3.11. The rest of the rules follows from the induction hypothesis and from the rules for the generate operator in Table 3.11. For the rules (9) and (16) we have to additionally use the fact shown in Lemma 3.8 that $\mathcal{G}_\Gamma[\![E\{\underline{X := E}/X\}]\!] \subseteq \mathcal{G}_\Gamma[\![E]\!] \cup \mathcal{G}_\Gamma[\![\underline{X := E}]\!]$ (the unique mapping for variables in $\Gamma_{\mathbf{Var}}[\![F]\!]$ is due to the fact that F is an IMC^{G} program). At the same time it is easy to see from the rules for the generate operator in Table 3.11 that $\mathcal{G}_\Gamma[\![\underline{X := E}]\!] = \mathcal{G}_\Gamma[\![E]\!] \subseteq \mathcal{G}_\Gamma[\![E\{\underline{X := E}/X\}]\!]$. Therefore $\mathcal{G}_\Gamma[\![E\{\underline{X := E}/X\}]\!] = \mathcal{G}_\Gamma[\![\underline{X := E}]\!]$ and the induction hypothesis is applicable.

□

Lemma 3.11 (Kill operator under transitions). Given an IMC^{G} program F, $\Gamma = \Gamma_{\mathbf{Var}}[\![F]\!]$, $F \xrightarrow{*} E$, $(\ell, M_1) \in \mathcal{K}[\![E]\!]$, $\bot_{\mathfrak{M}}[\ell \mapsto 1] \leq \mathcal{E}_\Gamma[\![E]\!]$, $(\ell, M_2) \in \mathcal{K}[\![F]\!]$, then $\mathcal{E}_\Gamma[\![E]\!] - M_1 = \mathcal{E}_\Gamma[\![E]\!] - M_2$ holds.

Proof. In order to be able to prove the lemma by induction on the structure of E we will also need to show that the (extended) statement of the lemma also holds for any subexpression of any IMC^{G} expression derivable from F, i.e. for $F \xrightarrow{*} E$ and $E' \preceq E'' \preceq E$, $\bot_{\mathfrak{M}}[\ell \mapsto 1] \leq \mathcal{E}_\Gamma[\![E']\!]$, holds: if $(\ell, M_1) \in \mathcal{K}[\![E']\!]$, $(\ell, M_2) \in \mathcal{K}[\![F]\!]$ and $(\ell, M_3) \in \mathcal{K}[\![E'']\!]$ then $\mathcal{E}_\Gamma[\![E']\!] - M_1 = \mathcal{E}_\Gamma[\![E']\!] - M_2 = \mathcal{E}_\Gamma[\![E']\!] - M_3$.

We prove the lemma by induction on the number of semantic steps in $F \xrightarrow{*} E$. For zero number of steps the statement holds for any subexpression of F due to its unique labeling and from the rules for the kill operator in Tables 3.13 and 3.14, in particular the rule (5) in Table 3.13: $M_3 \geq M_1$, but the difference exactly "covers" the exposed labels of the other summand. Otherwise we have to prove the induction step, namely, if the statement is true for E and $E \xrightarrow[C]{\alpha} E'$ then the statement is true for E'. We prove this by induction on the transition derivation.

The base cases are rules (1) and (10) from Table 3.5 and the statement is true because the right side of the transition is a subexpression of the left side. The statement follows from the induction hypothesis for the rules (2)-(3) and (11)-(12) in Table 3.5: for example, for the rule (2) the induction hypothesis is applicable to E because $E \preceq (E + F)$ and the statement holds for E' because the derivation tree for the transition $E \xrightarrow[C]{\alpha} E'$ is smaller than for the transition $(E + F) \xrightarrow[C]{\alpha} E'$.

We can deduce the statement from the induction hypothesis for the rules (4)-(6) and (13)-(14) from Table 3.5 because the labels of two synchronising processes are disjoint according to Lemma 3.3, therefore $\mathcal{E}_\Gamma[\![E']\!]$ and $\mathcal{E}_\Gamma[\![F']\!]$, $\mathcal{G}_\Gamma[\![E']\!]$ and $\mathcal{G}_\Gamma[\![F']\!]$ are label disjoint. The statement follows from the induction hypothesis for the rules (7)-(8) and (15) because $\mathcal{E}_\Gamma[\![\mathsf{hide}\ A\ \mathsf{in}\ E]\!] = \mathcal{E}_\Gamma[\![E]\!]$ and $\mathcal{K}[\![\mathsf{hide}\ A\ \mathsf{in}\ E]\!] = \mathcal{K}[\![E]\!]$ according to the rules for the corresponding operators.

For the rules (9) and (16) we have to prove that the induction hypothesis is applicable to $E\{\underline{X := E}/X\}$ if it is applicable to $\underline{X := E}$ (and therefore applicable to any subexpression of the last). It is clear that we only have to prove that the induction hypothesis is applicable for any $E''\{\underline{X := E}/X\}$ such that $E'' \preceq E$ and $E'' \notin \mathbf{Var}$. From Lemma 3.6 follows that $\mathcal{E}_\Gamma[\![E''\{\underline{X := E}/X\}]\!] = \mathcal{E}_\Gamma[\![E'']\!]$.

From Lemma 3.8 follows $\mathcal{K}[\![E''\{\underline{X := E}/X\}]\!] \subseteq \mathcal{K}[\![E'']\!] \cup \mathcal{K}[\![\underline{X := E}]\!]$. We can therefore apply the induction hypothesis for $\underline{X := E}$ to $E''\{\underline{X := E}/X\}$. □

Lemma 3.12 (Chains inclusion). Given a well-formed IMC$^{\mathrm{G}}$ expression F, $F \xrightarrow{*} E$, $\Lambda = \Lambda_{\mathbf{Lab}}[\![F]\!]$, then $\mathfrak{T}_\Lambda[\![E]\!] \subseteq \mathfrak{T}_\Lambda[\![F]\!]$ holds.

Proof. It is enough to prove that for any well-formed E from $E \xrightarrow[C]{\alpha} E'$ follows $\mathfrak{T}_\Lambda[\![E']\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!]$ for all $\Lambda \in 2^{\mathbf{Lab}\times(\mathbf{Act}\cup\{\tau\}\cup\mathbf{Rate})}$. We prove this by induction on the transition derivation.

Most of the rules from Table 3.5 are either easily provable base cases or follow by induction using the rules for the chain operator from Table 3.17. For the rules (9) and (16) we have to additionally use Lemma 3.9 in which we have proved that $\mathfrak{T}_\Lambda[\![E\{\underline{X := E}/X\}]\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!] \cup \mathfrak{T}_\Lambda[\![\underline{X := E}]\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$. Moreover from the proof of Lemma 3.3 follows that $E\{\underline{X := E}/X\}$ is well-formed. Our induction hypothesis is therefore applicable to it. □

Lemma 3.13 (Chains preservation). Given an IMC$^{\mathrm{G}}$ program F, $F \xrightarrow{*} E$, $\Lambda = \Lambda_{\mathbf{Lab}}[\![F]\!]$, $C \in \mathfrak{T}_\Lambda[\![F]\!]$, $dom(C) \subseteq Labs(E)$, then $C \in \mathfrak{T}_\Lambda[\![E]\!]$ holds.

Proof. We prove the lemma by showing that for any well-formed E, with $\Lambda_{\mathbf{Lab}}[\![E]\!] \subseteq \Lambda_{\mathbf{Lab}}[\![F]\!]$, for which the additional condition described below holds, if $E \xrightarrow[C]{\alpha} E'$ and for some $C \in \mathfrak{T}_\Lambda[\![E]\!]$ holds $dom(C) \subseteq Labs(E')$ then $C \in \mathfrak{T}_\Lambda[\![E']\!]$ and the mentioned condition holds for E'. This will prove the statement of the lemma (assuming the additional condition holds for F) because obviously from $E \xrightarrow[C]{\alpha} E'$ follows $Labs(E') \subseteq Labs(E)$. The additional condition is that for any $E' \preceq E'' \preceq E$ if for some $C \in \mathfrak{T}_\Lambda[\![E'']\!]$ holds $dom(C) \subseteq Labs(E')$ then $C \in \mathfrak{T}_\Lambda[\![E']\!]$. This is true for the initial IMC$^{\mathrm{G}}$ expression F due to its unique labeling.

We prove the statement by induction on the derivation of the transition $E \xrightarrow[C]{\alpha} E'$. For the rules (1) and (10) the statement follows from the induction hypothesis concerning any subexpression of E because the right-hand side of the

transition is a subexpression of E. For the rules (2)-(3) and (11)-(12) we can deduce the statement from the induction hypothesis.

For the rules (4)-(6) and (13)-(14) we can make use of the observation that the labels of two synchronising processes derived from some uniquely labelled and well-formed IMC$^{\mathrm{G}}$ expression are disjoint (see Lemma 3.3). Therefore, for example, for the rule (6) if $\mathfrak{T}_\Lambda[\![E \between A \between G]\!] \xrightarrow[C']{\alpha} \mathfrak{T}_\Lambda[\![E' \between A \between G']\!]$ and there is some $C \in \mathfrak{T}_\Lambda[\![E \between A \between G]\!]$ such that $dom(C) \subseteq Labs(E' \between A \between G')$, then we can uniquely identify $C_1 \in \mathfrak{T}_\Lambda[\![E]\!]$, $C_2 \in \mathfrak{T}_\Lambda[\![G]\!]$, such that $C = C_1 + C_2$, $dom(C_1) \subseteq Labs(E')$ and $dom(C_2) \subseteq Labs(G')$. From the induction hypothesis follows $C_1 \in \mathfrak{T}_\Lambda[\![E']\!]$ and $C_2 \in \mathfrak{T}_\Lambda[\![G']\!]$, therefore (as $\Lambda = \Lambda_{\mathbf{Lab}}[\![F]\!]$ and $\Lambda_{\mathbf{Lab}}[\![E' \between A \between G']\!] \subseteq \Lambda_{\mathbf{Lab}}[\![F]\!]$ can be easily shown) it also holds $C \in \mathfrak{T}_\Lambda[\![E' \between A \between G']\!]$. The additional condition holds on $E' \between A \between G'$ as well: it holds on all subexpressions of E' and G' due to the induction hypothesis and for the whole $E' \between A \between G'$ due to the disjoint labelling of E' and G'.

For the rules (9) and (16) we have to show that the induction hypothesis is applicable to $E\{\underline{X := E}/X\}$ if it is applicable to $\underline{X := E}$. We will see below that it will be enough to prove that for all $E'' \preceq \underline{X := E}$, with $X \in \mathbf{fpi}(E'')$, holds: $\mathfrak{T}_\Lambda[\![E'']\!] \subseteq \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$. This fact follows basically from the consideration that the parallel operator cannot be applied to E'' because $\mathbf{fpi}(E'') \neq \emptyset$ and the resulting expression will not be well-formed (see the rule (7) in Table 3.8). The rest of the syntactic constructs from Table 3.1 will preserve the chains from E'' in the resulting expression (see the rules for the chains operator in Table 3.17). We can deduce therefore that $\mathfrak{T}_\Lambda[\![E''\{\underline{X := E}/X\}]\!] \subseteq \mathfrak{T}_\Lambda[\![E'']\!] \cup \mathfrak{T}_\Lambda[\![\underline{X := E}]\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ if $X \in \mathbf{fpi}(E'')$ (the set inclusion follows from Lemma 3.9).

Now we are able to prove that the induction hypothesis is applicable to the result of substitution $E\{\underline{X := E}/X\}$ if it is applicable to $\underline{X := E}$. For $E' \preceq E'' \preceq \underline{X := E}$ the statement follows directly from the induction hypothesis for $\underline{X := E}$. For $E' \preceq \underline{X := E} \preceq E''$, if $E'' = E'''\{\underline{X := E}/X\}$, holds $\mathfrak{T}_\Lambda[\![E'']\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ and we know from the induction hypothesis that any chain from $\mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ all the labels in the domain of which are in $Labs(E')$ is also in $\mathfrak{T}_\Lambda[\![E']\!]$. For $\underline{X := E} \preceq E' \preceq E''$, with $E' = E'''\{\underline{X := E}/X\}$ and $E'' = E''''\{\underline{X := E}/X\}$, the statement follows because the sets $\mathfrak{T}_\Lambda[\![E']\!]$ and $\mathfrak{T}_\Lambda[\![E'']\!]$ are equal.

□

Lemma 3.14 (Transition existence). Given an IMC$^{\mathrm{G}}$ program F, $F \xrightarrow{*} E$, $\Lambda = \Lambda_{\mathbf{Lab}}[\![E]\!]$ and $\Gamma = \Gamma_{\mathbf{Var}}[\![E]\!]$, then $E \xrightarrow[C]{\alpha} E'$ holds if and only if $C \in \mathfrak{T}_\Lambda[\![E]\!]$ and $C \leq \mathcal{E}_\Gamma[\![E]\!]$, and one of the following cases occurs: $\alpha = Name^h_{\Lambda,\mathbf{fn}(E)}(C)$

and $\alpha \in \mathbf{Act} \cup \{\tau\}$ or $\alpha = Name^h_{\Lambda,\mathbf{fn}(E)}(C)$, $\alpha \in \mathbf{Rate}$ and there is no chain $C' \in \mathfrak{T}_\Lambda[\![E]\!]$, $C' \leq \mathcal{E}_\Gamma[\![E]\!]$ with $Name^h_{\Lambda,\mathbf{fn}(E)}(C') = \tau$.

Proof. Note that according to the rules for the chains operator in Table 3.17 any chain has in its domain only labels correspondent to one and the same action name or delay rate, therefore $Name^h_{\Lambda,\mathbf{fn}(E)}(C)$ is always equal to an action name or a delay rate if $C \in \mathfrak{T}_\Lambda[\![E]\!]$.

Another observation concerns the two cases mentioned in the statement of the lemma: the second case expresses that if the τ-action can be executed by some IMC$^{\mathrm{G}}$ expression then no delay can be executed by the same expression – this is clear from the SOS rules in Table 3.5. It is also easy to show that the transition is decorated by the action correspondent to the chain if that action has not been internalised and is decorated by τ if the corresponding action has been internalised or has been the τ-action from the beginning. It is therefore enough to prove that $E \underset{C}{\longrightarrow} E'$ iff $C \in \mathfrak{T}_\Lambda[\![E]\!]$ and $C \leq \mathcal{E}_\Gamma[\![E]\!]$.

We prove the lemma by induction on the syntactic structure of E. The statement is clear for the rules (1)-(4) in Table 3.1 due to the rules (1)-(4) for the exposed operator in Table 3.9, rules (1)-(4) for the chains operator in Table 3.17 and the semantic rules (1) and (10) in Table 3.5.

For the rule (5) we will need a consideration that if for some subexpression $(G + H) \preceq E'' \preceq E$ there exists $C \in \mathfrak{T}_\Lambda[\![E'']\!]$ such that $C \leq \mathcal{E}_\Gamma[\![G + H]\!]$ then either $C \leq \mathcal{E}_\Gamma[\![G]\!]$ or $C \leq \mathcal{E}_\Gamma[\![H]\!]$. This holds for the original expression F due to its unique labeling and the rule for the chains operator on a sum in Table 3.17. We need to show that this property is preservable under semantic transitions. This is clear for the base cases – semantic rules (1) and (10) in Table 3.5. The other rules follow by induction besides the rules (9) and (16) in Table 3.5. For them we need to show that if the relevant property holds for $\underline{X := E}$ then it also holds for $E\{\underline{X := E}/X\}$. This follows from Lemma 3.9 ($\mathfrak{T}_\Lambda[\![E'''\{\underline{X := E}/X\}]\!] = \mathfrak{T}_\Lambda[\![E''']\!] \cup \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ for well-formed E''') and Lemma 3.6 ($\mathcal{E}_\Gamma[\![H\{\underline{X := E}/X\}]\!] = \mathcal{E}_\Gamma[\![H]\!]$ and $\mathcal{E}_\Gamma[\![G\{\underline{X := E}/X\}]\!] = \mathcal{E}_\Gamma[\![G]\!]$ for any $(H + G) \preceq \underline{X := E}$).

Now we can also deduce the statement of the lemma for the sum operator from the induction hypothesis, using the consideration above. The rule (6) for the hide-operator easily follows from the induction hypothesis. For the parallel composition in the rule (7) we use $\Lambda = \Lambda_{\mathbf{Lab}}[\![E]\!]$ and the fact proved in Lemma 3.3 that labels of two synchronising processes are disjoint. We can therefore in case $C \in \mathfrak{T}_\Lambda[\![G \,{}_{\shortparallel A \shortparallel}\, H]\!]$ and $C \leq \mathcal{E}_\Gamma[\![G \,{}_{\shortparallel A \shortparallel}\, H]\!]$ uniquely determine C_1

and C_2, such that $C = C_1 + C_2$, $C_1 \in \mathfrak{T}_\Lambda[\![G]\!]$ and $C_1 \leq \mathcal{E}_\Gamma[\![G]\!]$, $C_2 \in \mathfrak{T}_\Lambda[\![H]\!]$ and $C_2 \leq \mathcal{E}_\Gamma[\![H]\!]$, and apply the induction hypothesis. On the other hand, if $G \parallel A \parallel H \xrightarrow[C]{\alpha} G' \parallel A \parallel H'$ then we can uniquely determine C_1 and C_2, such that $C = C_1 + C_2$, $G \xrightarrow[C_1]{\alpha} G'$ and $H \xrightarrow[C_2]{\alpha} H'$, and the induction hypothesis is applicable as well.

For the rule (8) we know from Lemma 3.2 that $\underline{X := E} \xrightarrow[C]{\alpha} E'$ iff $E \xrightarrow[C]{\alpha} E''$ for some E'' with $E' = E''\{\underline{X := E}/X\}$. From the induction hypothesis we know that $E \xrightarrow[C]{\alpha} E''$ iff $C \in \mathfrak{T}_\Lambda[\![E]\!]$ and $C \leq \mathcal{E}_\Gamma[\![E]\!]$. From the rules for the chains operator follows $\mathfrak{T}_\Lambda[\![E]\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ and from the rules for the exposed operator follows $\mathcal{E}_\Gamma[\![E]\!] = \mathcal{E}_\Gamma[\![\underline{X := E}]\!]$. This proves the statement of the lemma also for $\underline{X := E}$.

□

Lemma 3.15 (Unique exposed labels). Given an IMC$^{\mathrm{G}}$ program F, $F \xrightarrow{*} E$ and $\Gamma = \Gamma_{\mathbf{Var}}[\![F]\!]$, then either $\mathcal{E}_\Gamma[\![E]\!](\ell) = 0$ or $\mathcal{E}_\Gamma[\![E]\!](\ell) = 1$ holds for all $\ell \in \mathbf{Lab}$. Given two transitions $E \xrightarrow[C_1]{} E'$ and $E \xrightarrow[C_2]{} E''$ with different derivation trees, then $C_1 \neq C_2$ holds.

Proof. We will show that if for some well-formed IMC$^{\mathrm{G}}$ expression E for all $E'' \preceq E$ holds for all $\ell \in \mathbf{Lab}$ either $\mathcal{E}_\Gamma[\![E'']\!]\ell = 0$ or $\mathcal{E}_\Gamma[\![E'']\!]\ell = 1$ then the same holds for all E' such that $E \xrightarrow[C]{\alpha} E'$. This will be enough to prove the lemma because this obviously holds for F due to its unique labeling.

We prove this by induction on the transition derivation. Most of the rules in Table 3.5 are obvious. For the rules (5)-(7) and (13)-(14) we have to use the fact proved in Lemma 3.3 that the labels of two synchronising processes are disjoint. For the rules (9) and (16) we have to prove that the statement holds for all subexpressions of $E\{\underline{X := E}/X\}$ if it holds for all subexpressions of $\underline{X := E}$. It is clear for all $E'' \in \mathbf{Var}$ or $E'' \preceq \underline{X := E}$. Otherwise from Lemma 3.6 follows $\mathcal{E}_\Gamma[\![E''\{\underline{X := E}/X\}]\!] = \mathcal{E}_\Gamma[\![E'']\!]$.

The second statement of the lemma follows from the uniqueness of exposed labels (that are the only labels decorating the transition according to Lemma 3.14) by induction on a transition derivation tree. □

Theorem 3.16 (Pathway Analysis Exactness). Given an IMC$^{\mathrm{G}}$ program F, $F \xrightarrow{*} E$ and $E \xrightarrow[C]{\alpha} E'$, then $\mathcal{E}_\Gamma[\![E]\!] - \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{K}_{up}[\![E]\!]\} + \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{G}_\Gamma[\![E]\!]\} = \mathcal{E}_\Gamma[\![E']\!]$.

Proof. Note that from Lemma 3.14 follows that all the labels in the domain of the executable chain C are exposed. From the rules for the operator $\mathcal{K}_{up}$ in Table 3.13 we can deduce that it is defined on all the exposed labels. Moreover, it contains only one entry for all $\ell \in dom(\mathcal{E}_\Gamma[\![E]\!])$ – this follows from Lemma 3.15 on the uniqueness of the exposed labels for all derivative expressions of an IMC$^{\mathrm{G}}$ program. Also $\mathcal{G}_\Gamma[\![E]\!]$ contains only one entry for all $\ell \in dom(\mathcal{E}_\Gamma[\![E]\!])$ – this can be deduced from the unique labelling of F and from $\mathcal{G}_\Gamma[\![E]\!] \subseteq \mathcal{G}_\Gamma[\![F]\!]$ according to Lemma 3.10. We can also show that for all subexpressions $E'' \preceq E$ both $\mathcal{G}_\Gamma[\![E'']\!]$ and $\mathcal{K}_{up}[\![E'']\!]$ contain unique mappings for all $\ell \in dom(\mathcal{E}_\Gamma[\![E'']\!])$.

We prove the lemma by induction on the transition derivation. The rules (1) and (10) in Table 3.5 follow from the rules (1)-(4) for the generate and kill operators in Tables 3.11 and 3.13 and the rule (9) in Table 3.9 (applicable if the right side of the transition is a variable). The rules for the choice operator (2)-(3) and (11)-(12) follow from the induction hypothesis and the rule (5) for the $\mathcal{K}_{up}$ operator in Table 3.13: the exposed labels of the summand that does not participate in the transition are killed according to it. The rest of the rules directly follow from the induction hypothesis besides the rules (9) and (16).

For the last rules we can use the fact proved in Lemma 3.2 that if $\underline{X := E} \xrightarrow[C]{\alpha} E'$ then there also exists a transition $E \xrightarrow[C]{\alpha} E''$ for some E'' such that $E' = E''\{\underline{X := E}/X\}$. From the induction hypothesis follows therefore that $\mathcal{E}_\Gamma[\![E]\!] - \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{K}_{up}[\![E]\!]\} + \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{G}_\Gamma[\![E]\!]\} = \mathcal{E}_\Gamma[\![E'']\!]$ from the induction hypothesis. From $\mathcal{G}_\Gamma[\![\underline{X := E}]\!] = \mathcal{G}_\Gamma[\![E]\!]$, $\mathcal{K}_{up}[\![\underline{X := E}]\!] = \mathcal{K}_{up}[\![E]\!]$, $\mathcal{E}_\Gamma[\![E]\!] = \mathcal{E}_\Gamma[\![\underline{X := E}]\!]$ and $\mathcal{E}_\Gamma[\![E'']\!] = \mathcal{E}_\Gamma[\![E''\{\underline{X := E}/X\}]\!]$ (see Lemma 3.6) follows the statement $\mathcal{E}_\Gamma[\![\underline{X := E}]\!] - \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{K}_{up}[\![\underline{X := E}]\!]\} + \sum_{\ell \in dom(C)}\{M|(\ell, M) \in \mathcal{G}_\Gamma[\![\underline{X := E}]\!]\} = \mathcal{E}_\Gamma[\![E''\{\underline{X := E}/X\}]\!]$ as well. □

Theorem 3.17 (Worklist algorithm). Given an IMC$^{\mathrm{G}}$ program F, then the Worklist Algorithm terminates on F. The Workshop Algorithm creates a state M if and only if there exists an IMC$^{\mathrm{G}}$ expression E such that $F \xrightarrow{*} E$ and $M = \mathcal{E}_\Gamma[\![E]\!]$ with $\Gamma = \Gamma_{\mathbf{Var}}[\![F]\!]$. Moreover, from $F \xrightarrow{*} E_1$, $F \xrightarrow{*} E_2$, $M_1 = \mathcal{E}_\Gamma[\![E_1]\!]$ and $M_2 = \mathcal{E}_\Gamma[\![E_2]\!]$ follows that $E_1 \xrightarrow[C]{\alpha} E_2$ if and only if the Workshop Algorithm creates the transition $M_1 \xrightarrow[C]{\alpha} M_2$.

Proof. The Worklist algorithm terminates on F because the number of states that are introduced are maximally $2^{|Labs(F)|}$: according to Lemma 3.15 any label can have maximally one occurrence in the multiset of exposed labels of any expression derivable from an IMC$^{\mathrm{G}}$ program.

We can prove now the rest of the lemma's statements using the theoretical results proved before. We can prove that there is a state created by the Worklist algorithm for all E such that $F \xrightarrow{*} E$ by induction on the number of semantic transitions. It is clear for $E = F$. Otherwise if $M = \mathcal{E}_\Gamma[\![E]\!]$ has been added to the set of states by the Worklist algorithm, then for $E \longrightarrow E'$ also $M' = \mathcal{E}_\Gamma[\![E']\!]$ will be added to the set of states because all the existing transitions can be predicted based on the exposed chains of E (Lemma 3.14) and the exposed labels of E' can be calculated exactly (Lemma 3.16). Note that we can use $\mathfrak{T}_{\Lambda_{\mathbf{Lab}}[\![F]\!]}[\![F]\!]$ instead of $\mathfrak{T}_{\Lambda_{\mathbf{Lab}}[\![E]\!]}[\![E]\!]$ due to Lemma 3.13; we can use $\mathcal{G}_{\Gamma_{\mathbf{Var}}[\![F]\!]}[\![F]\!]$ and $\mathcal{K}[\![F]\!]$ instead of $\mathcal{G}_{\Gamma_{\mathbf{Var}}[\![E]\!]}[\![E]\!]$ and $\mathcal{K}_{up}[\![E]\!]$, due to Lemmas 3.7, 3.10 and 3.11. On the other hand, we can similarly prove that each state created by the Worklist Algorithm corresponds to the exposed labels of some derivative expression of F by induction on the number of transitions created by the Workshop Algorithm.

The last statement of the lemma also follows from Lemmas 3.14 and 3.16 and a number of results on applicability of the chains, generate and kill operators calculated on F to its derivative expressions. □

Bibliography

[Abe00] Robert B. Abernethy. *The New Weibull Handbook, Fourth Edition.* Robert B. Abernethy, North Palm Beach, Florida, 2000.

[AG97] Martín Abadi and Andrew D. Gordon. Reasoning about Cryptographic Protocols in the Spi Calculus. In *Proceedings of the 8th International Conference on Concurrency Theory, CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 1997.

[Alf99] Luca de Alfaro. Computing Minimum and Maximum Reachability Times in Probabilistic Systems. In *Proceedings of the 10th International Conference on Concurrency Theory, CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 66–81, London, UK, 1999. Springer.

[ASSB94] Adnan Aziz, Vigyan Singhal, Gitanjali Swamy, and Robert K. Brayton. Minimizing Interacting Finite State Machines: A Compositional Approach to Language to Containment. In *23rd International Conference on Computer Design*, pages 255–261. IEEE Computer Society, 1994.

[ASSB00] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-Checking Continuous-Time Markov Chains. *ACM Transactions on Computational Logic*, 1:162–170, July 2000.

[BCS07] Hichem Boudali, Pepijn Crouzen, and Mariëlle Stoelinga. A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains. In Kedar S. Namjoshi, Tomohiro Yoneda,

Teruo Higashino, and Yoshio Okamura, editors, *Proceedings of the Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2007.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control Flow Analysis for the pi-calculus. In *Proceedings of the 9th International Conference on Concurrency Theory, CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98, London, UK, 1998. Springer.

[BG98] Marco Bernardo and Roberto Gorrieri. A Tutorial on EMPA: a Theory of Concurrent Processes With Nondeterminism, Priorities, Probabilities and Time. *TCS: Theoretical Computer Science*, 202:1–54, July 1998.

[BH00] Ed Brinksma and Holger Hermanns. Process Algebra and Markov Chains. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *Euro Summer School on Trends in Computer Science*, volume 2090 of *Lecture Notes in Computer Science*, pages 183–231. Springer, 2000.

[BHH+09] Eckard Böde, Marc Herbstritt, Holger Hermanns, Sven Johr, Thomas Peikenkamp, Reza Pulungan, Jan Rakow, Ralf Wimmer, and Bernd Becker. Compositional Dependability Evaluation for STATEMATE. *IEEE Transactions on Software Engineering*, 35(2):274–292, 2009.

[BHHK03] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[BHKH05] Christel Baier, Holger Hermanns, Joost-Pieter Katoen, and Boudewijn R. Haverkort. Efficient Computation of Time-Bounded Reachability Probabilities in Uniform Continuous-Time Markov Decision Processes. *Theoretical Computer Science*, 345(1):2–26, 2005.

[BIM03] Antonia Bertolino, Paola Inverardi, and Henry Muccini. Formal Methods in Testing Software Architectures. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy*, volume 2804

of *Lecture Notes in Computer Science*, pages 122–147. Springer, 2003.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008.

[BT91] Dimitri P. Bertsekas and John N. Tsitsiklis. An Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research*, 16:580–595, August 1991.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CG98] Luca Cardelli and Andrew Gordon. Mobile Ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin / Heidelberg, 1998.

[CGH05] Muffy Calder, Stephen Gilmore, and Jane Hillston. Automatically Deriving ODEs from Process Algebra Models of Signalling Pathways. In *Proceedings of Computational Methods in Systems Biology, CMSB 2005*, pages 204–215, Edinburgh, Scotland, 2005. University of Edinburgh.

[CH09] Federica Ciocchetta and Jane Hillston. Bio-PEPA: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science*, 410:3065–3084, August 2009.

[CHZ08] Pepijn Crouzen, Holger Hermanns, and Lijun Zhang. No cycling? Go faster! On the minimization of acyclic models. Technical report, 2008.

[CJEF96] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design*, 9:77–104, 1996.

[CM04] Brian Chess and Gary McGraw. Static Analysis for Security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[dNFP98] Rocco de Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24:315–330, May 1998.

[EH86] E. Allen Emerson and Joseph Y. Halpern. “Sometimes” and “Not Never” Revisited: on Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33:151–178, January 1986.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the Association for Computing Machinery*, 5:345, June 1962.

[FLW] Harald Fecher, Martin Leucker, and Verena Wolf. Don't Know in Probabilistic Systems. In *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria*.

[GHR92] Norbert Gotz, Ulrich Herzog, and Michael Rettelbach. TIPP – a language for timed processes and performance evaluation. report Technical Report 4/92, IMMD VII, University of Erlangen-Nurnberg, 1992.

[GKPP95] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. A Partial Order Approach to Branching Time Logic Model Checking. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems (ISTCS'95)*, pages 130–139, Washington, DC, USA, 1995. IEEE Computer Society.

[GS97] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, 2 revised edition, July 1997.

[Her93] Holger Hermanns. *Semantik für Prozeßsprachen zur Leistungsbewertung*. Master's thesis, IMMD 7, Universität Erlangen-Nürnberg, 1993.

[Her02] Holger Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.

[HJ94] Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

[HKMKS00] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. Towards Model Checking Stochastic Process Algebra. In *Proceedings of the Second International Conference on Integrated Formal Methods, IFM '00*, pages 420–439, London, UK, 2000. Springer.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HR94] Holger Hermanns and Michael Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In *Proceedings of the 2nd Workshop on Process Algebras and Performance Modelling, PAPM '94*, pages 71–87, 1994.

[HS00] Peter G. Harrison and Ben Strulo. SPADES - a Process Algebra for Discrete Event Simulation. *Journal of Logic and Computation*, 10(1):3–42, 2000.

[Joh07] Sven Johr. *Model Checking Compositional Markov Systems*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2007.

[KKLW07] Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf. Three-Valued Abstraction for Continuous-Time Markov Chains. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 311–324. Springer, 2007.

[KKN09] Joost-Pieter Katoen, Daniel Klink, and Martin R. Neuhäußer. Compositional Abstraction for Stochastic Systems. In Joël Ouaknine and Frits W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary*, volume 5813 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2009.

[Kle52] S.C. Kleene. *Introduction to metamathematics*. North-Holland Publishing Company, 1952. Co-publisher: Wolters–Noordhoff; 8th revised ed.1980.

[KNP06] Marta Kwiatkowska, Gethin Norman, and David Parker. Game-based Abstraction for Markov Decision Processes. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, QEST '06*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[KNPS06] Marta Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance Analysis of Probabilistic Timed Automata Using Digital Clocks. *Formal Methods in System Design*, 29:33–78, July 2006.

[Koz83] Dexter Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KS83] Paris C. Kanellakis and Scott A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the second annual ACM symposium on Principles of distributed computing, PODC '83*, pages 228–240, New York, NY, USA, 1983. ACM.

[KS08] Bartek Klin and Vladimiro Sassone. Structural Operational Semantics for Stochastic Process Calculi. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary*, volume 4962 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2008.

[Lar88] Kim G. Larsen. Proof System for Hennessy-Milner Logic with Recursion. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming, CAAP'88*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, London, UK, 1988. Springer.

[LS89] Kim G. Larsen and Arne Skou. Bisimulation Through Probabilistic Testing. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages, POPL'89*, pages 344–352, 1989.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil90] Robin Milner. Handbook of Theoretical Computer Science (vol. B). chapter Operational and Algebraic Semantics of Concurrent Processes, pages 1201–1242. MIT Press, Cambridge, MA, USA, 1990.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (I and II). *Information and Computation*, 100:1–77, September 1992.

[NLLM09a] Rocco Nicola, Diego Latella, Michele Loreti, and Mieke Massink. On a Uniform Framework for the Definition of Stochastic Process Languages. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems, FMICS '09*, volume 5825 of *Lecture Notes in Computer Science*, pages 9–25, Berlin, Heidelberg, 2009. Springer.

[NLLM09b] Rocco Nicola, Diego Latella, Michele Loreti, and Mieke Massink. Rate-Based Transition Systems for Stochastic Process Calculi. In *Proceedings of the 36th Internatilonal Collogquium on Automata, Languages and Programming: Part II, ICALP'09*, volume 5556 of *Lecture Notes in Computer Science*, pages 435–446, Berlin, Heidelberg, 2009. Springer.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and Effect Systems. In *Correct System Design*, number 1710 in Lecture Notes in Computer Science, pages 114–136. 1999.

[NN07] Hanne Riis Nielson and Flemming Nielson. Program Analysis and Compilation, Theory and Practice. chapter Data Flow Analysis for CCS, pages 311–327. Springer, Berlin, Heidelberg, 2007.

[NN09] Hanne Riis Nielson and Flemming Nielson. A Monotone Framework for CCS. *Computer Languages, Systems and Structures*, 35(4):365–394, 2009.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer, 1999. Second printing, 2005.

[NNN07] Sebastian Nanz, Flemming Nielson, and Hanne Riis Nielson. Topology-Dependent Abstractions of Broadcast Networks. In *Proceedings of the 18th International Conference on Concurrency Theory, CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2007.

[NNN08] Sebastian Nanz, Flemming Nielson, and Hanne Riis Nielson. Modal Abstractions of Concurrent Behaviour. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain*, volume 5079 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2008.

[NNPR04] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, and Debora Rosa. Static Analysis For Systems Biology. In *Proceedings of the Winter International Symposium on Information and Communication Technologies, WISICT '04*, pages 1–6. Trinity College Dublin, 2004.

[NRNPdR07] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, and Debora S. da Rosa. Control Flow Analysis for BioAmbients. In *Proceedings of the First Workshop on Concurrent Models in Molecular Biology, BioConcur 2003, Electronic Notes in Theoretical Computer Science*, volume 180, pages 65–79, Amsterdam, The Netherlands, 2007. Elsevier Science Publishers B. V.

[NSK09] Martin R. Neuhäußer, Mariëlle Stoelinga, and Joost-Pieter Katoen. Delayed Nondeterminism in Continuous-Time Markov Decision Processes. In Luca de Alfaro, editor, *Proceeding of Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK*, volume 5504 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2009.

[NV90] Rocco De Nicola and Frits W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proceedings of the LITP Spring School on Theoretical Computer Science*, pages 407–419, London, UK, 1990. Springer.

[NZ10] Martin R. Neuhäußer and Lijun Zhang. Model Checking Interactive Markov Chains. In J. Esparza and R. Majumdar, editors, *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010, Paphos, Cyprus*, volume 6015 of *Lecture Notes in Computer Science*, pages 53–68, Berlin, April 2010. Springer.

[O'C99] Colm Art O'Cinneide. Phase-Type Distributions: Open Problems and a Few Properties. *Communications in Statistics: Stochastic Models 15*, pages 731–757, 1999.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer.

[Pil07] Henrik Pilegaard. *Language Based Techniques for Systems Biology.* PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2007.

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Pri95] Corrado Priami. Stochastic π-Calculus. *The Computer Journal*, 38(7):578–589, 1995.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[RPS+04] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Y. Shapiro. BioAmbients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

[SDBR84] Charles Antony Richard Hoare Stephen D. Brookes and Andrew William Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31:560–599, June 1984.

[Smi10] Michael J. A. Smith. Compositional Abstraction of PEPA Models for Transient Analysis. In Alessandro Aldini, Marco Bernardo, Luciano Bononi, and Vittorio Cortellessa, editors, *Proceedings of Computer Performance Engineering - 7th European Performance Engineering Workshop, EPEW 2010, Bertinoro, Italy*, volume 6342 of *Lecture Notes in Computer Science*, pages 252–267. Springer, 2010.

[TP97] Daniele Turi and Gordon Plotkin. Towards a Mathematical Operational Semantics. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, LICS '97, pages 280–291, Washington, DC, USA, 1997. IEEE Computer Society.

[Tri02] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., Chichester, UK, 2002.