Distributed Monitored Processes

Andrew Gauci

Faculty of ICT

University of Malta

Submitted for the degree of Master of Science

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Distributed Monitored Processes

submitted is my work, except where acknowledged and referenced.

Andrew Gauci

April 11

Abstract

Together with in an increase in system complexity, over these past years the rise of the internet and service-oriented architectures has also brought an increase in distributed and component-based systems. These trends have an adverse effect on system dependability, thus increasing the need for software verification techniques, in particular ones for distributed systems.

For monolithic systems, runtime verification has been shown to be a highly effective technique for ensuring correct behaviour of a system. However, runtime monitoring of distributed systems has proved to be a major challenge, and although various solutions have been proposed, it is unclear which approaches are most appropriate in which context. Issues include, but are not limited to where the monitoring processes should reside, avoiding monitor-induced information exposure, how to handle dynamic system topologies and setting up monitors of new properties at runtime.

In this thesis, we present mDPi, a π-calculus adaptation with explicit notions of monitoring and location. Through this calculus we formally reason about, compare and contrast different runtime verification techniques for distributed settings, investigating different monitoring strategies through a family of bisimulations. We also present the novel *migrating monitor approach*, capable of minimising exposure while also being applicable to systems admitting a dynamic topology. We show how a simple specification language can be monitored in different ways by presenting various conversions into mDPi monitors. Finally, a proof-of-concept implementation is also presented, whose aim is the discovery of practical aspects which emerge during an implementation of the calculus.

He who loves practice without theory
is like the sailor who boards ship without rudder and compass,
and never knows where he may cast.

- Leonardo da Vinci

Acknowledgements

I would firstly like to express my gratitude to both my supervisors, Adrian Francalanza and Gordon J. Pace, in equal measure for their consistent guidance, patience, and most importantly of all for making this masters degree a fruitful and educational exercise.

Special thanks also go to my family and Emily, for their support when I needed it the most. A final, but very special acknowledgement goes to my parents, who I can never thank enough for all they have done for me. I wouldn't have made it this far without them.

The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship Scheme (Malta). The scholarship is part financed by the European Union European Social Fund.

Contents

List of Figures

1. Introduction

1.1 Aims and Motivation

1.1.1 Background

As systems become more complex, standalone architectures are becoming less common, and distributed and component-based systems are becoming more frequent. This shift is emphasised further with the rise of the internet and service-oriented architectures. However, system distribution also implies an increased level of complexity, as well as hampered dependability, emphasising the need for software verification techniques tailored for distributed systems.

An increasingly pursued approach involves a process of *runtime verification*, concerned with the *formal* verification of *system traces*. More specifically, runtime verification involves the employment of an *executable monitors* which analyse the system's execution against a set of *formalised requirements*. Verification can either occur *on the fly*, during execution of the system, or on a *pre-recorded* trace. A trace can be broadly considered to consist of the sequence of internal states and/or events which the system goes through during its execution. In general, applying runtime verification techniques to particular scenarios is a non-trivial task. Firstly, one is to choose an appropriate specification language for the definition of *safety properties* which the system is to adhere to. Next comes the design of an adjacent monitoring algorithm, which verifies system correctness by comparing formalised requirements against the system's exhibited behaviour. Crucially, this approach is considered a *lightweight* verification technique, implying that the complexity of associated algorithms must be tractable. Moreover, given that the monitoring effort can be executed in conjunction with the underlying system, it is important that implied verification overhead is minimised, in order to refrain from consuming an unreasonable amount of resources otherwise available to the system. An advantage of verifying at runtime involves identifying software faults in a *timely* manner, possibly applying reparations to recover from anomalous states.

The following thesis is concerned with the study of runtime verification techniques applied to distributed settings. The primary characteristic of distributed systems is that they consist of autonomous, concurrently executing subsystems communicating through message passing techniques. Communication mediums are assumed to be unreliable and an expensive resource, typically considerably slower than local interactions. Moreover, subsystems (i) lack access to a global clock, implying asynchrony amongst distributed locations, and (ii) each admit local, possibly confidential memory (*i.e.,* no shared memory). By confidential we mean that knowledge of said information is to be kept local to the subsystem. In the presence of sensitive information, it is the responsibility of the monitoring framework to avoid unnecessary exposure of such data, both during (potentially unsecured) interactions, as well as across remote subsystems. We also consider the possibility of system configurations to change dynamically at runtime, in unpredictable ways. These changes are characterised through the addition (or removal) of new subsystems, as well as the evolution of the internal communication topology. Most service oriented architectures involving dynamic lookup, peer-to-peer systems, as well as Enterprise Service Bus architectures are instances of such systems.

As long as one only needs to verify each subsystem independently, runtime monitoring is no different from that used on standalone architectures. However, when properties refer to traces from more than one subsystem, various issues arise. In fact, the underlying system's distribution poses a major challenge to the application of runtime monitoring. More specifically, it is unclear whether the monitoring effort should be placed at a central location, or distributed accordingly. As long as the verified property concerns only public communication taking place between subsystems, the former approach works well by employing a monitor which overhears information as necessary. However, in the case of properties which refer to information local to subsystems, more pressing issues arise. Clearly, exporting local information to a central monitor is undesirable; not only does it unreasonably increase the communication overhead on the system, but it also exposes potentially sensitive information. For instance, applying a central monitor to some web service composition, with one of these web services entailing an online bank is an unsound choice; it is undesirable for the bank to transfer its local bank account information to the central monitor in order to execute necessary verification.

We hence turn to the latter approach, entailing the distribution of monitoring functionality across subsystems. By localising monitors, we avoid the need to expose information remotely, which also results in a reduction of bandwidth overheads. However, identifying an optimal instrumentation strategy for localised monitoring is not straightforward. This is especially true when facing dynamic architectures, since it is unclear how an already installed monitoring framework can be re-distributed at runtime to keep up with the system's unpredictable developments. Consider for instance some peer-to-peer system, such that a property is to be deployed over all contributing nodes; it is unclear how dynamically added nodes are to be instrumented with additional monitors without the need for system recompilation and restart. Another major issue is the lack of synchrony amongst subsystems, which means that certain consequentiality properties may not be monitored in a complete manner [58]. More specifically, the lack of a global clock means that the total order of events across different locations cannot always be observationally determined, and is hence something we will have to live with.

1.1.2 Contributions of this Thesis

Our aim is to distill and identify the core aspects of distributed monitoring, presenting solutions where possible. Although the field is vast, and admits numerous aspects which deserve in-depth study, we shall focus on the formalisation of the scenario of interest, through which we are later able to precisely reason about the application of different candidate monitoring approaches. To this effect, we shall provide the following contributions described below.

- This thesis' main contribution entails the presentation of mDPi, a π-calculus adaptation with explicit notions of monitoring and distribution. Apart from studying distributed monitoring at a formal level, this calculus aims to provide a general framework through which alternate monitoring approaches are evaluated, compared and contrasted. This is achieved through a family of bisimulations, which highlight differences, and/or ignore uninteresting detail of behaviour associated to different monitoring strategies.

- The proposal of the *migrating monitor* approach represents another contribution; employing monitors which verify locally, and migrate as necessary when remote subsystem behaviour becomes pertinent to the system's overall correctness. This novel instrumentation technique is our response to difficulties with monitoring dynamic architectures, while minimising issues with information confidentiality. By adopting migration as a language primitive, we argue that the global monitoring effort is re-distributable on the fly, since monitors can be redirected to new locations according to information learnt at runtime. Moreover, in the presence of confidential information monitors are transferred to the relevant locations in order to localise the monitoring effort, thus minimising exposure. Finally, we believe migrating monitors to allow for the monitoring of properties learnt at runtime *i.e.*, properties whose specification is partly known at compile time, only to be later instantiated at runtime.

- We also formalise desirable properties of mDPi, culminating in a proof ascertaining that the language is, in a sense, well-behaved. More specifically, we show that the defined monitoring semantics does not affect the underlying system's computation at a conceptual level.

- Another contribution lies with an illustration of the calculus' use by showing how a simple specification language can be monitored in different through different monitoring strategies, and is presented through various conversions into mDPi monitors.

- Finally, we present a proof-of-concept implementation, whose aim is the discovery of practical aspects which emerge during an implementation of the calculus. These considerations include issues with security, the implementation of monitor migration, as well as a practical solution for achieving a monitoring framework in the face of both dynamic and heterogenous environments.

1.2 Document Outline

The following dissertation is organised into four main parts as follows;

- The first part introduces necessary background topics required for an understanding of the presented work. Chapter 2 provides an overview of runtime verification, discussing its application, advantages and pitfalls. Chapter 3 discusses the extension of runtime verification to distributed settings. More specifically, we provide an overview of the extended difficulties faced due to system distribution, motivate a broad taxonomy of approaches to distributed monitoring, and also discuss the issue of obtaining a temporal ordering across remote unsynchronised locations. Chapter 4 provides an overview of the π-calculus.

- The next part deals with our theoretical development. Chapter 5 entails our main contribution, in the form of an in-depth presentation of the mDPi calculus. The chapter contains a description of the language, its extensible LTS semantics, as well as a discussion of the mechanism adopted for considering system behaviour at various levels of abstraction. We also adopt the standard bisimilarity relation as a measure of behavioural equivalence amongst mDPi systems, and use this tool to reason about various monitoring strategies. Finally, this chapter provides an overview of a proof of mDPi monitors' well-behaved semantics *wrt.* its underlying system computation. Chapter 6 presents various conversions of property specifications written using Regular Expressions into different mDPi monitors, with each conversion broadly representing a different monitoring approach.

- The third part presents an evaluation of our work. Chapter 7 entails a proof-of-concept case study. This is followed by chapter 8, which presents an in-depth overview of current approaches to distributed monitoring, and also contains an a posteriori comparison of current work to mDPi.

- The fourth part concludes presented work, presenting a short resumé and critical analysis of contributions presented in this thesis, and also investigates possible future work which may prove to be fruitful.

Part I

Background

2. Runtime Verification

The ever increasing ubiquitous nature of computing implies a growing need for *correct* software. Numerous techniques have been proposed for *verifying* correctness, with all approaches admitting unique characteristics for optimal application. This implies that, in general, no technique is best. The following chapter introduces *runtime verification*, an alternate approach concerned with *dynamically* verifying correctness of *system traces*. In other words, given a *log* of system behaviour as exhibited at runtime (*i.e.,* a trace), its contents are subsequently analysed against a set of *formalised requirements*. Section 2.1 motivates this approach, presenting an overview of the field in the process. Section 2.2 explores the issue of requirements specification, which is followed by section 2.3 discussing associated monitoring algorithms. Section 2.4 identifies the role of the system during the design of a runtime monitoring effort. Finally, section 2.5 concludes the chapter.

2.1 Overview

Ever since the inception of software engineering, there has been the need for *correct software*. By *correct* we understand that the system behaves as intended by its *requirements*. However, *software faults* i.e., deviations of a system's expected and exhibited behaviour [60] are an unfortunate reality of software development. Moreover, said faults can be costly, especially for systems of a *mission-critical* nature [25]. It is for this reason that we turn to the field of *software verification*, concerned with the study of techniques for obtaining guarantees of software correctness. Said guarantees are attained by ensuring that the system *implementation* conforms to its *specification*. The complexity of system implementations of any appreciable size implies that software verification is a non-trivial field. We broadly classify verification techniques under two categories; referred to as *static* and *dynamic analysis* [24, 86].

Static analysis techniques involve the interpretation of software *prior to their execution*. Verification ranges from preliminary analysis at the source code level, to formal approaches for proving correctness of system *models*. Techniques belonging to the former are best at identifying certain types of faults, including the potential for memory leaks, nullity references and infinite loops [86]. Software metrics also fall under this category [52]. On the other hand, the latter form of static analysis refers to *model checking* [23, 43] techniques. More specifically, model checking extracts a mathematical model of possible system behaviour, and attempts to prove its correctness with respect to a formalised set of requirements. These requirements are specified through some appropriate *formalism*. Although model checking conceptually offers the *most* guarantees by considering each possible execution path, it is intractable for systems of any appreciable size [23, 25, 87].

Dynamic analysis refers to verification techniques which analyse systems' runtime behaviour. Advantages relating to dynamic analysis relate to (i) the *tractability* of algorithms adopted at runtime, (ii) an increased *confidence* in system behaviour since we are dealing with implementations, rather than ideal models (which may not perfectly mirror the implementation), and (ii) the increased *precision*, due to said algorithms (and execution) having access to information only available at runtime. Relating to this latter point, certain aspects of system behaviour may also depend on the execution environment, emphasising the need for dynamic analysis [24]. Said dynamic verification often results in an analysis of select traces, whereby each *trace* is considered to represent a *single* execution path. Techniques belonging to this approach are based on the observation that although verifying *each* possible trace is intractable, checking *particular* traces against properties of interest (in a verification setting) is usually cheap. *Testing* [68] is the most common form of dynamic analysis; its application involves the identification of a set of *representative* system traces, whose verification gives sufficient *confidence* of the system's overall correctness [24]. Hence, the core aspect with respect to testing is the identification of suitable traces, taking the form of *test case scenarios*. Advantages with testing include (i) its scalability, and (ii) a relative ease of application (especially when compared to other more formal approaches to verification). However, testing can only find the presence, and not prove the absence of bugs [30], implying an inherent *lack of coverage*.

Runtime verification [24, 13, 25, 56, 60] offers an alternate approach, and is concerned with the *formal* verification of *system traces*. Its operation is summarised in Fig. 2.1. Each trace encodes a system's *execution* or *run*, entailing a sequence of *logged system events* obtained through a process of *instrumentation* (section 2.4.1). Typical events vary from simple variable updates, to interactions with outside entities. Verification is achieved through an *executable monitor*, responsible for checking the system's runtime trace against a set of *formalised requirements*. Hence note the crucial role placed in the *extraction* of the trace; subsequent monitoring algorithms are based on a correct tracing semantics. Monitoring can either be executed either at runtime (known as *synchronous* monitoring), or distinctly from system execution (*i.e., asynchronous* verification). Finally, given that synchronous monitors execute during system executions within the same *environment*, this implies that runtime monitoring should be pursued as a *lightweight verification* technique [60].

The attractiveness of this approach is its *illusion* of a tractable approach for achieving perfect

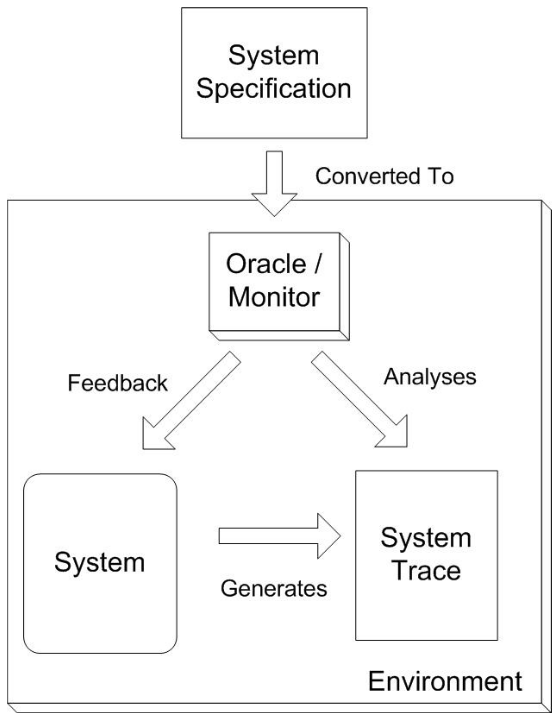


Figure 2.1: The runtime verification scenario.

coverage; by verifying *exhibited behaviour*, it appears as if no execution goes unverified. In other words, runtime verification exploits the fact that we are *mostly* interested in behaviour exhibited at runtime. Although faults may reside in other potential executions, we (mostly) do not identify them until their occurrence (at runtime). Notice the use of the word ‘mostly’; this is due to work on *predictive analysis* of possible traces constructed from extracted causal models [65]. Note the change in philosophy as opposed to other techniques; techniques such as testing and model checking try to identify incorrect behaviour *prior* to system deployment. On the other hand, runtime verification guarantees that the system will never go *beyond* an inconsistent state — but can reach said states. Hence, with runtime monitoring the verification process can go beyond deployment. The handling of inconsistent states varies between monitoring frameworks; whereas some [88, 88] are content with *identifying* incorrect behaviour, others go a step further, by specifying a *reaction* to violations (in the form of *feedback*), thus steering the system back to an acceptable state [25, 80, 65]. Hence, runtime verification can also be a crucial tool in the development of *fault-tolerant systems*. Finally note that the runtime aspect of monitoring algorithms imply the verification of properties relating to (i) *safety*, and (ii) *bounded liveness* (*i.e.*, an alternate form of safety) [24].

Runtime verification achieves a niche in the software verification field. It is advantageous compared to testing on two fronts; (i) by offering elevated guarantees through a solution giving the illusion of 100% coverage, and (ii) by *avoiding* the issue of test case generation, since it is the system execution which generates the trace to be verified (unlike testing, whereby test case generation is a burden left to the user). Unlike model checking, runtime monitoring (i) offers a tractable and scalable solution to formal analysis (albeit with less guarantees), and (ii) takes into consideration the execution environment. These issues, coupled with unique advantages with respect to fault tolerance, make runtime verification a novel field worth of further study. On the other hand, runtime verification also admits certain issues. Firstly, since synchronous runtime monitors execute within the same environment as the system (depicted above), then monitors pose an *overhead* by consuming resources otherwise available to the system. Moreover, runtime verification offers less guarantees as opposed to model checking [44]. Another issue is that we require more effort (as opposed to testing) to both (i) formally specifying requirements, as well as to (ii) instrument the monitor with the underlying system. Finally, we rely on a correct tracing semantics, and a properly executing monitor in order for verification to occur correctly. The interested reader is pointed to [60, 24] for more information *wrt.* the comparison of software verification approaches.

Although runtime verification might seem as a bridge between model checking and dynamic analysis due to its use of formalised specifications, it is conceptually more akin to testing [24]. This is due to both approaches being concerned with the *dynamic* verification of select *traces*, sometimes referred to as *oracle-based verification* [60]. Runtime verification is generally preferred for mission-critical systems, whereas testing is the chosen method of verification when the system runs under strict computational and memory consumption restrictions [24], or when the necessity of software correctness is less urgent.

In truth, the difference between static and dynamic analysis is often overemphasised [33]; numerous techniques can be applied in complimentary fashion. For instance, one can use static analysis to analyse system structure in order to generate representative test case suites [25]. Moreover, testing and runtime verification can also be complimentary [8, 24]; one could verify the system trace generated by test cases. Static analysis is also often used in conjunction with runtime verification. For instance, the approach in [43] considers the use of runtime monitors to guide static analysis in order to reduce the state space. On the other hand, [87] builds a monitoring framework on top of a static analysis tool.

We identify *three* key considerations during the design of a runtime verification framework; (i) suitable *requirements specification*, (ii) design choices for an appropriate *monitoring algorithm*, and (iii) an analysis of the underlying *system*, its *environment* and how the monitor affects/integrates with both. We shall explore each in further detail throughout this chapter.

2.2 Requirements Specification

The first step for the design of a runtime verification framework involves identifying an appropriate *formalism*. This language in turn allows us to *precisely* define necessary system requirements of interest. To this effect, the use of *temporal logics* has been intensely studied in a runtime verification setting [13, 55, 25, 24]. Temporal logics [23] define *logical assertions* qualified in terms of time, and are an important tool in specifying how systems *should* execute *i.e.,* by specifying *temporal properties* on traces. Numerous logics have been applied in a runtime verification setting [72, 27, 25, 17, 37], each with unique *characteristics*. These include issues of *expressivity*, the assumed *time model* [14], as well as varying *application domains* [27, 25]. Given this wide variety of choices, we identify a broad set of factors to keep in mind for identifying the most suitable specification language.

1. *Properties of Interest* — Properties typically of interest vary from *behavioural* requirements, to more *quantitative* aspects of execution. It is therefore our task to adopt a language which is capable of encoding necessary specifications. Behavioural properties include *sequentiality* ("Event a happens after b"), *invariance* (condition X on the state should never be true), *timing* issues ("Event a should not take more than X minutes") or a combination thereof. On the other hand, quantitative aspects are more related to the *quality of execution*, for instance related to performance profiling or throughput analysis [36].

2. *Monitoring Algorithm* — In general, there exists a *tradeoff* between logic expressivity and system overhead [25]. Therefore, given our interest in *lightweight monitors* [60], the choice of logic is limited by what can be *efficiently* verified in both time and space. Other design choices of the monitoring algorithm can also affect the choice of language, as shall be discussed in section 2.3.

3. *System Structure* — The underlying system structure can also potentially affect the choice of formalism. For instance, as shall be seen in chapter 3, we are faced with certain restrictions on what can be *reasonably* monitored within *distributed systems*. Also relating to the previous point, physical limitations on system resources (as is common in for example *embedded systems* [21]) may limit the choice of logic expressivity.

The interested reader is pointed to [25] for a more in-depth overview of logic characteristics, and subsequent applications. Keep in mind that it is unusual for *complete* system specifications to be available in a runtime verification setting [24]. This implies that we do not attempt to classify each behaviour as either correct or incorrect. Instead, we mostly deal with *partial* specifications, formally specifying mission-critical aspects of system executions.

System properties are often categorised as ether properties of *safety* or *liveness* [60]. In essence, the former dictates that something bad should not happen, whereas the latter states that something good eventually happens. Falcone [34] summarises differences between safety and liveness within the context of runtime verification as one of *trace finiteness*. In the case of safety properties, their validity can always be falsified by a finite trace. However, liveness properties cannot be falsified by finite traces, since any finite trace can be the prefix of an infinite one satisfying the property. Unfortunately, this implies that the semantics of our gradually verifying monitors cannot handle liveness properties. Conversely, we are restricted to the monitoring of safety properties during runtime verification. Some frameworks consider truncating traces for considering liveness properties (*i.e.*, bounded liveness) [26], which still however reduces to a form of safety.

Finally, a few example formalisms used in a runtime verification setting include *Linear Temporal Logic* (LTL) [72], *Regular Expressions* [83], *Dynamic Automata with Timers and Events* (DATEs) [25] and the *Duration Calculus* [17]. LTL is perhaps the most popular, serving as a theoretical basis for numerous runtime verification frameworks [45, 44, 40, 24]. More specifically, LTL is a linear-time logic particularly adept at specifying regular properties of discrete time systems [60]. Regular expressions have also been applied [74, 15], serving as an *intuitive* specification of temporal orderings on events. By being automaton-based, DATEs offer a more graphical approach to property specification, while still achieving elevated expressivity. The duration calculus is a real-time logic, involving the specification of properties over time intervals.

2.3 Monitoring Algorithm

Given a high-level specification of system requirements, the next step involves verifying *exhibited behaviour* for correctness with respect to the set of formalised properties. This process is carried out in a runtime verification setting by an executable *monitor*, tasked with analysing system traces and returning a *verdict* of their validity *wrt.* a set of desirable properties. Note that the conversion between property and monitor is usually carried out in automatic fashion [60].

Formally, given a set of n system properties $p_i \in \varphi$ *s.t.* $[\![p_i]\!]$ is the set of *valid traces* representing property p_i, the monitor is tasked with identifying whether trace $T \in \bigcup_{i=1}^{n}[\![p_i]\!]$. In other words, runtime verification is a problem of *word inclusion* [60], identifying whether the trace extracted at runtime belongs to the set of traces characterised by the system requirements. Apart from issues with the *algorithm complexity* (introduced in section 2.2), [60] identifies two requirements for the design of the ideal monitoring algorithm

- *Impartiality* — such that a verdict is never given prematurely. In other words, a monitor should never output its result if it is still possible for the remainder of the path to point to an alternate verdict.
- *Anticipation* — whereby a verdict is given as soon as possible. Consequently, a monitor should return its result as soon as the remainder of the path cannot change the verification outcome.

In general, any given runtime verification framework is based on well-defined (i) *monitoring* and (ii) *tracing* semantics. The former is necessary for specifying monitor operation during the *analysis* of traces against the set of requirements. The choice of specification language dictates the definition of monitoring execution, thus usually comprising of an operational semantics assigned to the language. On the other hand, the tracing semantics defines the extraction of traces during system execution, and is hence an important part of the instrumentation process (section 2.4.1). Both semantics are tightly bound; we need *appropriate* traces in order for monitors to *correctly* analyse runtime behaviour. Conversely, traces which incorrectly encode system behaviour nullifies subsequent monitoring efforts. The remainder of this section deals with design decisions faced during the definition of a monitoring semantics. Section 2.4.1 discusses instrumentation approaches.

2.3.1 Synchronous vs Asynchronous Verification

The process of trace analysis can be performed either (i) *on the fly* as the trace is being generated (*syncrhonous* verification), or (ii) *after the fact*, once extensive knowledge of the trace is available (*asynchronous* verification) [25]. In case of synchronous verification (also referred to as *online monitoring*), the monitor is tasked with analysing the *current* execution, updating its verdict *incrementally* with each pertinent system event. The monitor usually runs in parallel with the underlying system (within the same environment), receiving trace events as they occur. Moreover, the system sometimes waits for the monitor to complete its verification before proceeding. It is hence crucial for synchronous monitors to reach a verdict using as little resources as possible. This implies that synchronous monitors rarely have access to long trace histories, and each computational step must take reasonably efficient time. Moreover, given that these monitors can only return a verdict on the trace *seen so far*, this minimally imposes the need for three-valued logic; *valid*, *invalid*, or *inconclusive* [60]. On the other hand, asynchronous verification (*offline monitoring*) involves the analysis of pre-recorded traces. Given that verification is performed after the fact, this implies that complete, or at least extensive knowledge of the trace is usually known, allowing for analysis using a broader class of languages (see [36]). Efficiency can still be an issue with offline monitors, in that we are still intolerant to intractable verification algorithms (given our interest in lightweight monitors). However, the *urgency* for efficiency is somewhat relaxed, since offline monitors do not consume the system's resources.

The main desirable aspect of synchronous verification is its *timeliness*; by keeping up with the system execution the monitor (i) can exploit dynamic aspects of execution only available at runtime, and (ii) allows for reactions to violations. The first issue is particularly useful when dealing with the extraction of *causal* orderings on possible events [65], and will be crucial in our design of a distributed monitoring framework at a later stage. The second issue is also important, extending runtime verification as a valid tool for implementing *fault-tolerant systems*; allowing for the specification of *corrective measures* in case of software failure. On the other hand, synchronous monitors only have access to partial traces, decreasing the class of monitorable properties. Moreover, reaching a verdict can take a substantial amount of time, depending on the duration of system execution. Finally, by running within the same environment, the system consumes resources otherwise available to the system (more below). On the other hand, asynchronous verification avoids this issue by proceeding distinctly from system execution. Overhead is therefore minimised to the additional tracing effort for trace extraction. However, although complete knowledge of the trace has its advantages, asynchronous verification also implies that we cannot exploit the dynamic aspect of system execution. By extension, verification after the fact makes it impossible for the triggering of compensations to violations. This in effect limits the use of runtime verification as a flagging mechanism for property violations, limiting its usefulness on deployed systems [60].

2.3.2 Reaction to Violations

We are faced with a design choice when a system enters an inconsistent state. Although runtime verification is *mostly* concerned with identifying faults in system behaviour, many frameworks extend this notion by allowing for the execution of *compensatory actions* upon error detection [24]. The specification of violation reactions often take the form of additional code snippets associated to system properties. These actions serve the purpose of *damage limitation*, partially or fully *mitigating* the error imposed by the exhibited fault. As discussed in the previous section, there is an issue of *timeliness* with the *execution* of reaction violations. For instance, if we specify the property that a system should not have memory leaks, by the time we detect an error it will be too late, since the program would have already crashed. However, the property that no unauthorised user should gain access to the system admits a straightforward compensation, by terminating the user's login session. As these examples show, the applicability of violation compensations depends on the property. See [60] for a more comprehensive overview of violation reactions in a runtime verification setting. Example frameworks which support violation reactions include Larva [25], Java-MaC [55] and Java-MOP [20].

A stronger take on corrective verification is *runtime enforcement* [34], which advocates a *proactive* enforcement of correct behaviour (*i.e.,* prior to error occurrence) rather than a *reactive* approach after the fact. The approach in [34], adopts an internal memorization mechanism tasked with reading an input sequence of events, generating a corresponding output sequence in such a way that the desirable property is always fulfilled. Thus when a system is well-behaved, the enforcement monitor is unobtrusive. However, if the system is about to exhibit a deviation *wrt.* the desirable property, the internal memorization mechanism kicks in to prevent the fault.

2.4 The System and Environment

The underlying system and its execution environment are *primary stakeholders* in the design of effective monitoring frameworks. Clearly, systems worth verifying come in different shapes and sizes, and are implemented using different technologies. Moreover, the environment also admits its own characteristics, ranging from software to hardware restrictions. These considerations can substantially affect the design of a runtime monitoring framework, a fact which shall be made particularly apparent in chapter 3. The following section therefore entails the discussion of (i) the instrumentation effort, and (ii) implied overheads during runtime monitoring.

2.4.1 Instrumentation

Instrumentation can be generally considered as the *integration* of the monitoring effort with the underlying system [24, 25], and is considered on two levels;

(i) The *implementation* of a tracing semantics, and;

(ii) The choice of an appropriate *monitoring arrangement*.

The first issue relates to the implementation of a mechanism exposing necessary trace information, in turn allowing the monitoring effort to affect necessary verification. Numerous approaches have been attempted to this effect, typically defining automatic derivations (of this mechanism) from the requirements specification [25, 24, 60], while exploiting particular techniques and/or technologies in the process. One such technique which has been particularly successful includes *Aspect Oriented Programming* [54], with the associated AspectJ [53] as a particular technology. Other approaches include the alteration of the underling operating system kernel [78] or virtual machine [24]. Finally, although instrumentation is often performed in automated fashion in order to reduce manual errors, in doing so we implicitly *bind* the verification framework with specific programming languages and/or technologies (used during instrumentation). It is for this reason that certain runtime verification frameworks opt for manual (or semi-manual) instrumentation, for maximum generality. One such framework includes EAGLE [13].

An interesting dichotomy we identify is the potential distinction between *static* and *dynamic* instrumentation. By static instrumentation, the necessary monitoring effort is installed once prior to system execution, and remains unchanged throughout. Although correct, we identify the possibility of adopting a more powerful dynamic instrumentation effort, dissecting the monitoring effort into generic *instrumenter* and *verifier* components. The instrumenter is responsible for exposing the extracted trace (containing a sequence of events from a possible alphabet). With the trace at hand, the verifier can subsequently execute necessary trace analysis to obtain a verdict on the trace. What is appealing about this dynamic approach is the possibility to start verifying new *properties learnt at runtime* (henceforth referred to as *dynamic properties*), without the need for re-instrumentation and system restart. It is for this reason that we sometimes refer to dynamic instrumentation as a *property agnostic approach* (more in section 3.4.4). However, keep in mind that this enhanced machinery comes at the cost of additional complexity, which may not always be computationally permissible. It is also worth pointing out that this latter approach is more amenable to asynchronous monitoring, since the instrumenter records the trace at a possibly different rate than the verifier is willing to analyse.

Installing necessary monitoring artifacts can be a sensitive activity, especially if we want to perform *optimal* verification on runtime traces [24]. This notion of optimal monitoring typically refers to the use of *minimal* resources for obtaining reduced overhead (section 2.4.2). However, other issues can also become relevant, including issues with *security* and *information confidentiality*. In other words, by extracting information from the underlying system, monitors may also unwittingly expose confidential information, thus posing a security risk. Moreover, difficulties with achieving optimal verification can be magnified further according to the system (and environment) architecture. It is for this reason that the *configuration* of our monitoring effort, also referred to as the *monitoring arrangement*, can be a crucial factor in our endeavour for effective monitors. Consider for instance a distributed system, admitting numerous subsystems at physically distinct locations. Choosing an appropriate monitoring arrangement given such a scenario is not a straightforward task. More specifically, it is not clear which location(s) should be responsible for the global verification effort. We can go further, by saying that the monitoring configuration (given this scenario) is a strong factor in the *efficacy* of the distributed verification process, especially *wrt.* overheads associated with monitor interactions across sub-systems for verification purposes. We shall expand on this point in chapter 3.

Finally, an important issue to be kept in mind *wrt.* instrumentation involves the assumed level of *trust* between monitor and system. In other words, system administrators may require *assurances* on monitor behaviour in order to allow privileged, potentially intrusive monitors to run adjacent to (critical) system executions. Not only can malicious monitors incorrectly alter system executions, but also expose information. As designers of runtime verification frameworks, it is our duty to work within assigned boundaries of confidence, and must also strive to present required assurances if so required.

2.4.2 Overheads During Synchronous Verification

Given that synchronous monitors run in parallel with the system within the same environment, it is unavoidable that the monitors consume resources otherwise available to the system. Said resources take the form of both memory and time consumption. Monitors consume memory through their necessity to keep state. On the other hand, computational time is consumed on the occurrence of each event, with the monitor updating its verdict by taking into account new event information. In general, this issue is tackled through efforts on *minimising* said overheads to *reasonable* levels. What overhead is deemed reasonable depends on the scenario under consideration. Whereas certain scenarios can afford *some* overhead, in other situations any non-trivial overhead is deemed too much (perhaps eliminating runtime verification as a feasible approach altogether). More specific memory or time restrictions are also possible, perhaps due to physical restrictions on the hardware.

Various optimisation approaches applied to runtime verification frameworks has also been studied. [44] proposes a form of property rewriting for simplifying certain properties written in LTL [23]. On the other hand, [15] takes the approach of *partitioning* space, time, or both forms of overhead, in the hope of alleviating the monitor's impact on the system. [31] studies the effect of monitoring on certain data structures, providing upper bounds on their size. [21] proposes a form of properties admitting resource-bounded monitor implementations, achieved by exploiting resource guarantees of synchronous language Lustre[1]. Said properties are subsequently converted to LARVA specifications, thus exporting resource bounded properties to a wider audience of Java applications. Analogously, the framework Lola [27] achieves efficient memory bounded monitoring of synchronous systems.

Finally, it is worth pointing out that the overhead imposed by the online monitor, however small, may alter the underlying system's behaviour [60, 25]. In other words, it is possible that the system behaves differently had the computational time and memory consumed by the monitor been available. This could lead to situations where the monitor slows the system down in such a manner that it breaks a temporal property otherwise adhered to by the system had the monitor not been present. Conversely, analogous situations could arise where the presence of the monitor preserves properties which would have been broken had the monitor been absent. In general, online runtime verification techniques do not only monitor system executions, but in fact monitor system executions affected by the presence of the monitor. Simply put, by observing systems we are altering their behaviour, implying a certain amount of *uncertainty* in the system's true behaviour. A theory has been developed [25], characterising a set of properties written in a (subset of) duration calculus [17] which are *speedup* and *slowdown invariant*. In other words, said properties remain valid even after a system has more/less resources available.

2.5 Conclusions

This chapter presented runtime verification, an alternate approach to software verification concerned with verifying correctness of runtime traces. This approach favourably avoids the intractability of exhaustive analysis, while offering elevated coverage guarantees. Nevertheless, the field admits a set of non-trivial issues, requiring consideration during the design of a monitoring framework. These include the choice of formalism for the specification of requirements, minimising overheads associated with the monitoring process, as well as issues with monitor instrumentation. More importantly, this chapter serves as an introduction to the following, where we focus on the application of runtime verification techniques in a distributed setting.

[1]project website http://www-verimag.imag.fr/Synchrone,30

3. Runtime Verification In A Distributed Setting

The following chapter explores the extension of runtime verification techniques to distributed settings. More specifically, we recognise salient issues which enhance the difficulty in achieving required monitoring functionality, and also propose solutions where possible. To this effect, we identify a broad taxonomy of distributed monitoring approaches. Section 3.1 introduces the problem, and is followed by section 3.2 which recognises distributed system characteristics pertinent to the design of a monitoring framework. Section 3.3 defines an example scenario. Section 3.4 motivates the broad taxonomy, from which the novel *migrating monitor* approach is borne. Section 3.5 discusses difficulties with extracting temporal orderings on remote events. Finally, section 3.6 concludes the chapter.

3.1 Introduction

As systems become more complex, monolithic architectures are becoming less common, and distributed and component-based systems are becoming more mainstream. Furthermore, with the rise of the internet and service-oriented architectures, a monolithic view of certain systems is in certain situations not only undesirable, but also impossible [18]. At the same time, the increased size of the systems and the additional complexity due to the distributed implementations, hamper their dependability, thus emphasizing the need for effective software verification techniques tailored for distributed architectures. Moreover, the inherent concurrency and asynchrony present in distributed systems imply elevated non-deterministic system behaviour [80], thus significantly hampering the effectiveness of traditional approaches to software verification used in monolithic settings. Hence, techniques involving the analysis of system traces become more attractive, which get around non-determinism by letting the system choose the execution path. The following chapter reviews and identifies the most salient issues *wrt.* the application of runtime verification techniques in a distributed setting, culminating in a proposed broad taxonomy of distributed monitoring approaches. Moreover, we shall also present a novel approach which is borne out of this taxonomy.

Runtime monitoring usually involves the instrumentation of additional code within the system to signal relevant events, which are then processed by a central monitor to verify that certain properties are not violated at runtime. As long as one verifies each of a distributed system's components independently, monitoring is no different than the above approach used for monolithic architectures. However, when a property involves more than one subsystem simultaneously, runtime verification becomes considerably more complex. This enhanced complexity is especially due to pertinent characteristics inherent to distributed architectures, discussed below.

3.2 Distributed System Characteristics

We consider distributed systems as a set of autonomous, concurrently executing sub-systems communicating through message passing, as depicted in Fig. 3.1. Conversely, a distributed system can be considered one whose components are partitioned across various *environments*. By environment we refer to some computational space where sub-systems execute, and can differ on physical location, operating system, hardware *etc.*. Nevertheless, we shall often use the term environment and location interchangeably. Most internet-based and service-oriented systems, peer-to-peer systems and Enterprise Service Bus architectures [18] are all instances of this setting.

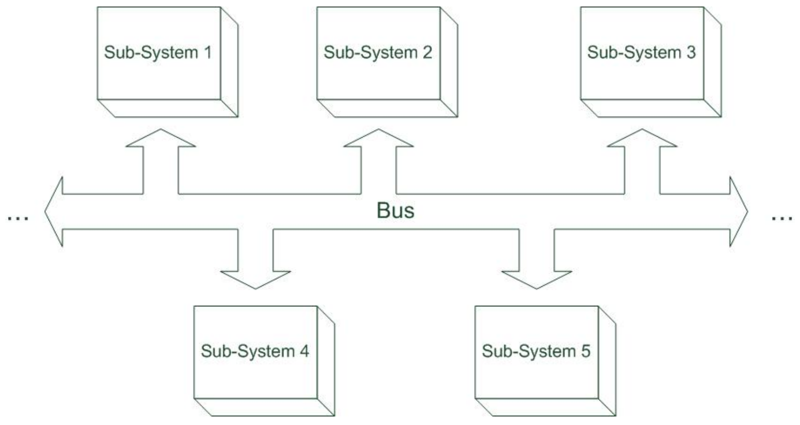


Figure 3.1: A general distributed system architecture

More precisely, a distributed system entails a set of *n sub-systems* and *m channels*. Each sub-system denotes located computation, interacting through channels which serve as a communication medium. Similarly to monolithic systems, each sub-system's execution is represented through a sequence of *states* whose update is triggered by *events*. We assume communication over channels to take an arbitrarily long (but finite) duration. This delay mirrors typical distributed settings which operate under *restricted bandwidth limitations*, which is considerably slower than local interactions. On the other hand, no assumption is made on the message ordering, implying that the order of sent messages is not necessarily mirrored at the receiving end. We adopt channels which are (i) *synchronous* (*i.e.,* the sender receives an acknowledgement from the receiver before proceeding), (ii) *bidirectional* (information can flow in both directions, at different instances), and (iii) *error-free*, in that a sent message is always *eventually* received (possibly after a number of retries).

We shall assume a general scenario whereby distributed systems operate absent access to a *global clock* or *shared memory*. Instead, the system admits a set of local unsynchronised clocks, with each sub-system admitting its local memory space. Moreover, it is impossible to (i) precisely synchronise sub-systems' clocks, and (ii) obtain a global snapshot of the system's state in a *timely* fashion [39]. The former is proven in [63], and is shown to be the result of synchronising over an uncertain communication medium. Moreover, given that channel uncertainty (including communication duration) cannot be completely resolved in a practical setting, this implies a more general impossibility. The latter emerges from the voluminous information transfer involved for such an operation, coupled with delays during channel communication.

The above restrictions are especially poignant to the design of a distributed monitoring framework. Consider the lack of synchrony amongst remote clocks. This implies that it is generally impossible for a monitoring framework to extract a total ordering on remote events, since we cannot precisely relate remote timestamps assigned to events at differing locations. Instead, the best we can do is extract a total ordering for each partition of local events, and a partial ordering globally [58]. Section 3.5 promotes the use of *causality* in order to extract this partial global order.

Although algorithms for the extraction of global state projections do exist (including [9]), they are incompatible with our need for lightweight monitors (section 2.2). More specifically, the voluminous nature of data transfer required for such operations dictates a *high bandwidth overhead* on the underlying system. By extension, monitors operating in a distributed setting are forced to operate without access to a global state. Predicates on the state have to also be altered accordingly, by respecting state location. This point leads to a more general requirement; an efficient monitoring algorithm in a distributed setting is required to *minimise remote communication* for monitoring purposes (as well as limiting consumption of memory and computational time).

Moreover, we also consider subsystems admitting *confidential information*, which should not be exposed globally or to other sub-systems. This issue is especially pertinent when handling certain mission-critical components (such as an online bank, or a hospital's medical record management module), as well as when a degree of *competitiveness* is prevalent amongst sub-systems (including web-service compositions involving agents which provide competing products). In general, it is the responsibility of a distributed runtime verification framework to respect *infor-*

mation locality, since failure to do so leads to *data exposure*. Data exposure can take the form of

1. Exposure *during* remote communication across unsafe mediums, as well as
2. Exposure *across* non-privileged locations.

Finally, we consider the effect of the system's *topology wrt.* to the distributed monitoring effort. A distributed system can admit either a *static* or *dynamic* topology/architecture. The former refers to systems whose contributing nodes are known at compile time, and remain unchanged during computation. On the other hand, systems which admit dynamic architectures admit configurations which *evolve at runtime*. Said dynamicity refers to the evolution of (i) the number of contributing nodes, and/or (ii) the channel links between sub-systems. Distributed systems admitting a dynamic configuration have become more prevalent, and include most login-based systems (since contributing users are coming and going at runtime), as well as web services involving dynamic lookup (such as typical agent broker based scenarios). The possibility of dynamic architectures enhances the difficulty of monitoring their behaviour at runtime. More specifically, identifying which nodes participate in the monitoring of a system property can often only be identified at runtime, since the system evolves in unpredictable ways. It is this unpredictability which points to the need for a monitoring framework capable of keeping up with the system's evolution.

3.3 A Motivating Example

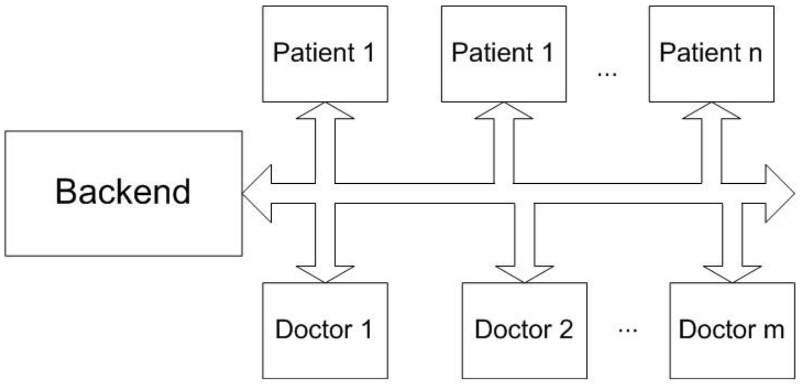


Figure 3.2: A Hospital Management System

Figure 3.2 depicts a typical distributed system, entailing a *hospital management system* running at the backend. The system is responsible for numerous activities, including appointment scheduling, billing, help desk queries and most importantly, patient administration. Moreover, the system admits an online front end, where both doctors and patients can log in for administrative purposes. Doctors have the capability to (i) register patients, (ii) view scheduled appointments and (iii) submit medical diagnoses, whereas patients can (i) book appointments, (ii) submit queries and (iii) request the release of their medical record (containing the latest diagnosis, past medical history etc). All associated confidential information is stored at the backend, but can be periodically requested by doctors and/or patients if necessary. In general, there exists a many-to-many relationship between patients and doctors, where one patient can have more than one supervising doctor, and each doctor supervises numerous patients.

The above scenario shall serve as a running example throughout this dissertation. More specifically, we shall focus on the runtime monitoring of (a subset of) the system's patient record management module. In general, this module admits mission-critical requirements, both to *ensure* its correct operation in order to avoid the misplacement of medical records, as well as to *avoid* leaking confidential information to unauthorised entities. To this effect, we are particularly interested in verifying the protocol adhered to by the system in case of a patient's request for the release of her personal medical record, described below

1. The patient logs into the system.
2. A request is submitted (by the patient) for the release of her personal medical record.
3. The request is received at the backend.
4. The backend fetches the list of doctors enlisted as responsible for the patient's care.
5. A request for the permission to release the record is submitted to each doctor.
6. If one or more doctors object to releasing the record, the patient request is denied. Conversely, all supervising doctors must consent to the patient's request. If all doctors consent, the record is released.

For security reasons, the system grants each patient at most one opportunity to release their medical record from an online request. In other words, multiple requests are satisfied by at most one response. This measure is in place to limit possible exposure of highly confidential information. Any additional requests are manually handled outside the system.

The hospital management systems exhibits typical characteristics described in section 3.2. Clearly, the system is *distributed*, since both the system's contributing entities (backend, doctors, patients) and the system's memory space is partitioned into a set of distributed sub-systems. Moreover, the notion of *information confidentiality* is of considerable importance. More specifically, the backend admits sensitive medical records which should only be exposed to certain entities, at appropriate moments *i.e.*, to the patient and her supervising doctors. Data exposure can take numerous forms; exposure of medical records across unreliable communication mediums (in this case, the internet), as well as exposure across unauthorised entities (for instance by incorrectly sending a patient's record to another unrelated patient). Given that the system front end happens to be online, this implies that the system globally operates within *restricted bandwidth limitations*. Finally, the system admits a *dynamic topology*, since both patients and doctors are logging in and out of the system at runtime, in unpredictable ways. Moreover, determining which entities participate during the handling patient requests is data dependant, since the choice of doctors which contribute to satisfying a particular request can only be known at runtime.

System correctness *wrt.* the handling of medical records and patient requests is of critical importance. As we shall see in more detail in chapter 6, required correctness shall be formally defined through properties on the system's execution. These properties are enlisted below

- *No patient is to be given another patient's record as a response* — This first property prohibits the exposure of a patient's record to another (unintended or otherwise).
- *Multiple patient requests should be responded by at most one medical record release* — In other words, the second property ensures that the hospital management's policy on singleton releases of medical records are adhered to.
- *The release of a patient's record must be approved by supervising doctors* — This final property encapsulates our verification of the described protocol for the handling of patient requests.

Validity of the above three properties offers elevated guarantees of the system's trustworthiness, which is why we shall strive to provide necessary assurances. However this is a non-trivial task for reasons outlined in section 3.2. The following section shall hence investigate different approaches to the monitoring of the above properties.

3.4 Distributed Monitoring

Section 3.2 motivated the enhanced difficulty of applying runtime verification in a distributed setting (which is a consequence of distributed system characteristics). Presently, there exist numerous architectures and tools addressing the runtime monitoring of distributing systems, to varying degrees of success. We argue in favour of their broad classification across two main categories; referred to as *orchestration-based* and *choreography-based* approaches.

In orchestration-based approaches, verification responsibility lies firmly with a central monitor overhearing all information pertinent to the system's global correctness, as seen in figure 3.3. Although this approach works seamlessly on monolithic systems, its application is not as straightforward in a distributed environment. In the case where the monitored property concerns only public communication between subsystems, this approach works well by constructing a monitor overhearing all such communication, modifying its state accordingly. However, when the system property involves local subsystem information, this approach is less than ideal. Firstly, communication of local confidential information across remote locations leads to data exposure. Furthermore, the volume of system event information required for centralised monitoring is substantial, often resulting in unreasonable bandwidth overhead. Finally, orchestrated approaches pose a security risk by presenting a central point of attack, in the form of the monitor, through which sensitive information can be tapped.

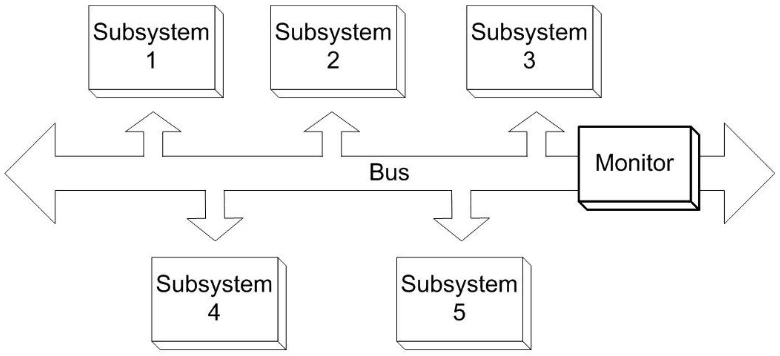


Figure 3.3: An orchestration-based approach.

Choreography-based monitoring takes a more *dataflow dependent* approach, whereby subsystem events drive the choice of monitoring location; thus often leading to a distribution of monitoring functionality throughout the distributed system as seen in figures 3.4 and 3.5. In general, choreography-based monitoring is more advantageous as opposed to orchestrated approaches. By pushing verification locally, a choreographed monitoring approach minimises data exposure and communication overhead by eliminating the need to report back to a central monitor. This does not stop localised monitors from communicating over the communication medium, however the *volume* of information for monitor synchronisation is usually substantially less. Finally, removing the central monitor eradicates the security risk of providing a central attack point. Nevertheless, applying choreography usually requires more complex instrumentation, and is only applicable if *all* subsystems allow for the installation of local monitoring code. In conclusion, although choreographed approaches are usually better, they should only be used in scenarios where it is clearly advantageous to do so.

The choice of specification language is another crucial issue in a distributed setting. Apart from issues such as the language expressivity, the most salient issue particularly relevant to distributed systems is the dichotomy of *static vs dynamic properties* hinted at in section 2.4.1. Static properties refer to properties whose specification is entirely known at compile time, and remains unchanged throughout system execution. The set of events of interest to the verification of static properties hence also remains unchanged, which points to the suitability of a static instrumentation strategy. However, static properties are not sufficiently expressive in case of dynamic architectures (section 3.2). Instead, we turn to the class of dynamic properties *i.e.,* properties whose abstract specification is fully instantiated at runtime through learnt information (sometimes referred to as *contextual* properties), and/or properties completely learnt during system execution. Although this motivated dichotomy is not bound to distributed systems, dynamic properties are particularly pertinent to dynamic configurations by allowing us to quantify over evolving configurations. One finds dynamic properties, for instance, in security-related intrusion detection scenarios [29], where suspicious user behaviour can only be learnt at runtime after observing the system to learn what typical behaviour looks like. The third property presented in section 3.3 can also be considered dynamic, since the choice of contributing doctor to monitor depends on (i) the patient, and (ii) supervision information (*i.e.*, which doctor is responsible for which patient) stored at the backend. Clearly, although dynamic properties are more expressive than their static counterparts, the necessary machinery for dynamic instrumentation (section 2.4.1) is considerably more complex, and once more should only be considered when necessary.

These criteria lead to the possibility of four distinct *instrumentation strategies* for distributed monitoring, namely (i) *static orchestration*, (ii) *static choreography*, (iii) *dynamic orchestration* and (iv) *dynamic choreography*. The choice of approach often depends on necessity *i.e.*, depending on both underlying system characteristics and its properties worth verifying. We next consider each approach.

3.4.1 Static Orchestration

Conceptually the simplest approach, static orchestration involves employing a central monitor overhearing information over the communication medium, and verifying a set of pre-determined properties. This approach is evidenced in [10], where web service compositions implemented in BPEL [69] are monitored in orchestrated fashion. Advantages with this approach include (i) its simplistic nature, both in concept and in application, and (ii) its applicability when monitoring properties dealing with public information over the communication medium. However, static orchestration admits prevalent issues discussed above. Namely, static orchestration may lead to data exposure, poses a security risk, could also potentially result in unreasonable bandwidth overhead and is also incapable of handling dynamic properties (hence, no dynamic configurations).

Although a statically orchestrated approach is not applicable to the hospital management scenario in section 3.3 due to its dynamic architecture, we identify other applicable settings. Consider the example system presented in [10], entailing a virtual shop which interacts with an online bank offering its services through a web service. The shop periodically makes a request for a transaction on the client's behalf, which is acknowledged by the bank upon completion. Hence, we would like to verify the property that each transaction request is given a response within a given time frame. Given that both the shop and the bank components are known a priori (and remains unchanged), this system admits a static configuration. A statically orchestrated approach suffices in the case of the above property, since a central monitor can pair transaction requests with bank acknowledgements. Nevertheless, the requirement of an overhearing monitor exemplifies our understanding of data exposure, since sensitive bank information is exposed to a remote central location.

3.4.2 Static Choreography

Static choreography involves breaking down specifications into parts which can be monitored locally to subsystems, occasionally synchronising between monitors only when necessary, as seen in Fig. 3.4.

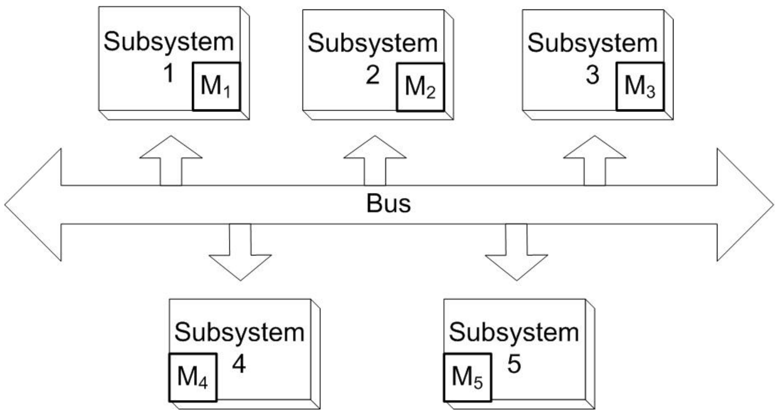


Figure 3.4: A static choreography-based approach.

Current static choreograph based approaches include [80, 77, 57, 64, 88]. By monitoring locally, statically choreographed monitors can avoid data exposure. Note that monitor interactions for synchronisation purposes do not denote exposure. Moreover, localised monitors reduce bandwidth overheads by eliminating the need to transfer event information to the central monitor. Finally, we reduce security risks related to central attack points.

On the other hand, given that monitor distribution occurs once a priori to system execution, this implies that a static choreography based approach is not resilient to dynamic configurations. More specifically, all new nodes added *after* the start of execution remain unmonitored. Even worse, nodes present at the start of computation, but which terminate at some point also halt their local monitoring effort, possibly invalidating the global monitoring framework as a result. By extension, statically choreographed monitors also cannot handle evolving system properties, or properties learnt at runtime.

Consider the hospital management scenario, and its related properties (section 3.3). Given that both patients and doctors are logging in and out of the system at runtime, it is unclear where to place localised monitors. Moreover, patient requests arrive at random, and verifying for instance the third property (no record is released unless approved by supervising doctors) is dependent on the system's current state (*i.e.*, which doctors are currently marked as supervising the patient). This makes it very hard to instrument an appropriate monitoring framework a priori, since knowledge of both the patient request (happening at the dynamic patient node) and the respective doctors whose approval we are required to verify can only be learnt at runtime. On the other hand, we can feasibly monitor the virtual shop scenario presented in the previous section. We can for instance adopt two local monitors — one at the shop and another at the bank — such that as soon as a transaction request is monitored at the shop location, the former monitor signals to the latter to check for a corresponding acknowledgement. If it arrives within the specified time frame, the property is satisfied, else it is violated. Note that by localising monitors we have both reduced associated overheads, as well as avoiding exposure of sensitive transaction information to another location.

3.4.3 Dynamic Orchestration

Dynamic orchestration-based approaches involve the adoption of a central monitor remotely observing sub-system behaviour, which however allows for the monitoring of dynamic properties too; possibly by adopting a dynamic instrumentation technique (section 2.4.1). An instance of dynamic orchestration is seen in [1], involving the centralised monitoring of web services against BPMN work flow specifications [16, 4]. Moreover, this approach allows for the deployment of the verification of contracts (representing system properties) *on-the-fly*, discovered or made known at runtime. However, the approach in [1] exemplifies one of the main disadvantages with dynamic orchestration; since subsystem events which contracts depend on are not known beforehand, subsystems typically send to the central monitor all information which may *potentially* be required during the verification of contracts instantiated at runtime, leading to significant inefficiencies. In other words, not only is substantial volumes of information sent to the monitor (leading to bandwidth overheads), but this overhead is often unnecessary, since only a subset of sent information shall be eventually referred to during monitoring.

In general, the main advantage of dynamic orchestration over its static counterpart is the capability to handle dynamic properties. For instance, using a dynamically orchestrated approach we can monitor the properties described in section 3.3. One could install a central monitor which instantiates the first property on *each* patient request, and checks that the medical record given as a response (if at all) by the backend always belongs to the same patient. However, although the above strategy works, it leads to data exposure, since medical records are sent remotely to the central monitor. One could easily anticipate that hospital management would disapprove of the transfer of their records. In conclusion, dynamic orchestration still suffer from the same drawbacks (bandwidth overhead, data exposure *etc.*) of more traditional orchestrated approaches.

3.4.4 Dynamic Choreography

Dynamic properties are the key to monitoring systems whose architecture evolves during execution. Through information learnt at runtime, we can instantiate and verify new properties accordingly. This can be rather easily achieved in an orchestrated setting, since necessary monitoring occurs at one central location. However, achieving the same machinery through a choreographed approach is considerably more challenging. The major problem with dynamic choreography is that property *decomposition* and *redistribution* must occur at runtime. To the best of our knowledge, presently no existing monitoring framework falls in this category.

To this effect, we propose the study of dynamic choreography through the use of a *migrating monitor approach i.e.,* employing monitors running locally to subsystems where confidential information lies, before physically migrating to other locations when their behaviour becomes pertinent to the system's global correctness, as depicted in Fig. 3.5.

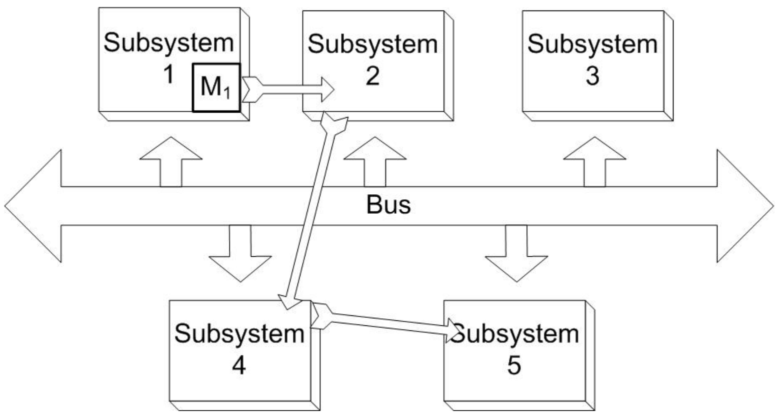


Figure 3.5: A migrating monitor approach.

Note that migrating monitors describe a choreographed approach, since (i) verification occurs locally, and (ii) monitor placement is data driven. However, note that by adopting migration as a framework primitive, we are *also* able to redirect monitoring effort at runtime as necessary. In other words, we can choose which locations are worth verifying based on information obtained at runtime. This leads to a monitoring approach which is dynamically *re-distributable on the fly*, without the need for recompilation. By extension, new locations are also at the reach of the monitoring framework. In effect, we believe migrating monitors to be tolerant to dynamic architectures, while admitting advantages pertaining to choreography-based approaches *i.e.,* minimising bandwidth overhead and avoiding data exposure.

Consider once more the third property of the hospital management scenario. This property can be easily verified through a migrating monitor approach. On each patient request, a monitor situated at the backend retrieves necessary doctor-patient dependencies. This information is used to subsequently redirect a migrating monitor to each doctor responsible for that patient.

Each monitor listens for an event denoting a doctor's disapproval; if it occurs (at any doctor location) this monitor synchronises with the other monitor at the backend, in order to listen for a subsequent record release event. If this event does indeed occur this implies that the property has been broken, since the record would have been released even after a doctor's order to the contrary. Note that through this instrumentation strategy, all necessary verification is done locally, thus avoiding exposure.

We argue that migrating monitors are best implemented through a property agnostic approach (section 2.4.1), such that each sub-system exposes a local alphabet a events which can be later analysed by monitors. Through this approach, we can also verify properties learnt at runtime. However, necessary instrumentation dynamicity can also be seen as a disadvantage of migrating monitors, in that nodes need to be willing to install monitors known only at runtime. This point leads to a question of *trust*; system administrators are to trust external monitors which (i) migrate to and from the system as necessary, while (ii) having access to confidential information, as well as (iii) potentially altering the underlying system's execution.

These requirements may clearly be a step too far in certain scenarios. For instance, one might question the *integrity* of the monitors themselves *i.e.*, who is to say that monitors are not exporting local information to external entities? Additionally, there are no guarantees that monitors (possibly coming from untrustworthy parties) do not import and execute malicious code, damaging the system in the process. To get around this problem, one could require monitors to be *encrypted* in such a way that each location can only view the monitoring effort intended for that subsystem. Another interesting approach could involve *proof-carrying code*, in order to provide assurances of the monitor's trustworthiness. Finally, another issue with migrating monitors is the more involving machinery required for their implementation; dynamic instrumentation coupled with code mobility can be expensive. We shall return to both issues in chapter 7.

In conclusion, migrating monitors offer an alternate approach to distributed monitoring, whose applicability is most rewarding when facing (i) strong information confidentiality restrictions (ii) in a highly dynamic environment. Conversely, applying a migrating monitor approach absent either precondition would result in an unnecessarily complex monitor instrumentation approach.

3.5 Extracting A Temporal Order on Remote Events

Section 3.2 described our interest in distributed systems which lack a global clock (or equivalently, local synchronised clocks). This design choice mirrors real-life distributed system implementations, where perfect synchronisation is unattainable. The work presented by Lamport [58] showed that, given this setting, the best we can do with is extract a *partial order* on events across locations. To this effect, he formalised the notion of the *happened-before relation*, exploiting interactions amongst distributed sub-systems to define a temporal relationship amongst remote events. More precisely, this relation first states that local events are ordered totally *wrt.* their *local timestamp*. Moreover, a send event at one location and the corresponding receive event at another (location) implies that the former must have happened before the latter. Finally, the happened-before relation is transitive, in that if event e_1 happened before e_2, with e_2 happening before e_3, then e_1 happened before e_3.

Definition 3.5.1. *(Happened-Before Relation) The happened-before relation, denoted by $\leq$, is defined as the smallest relation that satisfies the following properties*

1. *If e_i happened before e_j locally, then $e_i \leq e_j$.*
2. *If e_i^k is a* send *event at location k, with e_j^l being the corresponding* receive *event at l, then $e_i^k \leq e_j^l$.*
3. *If $e_i \leq e_j$ and $e_j \leq e_k$ then $e_i \leq e_k$.*

In general, although distributed systems admit an inherent total ordering on events, the best we can do is infer a partial order using runtime monitors, such as that implied by the happened-before relation. This view can be modelled through a space-time diagram, exemplified in Fig. 3.6. Moreover, given that a sub-system can only *affect* another by interacting with it, this happened-before relation represents the notion of a *causal ordering* on events [58, 35]. Informally, given any two events pertaining to a happened-before relation, the former is said to *possibly* be the *cause* of the latter (or they are unrelated). Conversely, the latter is (possibly) the *effect* of the former. Numerous structures have been attempted in the literature which help during inference of the happened-before relation, including *Lamport Timestamps* [58] and *Vector Clocks* [35]. Both adopt the use of simple counters (acting as a logical clock) in order to keep track of logical orderings on events.

Irrespective of the monitoring algorithm adopted, as designers of a distributed monitoring framework it is our responsibility to extract (i) the (total) temporal order of events at a local level, and (ii) as many temporal relationships between remote events as possible. From a monitoring perspective, defining machinery which implements *Def*$^{\underline{n}}$ 3.5.1 involves installing a monitor at both ends of system interactions, listening to communication as it occurs and recording causal orderings as necessary (possibly through the use of logical clocks). This approach is exemplified in [80]. However, we believe this technique is only suitable for statically choreographed approaches which install fixed monitors at *each* location a priori. What happens with newly added locations? Clearly, given that no monitor is present at these new locations, we cannot infer any order on their events. In general, this implies that extracting a temporal order through system interactions is not ideal for systems admitting dynamic configurations, which periodically require re-distribution.

We hence recognise the need to extract temporal relationships amongst remote events in a different way. This is presented by the *monitored-before relation*; an adaptation of the happened-before relation which extracts a partial order by instead *exploiting monitor execution*. More specifically, we propose the exploitation of the migration act's sequential nature; events monitored by the system at the previous location must have happened before those analysed at the new location (after migration). In case where migration is not necessary, we can also exploit monitor synchronisation over channels to infer a temporal succession of events. Once more, this relation assumes a total ordering on events *per location*, extracted through the local clock. Hence, monitoring events locally mirrors this ordering.

Definition 3.5.2. *(Monitored-Before Relation) The monitored-before relation, denoted by $\leq_M$, is defined as the smallest relation that satisfies the following properties*

1. *If e_i is locally observed before e_j, then $e_i \leq_M e_j$.*
2. *If a monitor observes e_i^k at k, subsequently migrates to/synchronises with a monitor at l, which later observes e_j^l, then $e_i^k \leq_M e_j^l$.*
3. *If $e_i \leq_M e_j$ and $e_j \leq_M e_k$ then $e_i \leq_M e_k$.*

Admittedly, it is possible for the monitored-before relation to be unfruitful, in that we are dependent on the monitor *keeping up* with the system's execution. In other words, it is conceivable that the monitor is too slow to extract necessary temporal orderings, since by the time the monitor migrates/synchronises, other pertinent events would have already happened. However, what we lose in precision we gain in generality; whereas $\leq$ depends on the underlying system execution presented to the monitor — which is (mostly) applicable to a statically choreographed approach — through $\leq_M$ responsibility of extracting a temporal ordering falls on the monitor's execution. By extension, we can now define various instrumentation approaches to extract required information. We shall define an example mechanism which implements *Def$^{\underline{n}}$* 3.5.2 in chapter 5.

Consider the scenario depicted in Fig. 3.6, describing event sequences at locations *k* and *l*, such that a monitor *eventually* migrates from *k* to *l* taking an arbitrary duration (depicted by an arrow).

Clearly, although e_2^k precedes e_3^l, the temporal ordering inferred by migration does not capture this relation. The area under the arrow denotes the imprecision of *Def$^{\underline{n}}$* 3.5.2; any events which occur within that timespan are ignored. One can strive to minimise this duration by employing faster communication mediums/optimising monitors, however this issue cannot be completely factored out. Even worse, one cannot infer its precise duration due to asynchrony amongst remote clocks. Consider for instance the monitoring strategy for dynamic choreography described in the previous section. Clearly, by the time the monitor migrates to the doctor's location it might have been too late, since the doctor would have already rejected the patient's request without the monitor knowing.

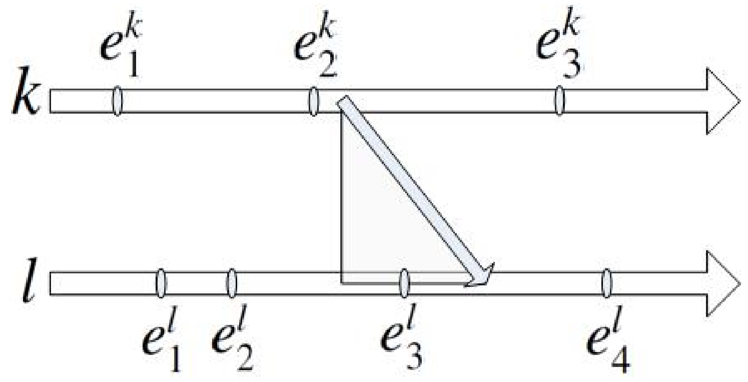


Figure 3.6: An example space-time diagram

In conclusion, due example scenarios such as that depicted in Fig. 3.6, this implies that we shall at most aim for a *sound* monitoring framework (*i.e.*, a framework where all reported violations are true). On the other hand, we shall not strive for *completeness* — we recognise that certain violations will go uncaught. We believe this to be an inherent limitation of distributed architectures, and is an issue we shall not study further throughout this dissertation.

3.6 Conclusions

This chapter motivated issues relevant to the design of a distributed runtime verification framework. More specifically, we saw the increased complexities of installing a monitoring framework in a distributed setting, involving issues such as information exposure, bandwidth overheads and configuration dynamicity. This lead to the definition of a broad taxonomy on distributed monitoring approaches, identifying scenarios where each approach is best applied. Moreover, we introduced the novel migrating monitor approach, employing monitors which monitor locally, and migrate remotely when necessary. This chapter also described the difficulty in achieving a temporal ordering on remote events. This lead us to the definition of an approach which infers temporal orderings through monitor execution. We shall return to the issues and techniques introduced above in chapter 5, where we propose a general distributed monitoring framework.

4. The π-calculus

The following chapter presents an overview of the *π-calculus*; a formalism concerned with the description of *concurrent processes* communicating through *message passing* techniques. This language is particularly appealing due to its capability to describe *evolving configurations*. Section 4.1 motivates the use of the π-calculus within the context of distributed monitoring scenarios. Section 4.2 presents an overview of the calculus' syntax, and defines an action semantics. This is followed by sections 4.3 and 4.4, which present notions of structural and behavioural equivalence for the language. Finally, section 4.5 concludes the chapter.

4.1 Motivation For π

We recognise the need to supplement our interest in distributed monitoring with a more formal investigation. In turn, this will allow us to *precisely* study the scenario's *capabilities*, as well as its *limitations*. Our thoughts hence turn to the choice of formalism for the description of required scenarios — a crucial choice since it shall dictate what can be described during our formal analysis. To the best of our knowledge, no formalism exists which *explicitly* formalises notions of distributed monitoring described in chapter 3. We therefore aim to to extend an existing formalism in order to satisfy our requirements. Hence, our immediate task becomes that of choosing an appropriate formalism which serves as a suitable basis.

The choice of some *process calculus* is immediately appealing. Process calculi represent the class of formalisms concerned with the modelling of *concurrency* and *communication*, and are frequently used for *modelling* distributed systems. Initial attempts include calculi such as CSP [49] and CCS [66]. Both languages are very good at structurally describing concurrent processes, which communicate through channels (*i.e.,* via *message passing* [28] techniques). These channel links serve as a description of the system's network topology, and remains unchanged during computation (referred to as a *static* topology). Although successful, both approaches are however unsuitable for our requirements. Their main limitation lies with their static communication topology, unsuitable for modelling modern systems which are highly *dynamic*, especially with the proliferation of the internet. *Dynamic systems* involve systems which can grow, shrink and move about during execution [67]. Hence, system dynamicity can take two forms; (i) in one form, it is the *channel links* which move in a space of concurrent processes, such that it is the *communication topology* which evolves; (ii) another form refers to *process mobility*, describing the capability of processes to move within some computational space.

To this effect, we turn to the *π-calculus* [76, 46, 67], a formalism used for describing *communicating mobile systems*. The π-calculus was developed as an extension to CCS, enhancing its capabilities with the ability to describe systems whose structure evolves during execution (hence the term *mobile*). The core calculus admits two entities; *processes* and *channels*. Process represent *computational entities*, of which a finite number execute in parallel within a given system. On the other hand, channels represent a *communication link* between processes. The crux of the approach lies with the treatment of *channels as names*; used *both* during communication and as data that processes exchange. This implies that a channel name received during one interaction can be actively used as a communication medium in another. By receiving channel information, a process acquires the capability to interact with other processes previously unknown to it. Hence, a system's structure *wrt.* its communication links can evolve over time, in *unforeseeable* ways.

We believe the choice of the π-calculus to be a natural one, due to its inherent expressivity as well its potential for extension, as argued in [76]. Moreover, the same text also proves that the π-calculus is at least as expressive as the λ-calculus [11], taking *interaction* as its primitive. This implies that we are dealing with a universal model of computation — our job is to subsequently render *explicit* required distributed monitoring concepts, rather than increase the calculus' expressivity. The following chapter serves as necessary background for an understanding of the core π-calculus.

4.2 The Calculus

The following section presents a (i) *first-order*, (ii) *synchronous*, and (iii) *polyadic* π-calculus variant. By first-order we refer to the capability of processes to pass on channel names and other basic data types during communication (as opposed to *higher-order calculi* [76]). Synchronous communication requires the sender to wait for acknowledgement of receipt at the receiving end before continuing execution. Finally, a polyadic π-calculus allows for the transfer of value tuples over channels, as opposed to *monadic* variants which permit transfer of one value at a time.

4.2.1 Syntax

The syntax presupposes denumerable set of names $c, n, m \in \textsc{Chans}$, variables $x, y, z \in \textsc{Vars}$ and basic values $a, b \in \mathrm{BV}$. Channel names can be thought of as *references* to communication links. Moreover, knowledge of said names can be transferred between processes, and referred to during further interactions. Variables act as *information placeholders*, whose content is updated throughout computation. These variables can either contain (i) a channel name or (ii) a basic value. The set of basic values BV refers to an unspecified collection of strings, integers, booleans *etc.* and represents possible information transfer not involving channel information. We shall consider *identifiers* $u, v \in \textsc{Idents} = \textsc{Chans} \cup \textsc{Vars}$ *i.e.*, ranging over channels and variables. Lists of identifiers $v_1, \ldots, v_n$ are denoted as $\bar{v}$, with lists of variables analogously written as $\bar{x}$. Processes $P, Q \in \textsc{Proc}$ encode concurrent computation in the π-calculus; their syntax is inductively defined in Fig. 4.1.

$$P, Q \quad ::= \quad \mathsf{stop} \mid u!\bar{v}.P \mid u?\bar{x}.P \mid P \parallel Q \mid \mathsf{if}\ u{=}v\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid *P \mid \mathsf{new}\, c.P$$

Figure 4.1: The π-calculus

We informally interpret each process as follows.

- Process stop is a terminal process, and does nothing.
- Process $u!\bar{v}.P$ outputs tuple $\bar{v}$ on u, and proceeds as dictated by P (on completion of transfer). The use of identifier u implies that output can either occur directly by referring to a channel name, or indirectly through a variable containing a channel reference.
- $u?\bar{x}.P$ denotes a process which accepts a tuple $\bar{v}$ on u, and *substitutes* values $v_i \in \bar{v}$ for each instance of $x_i \in \bar{x}$ in P. Computation proceeds as dictated by the resulting process (after substitution). Note that tuples $\bar{v}$ and $\bar{x}$ are required to admit the same *arity*, in order for the input action to make sense.
- Process $P \parallel Q$ composes processes P, Q in parallel. Both P and Q are taken to execute concurrently, occasionally communicating through some shared name.
- $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is the matching operator, and acts as a test on the identifier tuples. The composite process subsequently decides on how to proceed depending on the outcome of the test, thus operating as P if the test succeeds, or as Q otherwise.
- $*P$ defines recursive processes, and loosely behaves as unbounded number of copies of P composed in parallel. Hence, this operator is crucial when describing infinite computations.
- $\mathsf{new}\, c.P$ acts as a scoping mechanism for channel names, such that knowledge of c is restricted to P. Conversely, P is *s.t.b.* the scope of c. Channel c can hence be internally used for communication within P, but is unknown to outside processes.

We shall refer to processes $u!\langle\rangle.P$, $u?\langle\rangle.P$ when channel communication occurs for *synchronisation purposes* i.e., no information is passed over the channels. The π-calculus derives its strength for describing *evolving configurations* from its treatment of channels as data. More specifically, by allowing for the transfer of channel names, their scope can evolve during execution. In other words, knowledge of name c can be *extruded* to alternate processes at runtime. Through this newly acquired information, these processes can now interact with other entities previously unaware of each other's presence. In general, knowledge of c is received as part of a variable tuple $(c = v_i) \in \bar{v}$, and is eventually used during output $x_i!\bar{v}.P$ or input $x_i?\bar{y}.P$, by substituting c for x_i.

Binders

Variables in the π-calculus can be either *free* or *bound* in the standard manner. Variable instantiation through binding is necessary in order for terms to make sense. Input operator $u?\bar{x}.P$ acts as a variable binder, where variables $x_i \in \bar{x}$ are *s.t.b.* bound in P. On the other hand, free variables describe a system's capability for action; for P to send x, to send via the name referred to by x, or to receive via x it must be the case that x is free [76]. All variables which are not bound in a term are *s.t.b.* free. We shall use notation $\mathsf{fv}(P)$ to denote the set of free variables in P, and $\mathsf{bv}(P)$ to denote those which are bound.

Example 1. Consider term $P_1 \triangleq (c_1?(x_1, x_2).x_3!(x_1, x_2).\mathsf{stop}) \parallel \mathsf{new}\, c_2.(x_4!c_2.\mathsf{stop})$. The set of free and bound variables work out in this case to

$$\mathsf{fv}(P_1) = \{x_3, x_4\}$$

$$\mathsf{bv}(P_1) = \{x_1, x_2\}$$

■

Given the finite description of P, it can be proven by structural induction that $\mathsf{fv}(P)$ and $\mathsf{bv}(P)$ are finite [76].

Definition 4.2.1. *(Closed term) A closed term is defined as a process P with no free variables i.e., $\mathsf{fv}(P) = \emptyset$. Intuitively this implies that a closed term is a process which does not need any of its variables to be bound in order to obtain meaning.*

The π-calculus also makes use of *name* binders, with $\mathsf{new}\, c.P$ implying that name c is bound in P. Name binding is used throughout the calculus to specify channel *scope*, such that if c is known to P then this process can make use of the channel to communicate. Moreover, processes may eventually be bound to additional names during execution by scope extrusion and/or the creation of new channels. Like variables, channels which are not bound in a name are *s.t.b.* free. We use notation $\mathsf{fn}(P), \mathsf{bn}(P)$ to denote the set of free and bound names in P. Therefore $\mathsf{fn}(P_1)$ and $\mathsf{bn}(P_1)$ work out to

$$\mathsf{fn}(P_1) = \{c_1\}$$

$$\mathsf{bn}(P_1) = \{c_2\}$$

respectively.

Substitution

A consequence of variable binders is the notion of *substitution*, written

$$P\{{}^{v}\!/\!_{x}\}$$

to denote that identifier v is substituted for *free* instances of variable x in P. This notation is extended to tuples, written $P\{{}^{\bar{v}}\!/\!_{\bar{x}}\}$, representing a sequence of operations $P\{{}^{v_i}\!/\!_{x_i}\}$ for each (v_i, x_i) pair in $(\bar{v}, \bar{x})$. Hence, substitution on tuples only makes sense if *both* the identifier and variable tuples are of the same *arity*. Intuitively, substitution is necessary to assign a valuation to a variable placeholder in P; its use is apparent when passing on received information during input operation ? to the residual system (after communication). Note that in order for $S\{{}^{v}\!/\!_{x}\}$ to be well-defined, v must not be captured by any of the binding constructs ?, or new. If this is not the case, we resolve the clash by *renaming* the capturing variable's binding scope to a *fresh* variable, *i.e.*, one which is neither free nor bound in P. Variable renaming is performed by substituting a free variable name with another.

Example 2. Consider process $P_2 \triangleq c?\langle x_1, x_2\rangle.x_1!\langle x_2\rangle.\mathsf{stop}$ which receives tuple $\langle x_1, x_2\rangle$ on c, and subsequently sends x_2 on x_1 (thus assuming x_1 is substituted by a channel name). Now consider placing $P_3 \triangleq c!\langle d, 1\rangle.\mathsf{stop}$ in parallel with P_2 *i.e.*, $P_2 \parallel P_3$; communication can now occur over c. Process P_2 hence substitutes tuple $\langle d, 1\rangle$ for $\langle x_1, x_2\rangle$ *i.e.*, $(x_1!\langle x_2\rangle.\mathsf{stop})\{{}^{\langle d,1\rangle}\!/\!_{\langle x_1,x_2\rangle}\}$, which evaluates to $d!\langle 1\rangle.\mathsf{stop}$. Hence, through substitution we successfully modeled the information transfer of $\langle d, 1\rangle$ from P_3 to P_2. ■

We use notation $P\sigma$ to refer to substitution in its most general form; see *Def$^{\underline{n}}$* 4.2.2. Hence, previous notation $P\{{}^{v}\!/\!_{x}\}$, $P\{{}^{\bar{v}}\!/\!_{\bar{x}}\}$ are instances of $P\sigma$. Function $\sigma :: \textsc{Vars} \rightarrow \textsc{Idents}$ is defined as a partial function from variables to identifiers, mapping variables to specific values. Note that the definition below assumes renaming of bound variables in case of unintended variable capture. Moreover, overloaded notation $v\sigma$ on identifiers returns v when $v \notin dom(\sigma)$, and $\sigma(v)$ otherwise. This notation is extended to identifier lists. Finally, operator $S \trianglelefteq R$ denotes *domain co-restriction* [82] of relation R to elements not in set S *i.e.*, if $R :: X \leftrightarrow Y$, $S :: X$ then $S \trianglelefteq R \triangleq \{x : X, y : Y \mid x \notin S \;\wedge\; (x, y) \in R \bullet (x, y)\}$.

Definition 4.2.2. *(Substitution $P\sigma$) We define substitution of σ on $P \in$ Proc, written $P\sigma$, as follows*

$$P\sigma \triangleq \begin{cases} \textsf{stop} & P = \textsf{stop} \\ ((P_1\sigma) \parallel (P_2\sigma)) & P = (P_1 \parallel P_2) \\ (u\sigma)!(\bar{v}\sigma).(P_1\sigma) & P = u!\bar{v}.P_1 \\ (u\sigma)?\bar{x}.(P_1\sigma') & P = u?\bar{x}.P_1 \wedge (\sigma' = \bar{x} \trianglelefteq \sigma) \\ P = \textsf{if}\,(u\sigma)=(v\sigma)\ \textsf{then}\ (P_1\sigma)\ \textsf{else}\ (P_2\sigma) & P = \textsf{if}\ u=v\ \textsf{then}\ P_1\ \textsf{else}\ P_2 \\ *(P_1\sigma) & P = *P_1 \\ \textsf{new}\,c.(P_1\sigma') & P = \textsf{new}\,c.P_1 \wedge (\sigma' = \{c\} \trianglelefteq \sigma) \end{cases}$$

Note that the result of substitution may not always be a well defined term — we may for instance substitute a basic value for a variable which is to be used as a channel. However, we shall avoid considering such scenarios throughout the text. The interested reader is pointed to [76, 47] for a discussion of type systems which help in disallowing such scenarios.

α-Equivalence

Variable renaming through substitution gives us a notion of *α-equivalence* for π-calculus terms.

Definition 4.2.3. *(α-equivalence) Two systems P_1,P_2 are said to be α equivalent, written $P_1 \equiv_\alpha P_2$, if we can obtain P_2 from P_1 in a finite number of variable renaming operations, and vice versa.*

In other words, two systems are deemed α equivalent if they are the same, except in their use of bound variables [46]. Below are a few examples of α-equivalent systems.

$$\begin{array}{rcl} \textsf{new}\,c.c!1.\textsf{stop} & \equiv_\alpha & \textsf{new}\,d.d!1.\textsf{stop} \\ \textsf{new}\,c.(c!1.\textsf{stop} \parallel c?x.d!x.\textsf{stop}) & \equiv_\alpha & \textsf{new}\,e.(e!1.\textsf{stop} \parallel e?x.d!x.\textsf{stop}) \\ \textsf{new}\,c.(c!1.\textsf{stop} \parallel c?x.d!x.\textsf{stop}) & \equiv_\alpha & \textsf{new}\,e.(e!1.\textsf{stop} \parallel e?y.d!y.\textsf{stop}) \end{array}$$

This preliminary notion of equality makes sense, since the name of variable placeholders should not affect the process' behaviour. Sections 4.3 and 4.4 provide more refined notions of equality, pairing processes on notions of *structural equivalence*, and finally a notion of equality based on exhibited *behaviour*.

Contexts

We shall often be interested in considering π-calculus terms as part of some larger environment, also known as a *context*. As we shall see in section 4.2.2, reasoning about *processes-in-context* give us the most profound understanding of their operation, by also considering their interaction with the external surroundings. A context is informally considered a 'process with a hole' of the form $P \parallel \_$, where $\_$ can be replaced with any valid process. Moreover, although we are interested in reasoning about processes when running in parallel with other entities, we also include an additional context $\textsf{new}\,c.\_$ for coinductive reasons [76, 67] — more below. The property of *contextuality* serves as a means for formally reasoning about contexts (*Def$^{\underline{n}}$* 4.2.4).

Definition 4.2.4. *(contextual relation) A relation $\mathcal{R}$ :: $\textsc{Proc} \leftrightarrow \textsc{Proc}$ is s.t.b. contextual if it is preserved by operators $\|$ and* new*, hence adhering to the following two properties:*

- $(P_1 \;\mathcal{R}\; P_2) \Rightarrow (((P_1 \parallel Q) \;\mathcal{R}\; (P_2 \parallel Q)) \wedge ((Q \parallel P_1) \;\mathcal{R}\; (Q \parallel P_2)))$
- $(P_1 \;\mathcal{R}\; P_2) \Rightarrow (\mathsf{new}\, c.P_1 \;\mathcal{R}\; \mathsf{new}\, c.P_2)$

A relation is *s.t.b.* contextual if it is preserved by contexts. Contextuality is either included as part of a relation's definition, or subsequently proven as a property.

4.2.2 Action Semantics

The following section defines an *action semantics* for the π-calculus [47, 76, 67]. This choice allows us to obtain a more general view of process behaviour, describing process computation both *internally* as well as when placed within a *larger computational environment* (*i.e.*, its context) [47]. In turn, this allows for the *compositional analysis* of process behaviour, allowing us to consider a process as the sum of its constituent parts' execution. We model this view of process behaviour through a *Labelled Transition System* (LTS), presented in *Def$^{\underline{n}}$* 4.2.5, describing a process' *capability for action* at each step in its computation.

Definition 4.2.5. *(LTS) A labelled transition system is defined as a triple $(\mathcal{S},\ Act,\ \xrightarrow{a})$ such that*

- *A set of states/configurations $\mathcal{S}$;*
- *A set of action labels Act;*
- *A next state transition relation $\xrightarrow{\alpha}$ for each $\alpha \in Act$.*

Given an LTS representation for a π-calculus term, its states refer to intermediate configurations (*i.e.*, also processes) adopted by the term during its execution. The set of action labels represent the process' capabilities to *send* or *receive* messages from its external environment, as well as the capability for internal computation. We shall henceforth use transition $P \xrightarrow{\alpha} Q$ to specify that process P evolves to residual Q, affecting action α in the process. More specifically, processes *compute* by performing one of three transitions;

- $P \xrightarrow{c?\bar{d}} Q$; the ability of process P to receive tuple $\bar{d}$ on c, evolving to residual process Q in the process.
- $P \xrightarrow{(\bar{b})c!\bar{d}} Q$; the capability of P to transmit tuple $\bar{d}$ on c, afterwards evolving to Q. Moreover, this judgement encodes the ability of P to *export* knowledge of bound names $\bar{b}$ *during* channel output, such that $\bar{b} \subseteq \bar{d}$ and $c \notin \bar{b}$. In other words, we restrict the exporting of names to those which are part of the transferred tuple in order to avoid unintended capture of free names.
- $P \xrightarrow{\tau} Q$; denoting an *internal* action executed by P, evolving to Q in the process.

The use of τ describes a silent action performed internally by subcomponents in P. We shall therefore also combine the first two judgements to extract τ, in order to describe internal communication within P.

Figure 4.2 presents the calculus' transition rules, defined over closed terms *i.e.,* $\mathsf{fv}(P) = \emptyset$. Through these rules we are able to describe process behaviour by extracting its LTS representation. Crucially, by adopting an LTS view (of process behaviour) we also obtain standard *coinductive equational reasoning* through a notion of *bisimilarity* [67, 75, 47, 76] (section 4.4).

Notation 4.1. *Overloaded notation* $\mathsf{fn}(\alpha)$ *and* $\mathsf{bn}(\alpha)$ *is taken to represent the set of free names and bound names in action* α*. If* α *is an input label, then the set of bound names is empty. However, if* α *is an output label of the form* $(\bar{b})\alpha_o$*, then the set of bound names is* $\bar{b}$*. Names which are not bound are s.t.b. free. Taking action* $\alpha = (\bar{b})c!\bar{d}$ *as an example;*

$$\begin{aligned}\mathsf{fn}((\bar{b})c!\bar{d}) &= \{c, \bar{d}\}/\bar{b}\\ \mathsf{bn}((\bar{b})c!\bar{d}) &= \bar{b}\end{aligned}$$

$$\textsc{P-In}\ \frac{}{c?\bar{x}.P \xrightarrow{c?\bar{v}} P\{\bar{v}/\bar{x}\}} \qquad \textsc{P-Out}\ \frac{}{c!\bar{v}.P \xrightarrow{c!\bar{v}} P}$$

$$\textsc{Com}\ \frac{P \xrightarrow{c?\bar{v}} P',\ Q \xrightarrow{(\bar{b})c!\bar{v}} Q'}{P \parallel Q \xrightarrow{\tau} \mathsf{new}\,\bar{b}.(P' \parallel Q')}\ [\bar{b} \cap \textsc{fn}(P) = \emptyset] \qquad \textsc{P-Open}\ \frac{P \xrightarrow{(\bar{b})c!\bar{v}} P'}{\mathsf{new}\,n.P \xrightarrow{(\bar{b}\cup\{n\})c!\bar{v}} P'}\ [n \in \bar{v}]$$

$$\textsc{P-Rec}\ \frac{}{*P \xrightarrow{\tau} P \parallel *P} \qquad \textsc{P-Eq}\ \frac{}{\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P}\ [v_1 = v_2]$$

$$\textsc{P-Neq}\ \frac{}{\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q}\ [v_1 \neq v_2]$$

$$\textsc{P-Cntx}_1\ \frac{P \xrightarrow{\alpha} P'}{(P \parallel Q) \xrightarrow{\alpha} (P' \parallel Q)}\ [\textsc{bn}(\alpha) \cap \textsc{fn}(Q) = \emptyset]$$

$$\textsc{P-Cntx}_2\ \frac{P \xrightarrow{\alpha} P'}{(Q \parallel P) \xrightarrow{\alpha} (Q \parallel P')}\ [\textsc{bn}(\alpha) \cap \textsc{fn}(Q) = \emptyset] \qquad \textsc{P-Cntx}_3\ \frac{P \xrightarrow{\alpha} Q}{\mathsf{new}\,c.P \xrightarrow{\alpha} \mathsf{new}\,c.Q}\ [c \notin \textsc{fn}(\alpha)]$$

Figure 4.2: An action Semantics for π

Rule (P-Out) describes a process' capability to *input* $\bar{v}$ on c, evolving to $P\{\bar{v}/\bar{x}\}$. Dually, rule (P-Out) describes the capability of $c!\bar{v}.P$ to *output* $\bar{v}$ on c, proceeding as P. These two capabilities subsequently interact as described through rule (Com) (here we elide its symmetric rule) to describe *process communication*. For processes to synchronise, we require corresponding input and output action labels to match on (i) the communication channel c, and (ii) the value tuple $\bar{v}$. Note that bound names $\bar{b}$ exported over the output action are now bound in residual process $\mathsf{new}\,\bar{b}.(P' \parallel Q')$. Hence, through communication the scope of channel names can be extended beyond their original configuration. Finally, note the side condition on $\bar{b}$ in order to avoid unwanted capture of free names.

Rule (Com) however only describes the *importing* bound channel information from output labels. In order to *export* names, we require rule (P-Open) which allows for the export of bound name n on the output label of a known transition. However, we are restricted by the need for n to be part of $\bar{v}$, implying that we can only export names which are currently being output by the process. In effect, (P-Open) implements the necessary mechanism for *scope extrusion*.

Rule (P-Rec) describes the unraveling of a recursive call, by executing a fresh copy of the program in parallel. Rules (P-Eq) and (P-Neq) describe behaviour of the branching operator; if the test on identifiers $v_1 = v_2$ returns true, the composite process executes as dictated by P, else it proceeds as Q. Finally, rules (P-Cntx$_1$), (P-Cntx$_2$) and (P-Cntx$_3$) allows for the deduction of process behaviour when placed in a context. The former two rules describe behaviour when placed in parallel with other processes, whereas the latter rule describes process behaviour when encapsulated within a channel scope. In all three cases side conditions ensure no unwanted capture of free names as a byproduct.

Example 3. Consider processes $P_3 \triangleq c_1!\langle c_2\rangle.\mathsf{stop}$, $P_4 \triangleq c_2?\langle\rangle.\mathsf{stop}$, $P_5 \triangleq c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop}$, such that

$$\mathsf{new}\,c_1.(\mathsf{new}\,c_2.(c_1!\langle c_2\rangle.\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop}) \parallel c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop})$$

i.e., $\mathsf{new}\,c_1.(\mathsf{new}\,c_2.(P_3 \parallel P_4) \parallel P_5)$. Given this initial configuration, process P_3 communicates over c_1 with P_5, exposing knowledge of channel c_2. This latter process subsequently synchronises with P_4 over c_2. Hence, by extruding knowledge of channel c_2, P_5 learns about P_4 (of which it was previously unaware), with which it subsequently synchronises. Let us describe this behaviour through the above rules. By rule (P-Out) we infer transition $c_1!\langle c_2\rangle.\mathsf{stop} \xrightarrow{c_1!\langle c_2\rangle} \mathsf{stop}$. Moreover, by (P-Cntx$_1$) and (P-Open) we infer

$$\mathsf{new}\,c_2.(c_1!\langle c_2\rangle.\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop}) \xrightarrow{(\langle c_2\rangle)c_1!\langle c_2\rangle} \mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop} \quad ...(i)$$

Now consider the execution of $c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop}$. By rule (P-In) we deduce

$$c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop} \xrightarrow{c_1?\langle c_2\rangle} x!\langle\rangle.\mathsf{stop}\{^{\langle c_2\rangle}/_{\langle x\rangle}\} \quad ...(ii)$$

Note the corresponding action labels for transitions (i) and (ii), which combine by rule (Com) to describe the interaction of P_3 and P_5 over c_1 *i.e.*,

$$\mathsf{new}\, c_2.(c_1!\langle c_2\rangle.\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop}) \parallel c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop} \xrightarrow{\tau} \mathsf{new}\, c_2.(\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop} \parallel c_2!\langle\rangle.\mathsf{stop})$$

The scope of c_2 has therefore been extended to the residual of P_5. We next add the scope for channel c_1 through rule (P-CNTX$_3$), resulting in

$$\begin{aligned}\mathsf{new}\, c_1.(\mathsf{new}\, c_2.(c_1!\langle c_2\rangle.\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop}) \parallel c_1?\langle x\rangle.x!\langle\rangle.\mathsf{stop}) &\xrightarrow{\tau} \\ &\mathsf{new}\, c_1.(\mathsf{new}\, c_2.(\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop} \parallel c_2!\langle\rangle.\mathsf{stop}))\end{aligned}$$

which describes the first computational step taken by the original configuration. Sub-process $\mathsf{new}\, c_2.(\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop} \parallel c_2!\langle\rangle.\mathsf{stop})$ can now perform internal communication over c_2. Rule (P-OUT) dictates output $c_2!\langle\rangle.\mathsf{stop} \xrightarrow{c_2!\langle\rangle} \mathsf{stop}$, whereas (P-IN) describes process input $c_2?\langle\rangle.\mathsf{stop} \xrightarrow{c_2?\langle\rangle} \mathsf{stop}$. These latter two transitions combine by (COM) to give $c_2?\langle\rangle.\mathsf{stop} \parallel c_2!\langle\rangle.\mathsf{stop} \xrightarrow{\tau} \mathsf{stop} \parallel \mathsf{stop}$. By rules (P-CNTX$_2$) and (P-CNTX$_3$) twice we infer

$$\mathsf{new}\, c_1.(\mathsf{new}\, c_2.(\mathsf{stop} \parallel c_2?\langle\rangle.\mathsf{stop} \parallel c_2!\langle\rangle.\mathsf{stop})) \xrightarrow{\tau} \mathsf{new}\, c_1.(\mathsf{new}\, c_2.(\mathsf{stop} \parallel \mathsf{stop} \parallel \mathsf{stop}))$$

Through the above transitions we have hence described the required computation, where knowledge of c_2 was transferred on c_1, which was in turn used during an eventual interaction. ■

4.3 Structural Equivalence

The syntax described in Fig. 4.1 is too discriminating; it forces us to *syntactically distinguish* between terms which can be considered equivalent. To this effect, we introduce the notion of *structural equivalence* (*Def$^{\underline{n}}$* 4.3.1), which allows us to abstract over *inessential details of process structure* without affecting its meaning. Our understanding of structural equivalence is defined through a set of equational rules, and is based on the approach presented in [47].

Definition 4.3.1. *(Structural Equivalence ≡) Structural equivalence relation ≡ :: PROC ↔ PROC is defined as the least relation which (i) extends α-equivalence, (ii) is an equivalence relation, (iii) is contextual, and (iv) satisfies the following equalities*

$$\begin{array}{llcll}
(\text{P-EXTR}) & \mathsf{new}\, n.P \parallel Q & \equiv & P \parallel \mathsf{new}\, n.Q & n \notin \mathsf{fn}(P) \\
(\text{P-COM}) & P \parallel Q & \equiv & Q \parallel P & \\
(\text{P-ASSOC}) & (P \parallel Q) \parallel R & \equiv & P \parallel (Q \parallel R) & \\
(\text{P-STOP}_1) & P \parallel \mathsf{stop} & \equiv & P & \\
(\text{P-STOP}_2) & \mathsf{new}\, n.\mathsf{stop} & \equiv & \mathsf{stop} & \\
(\text{P-FLIP}) & \mathsf{new}\, n.\mathsf{new}\, m.P & \equiv & \mathsf{new}\, m.\mathsf{new}\, n.P &
\end{array}$$

Condition (i) emphasises that structural equivalence is a more *refined* notion of equality than α-equivalence (*Def$^{\underline{n}}$* 4.2.3). Condition (ii) implies that $\equiv$ is reflexive, symmetric and transitive. Condition (iii) implies that reasoning about structural equivalence can be applied to process sub-terms. Rules (P-COM) and (P-ASSOC) define commutativity and associativity of parallel composition operator $\|$. Rules (P-STOP$_1$) and (P-STOP$_2$) serve as a form of garbage collection, removing terminal processes and unnecessary bound names. Rule (P-FLIP) negates the order of defined bound channels. Finally, rule (P-EXTR) is the most involving, by describing *scope extrusion* based on process structure. If name c bound in Q is not free in P, then knowledge of c can be extruded to the latter. Clearly, if c is free in P, exporting its name should be disallowed to avoid inadvertent name capture.

4.4 Bisimilarity

The notion of bisimilarity is a long studied form of *behavioural equivalence* within the context of the π-calculus. More specifically, we are now able to *compare* process behaviour based on their LTS representations. Informally, two processes are *bisimilar* if we cannot distinguish between their behaviour. The reasons for its appeal are numerous, and are discussed below (based on [75]).

1. Bisimilarity is accepted as the *finest equivalence relation* one could impose on processes. In other words, we are able to pair the *most* processes by comparing their behaviour. Moreover, proof of equality through coarser forms of equivalence relations (including structural equivalence) implies that the processes are also bisimilar.

2. The notion of bisimilarity admits desirable mathematical properties, the most important of which is its straightforward *coinductive proof technique* [50, 67, 76]. Proving behavioural equality of two processes simply involves proving the existence of a bisimulation relation (between the pair) adhering to certain properties — more below. Moreover, efficiency of its associated algorithms are of significant aid in automating (or semi-automating) behavioural proof techniques in certain cases [41].

3. Bisimilarity can be used to *abstract* over uninteresting detail of system behaviour. Clearly, what information is deemed irrelevant depends on the particular scenario. However, once said information is identified, it can be ignored in straightforward fashion. For instance, a particular scenario may dictate that we do not care about input actions, as long as exhibited output behaviour is identical. Admittedly, it is unclear at this point what we mean by the abstraction of uninteresting detail. However, we shall make extensive use of this property in chapter 5.

4. The *compositional* approach to the analysis of process behaviour derives its strength from the definition of bisimilarity. This is particularly true due to its check *locality*, and its *lack of hierarchy* (on checks) [75]. The former implies that one need only verify the immediate

transitions emerging from processes under consideration. On the other hand, the latter implies that no ordering on the checks is required, as long as we ascertain validity for each process pair. This is in sharp contrast with inductive proofs, which require a precise ordering on checks.

Notions of bisimilarity and co-induction are intimately intertwined [75]. However, note that this work should not be considered as an investigation of either concept — we are simply motivating their use. The interested reader is pointed to [50, 67, 76, 75, 47] for in-depth discussions of both, and applications for their use.

We next attempt to formalise the notion of bisimilarity. More precisely, two processes are deemed bisimilar if they match each other's actions. The term *bisimilarity* shall henceforth refer to to a particular form which is *weak* up to silent actions. In other words, *weak bisimilarity* ignores silent actions, as long as processes' external actions match. The rationale behind this approach is the fact that we should not be able to discern any difference in behaviour based on actions which cannot be observed. In general, weak bisimilarity has been accepted as a more *natural* form of behavioural equivalence (as opposed to its strong counterpart, which also pairs τ actions) [47], and shall be adopted throughout this dissertation.

Bisimilarity equivalence is based on the definition of the *bisimulation relation*, which is defined over LTS structures (*Def*$^{\underline{n}}$ 4.2.5). The process of abstracting over τ actions is provided by the *weak action*, defined below.

Definition 4.4.1. *(Weak action $\overset{\widehat{\alpha}}{\Rightarrow}$) We define the weak action for $\alpha \in Act$, written $P \overset{\widehat{\alpha}}{\Rightarrow} Q$ as follows*

- $P(\overset{\tau}{\rightarrow})^* Q$ *if* $\alpha = \tau$
- $P(\overset{\tau}{\rightarrow})^* P' \overset{\alpha}{\rightarrow} Q'(\overset{\tau}{\rightarrow})^* Q$ *if α is an external action.*

Hence, $P \overset{\widehat{\alpha}}{\Rightarrow} Q$ signifies that process P evolves to Q in a number of steps. Moreover, this sequence of transitions depends on α. If $\alpha = \tau$, then this transition sequence represents an arbitrarily long succession of silent actions. However, if α is an external action then the occurrence of α is preceded and succeeded by (arbitrarily long; possibly empty) sequences of silent actions. Using this notion of the weak action we can now define the *bisimulation relation*.

Definition 4.4.2. *(Bisimulations) Given LTSs* $(S_1, \textsc{Act}_1, \rightarrow_1)$, $(S_2, \textsc{Act}_2, \rightarrow_2)$*, we say a relation* $\mathcal{R} :: S_1 \leftrightarrow S_2$ *is a bisimulation, if whenever* $(P \mathcal{R} Q)$*,*

$$\bullet\, (P \overset{\alpha}{\rightarrow} P')\, implies\, ((Q \overset{\widehat{\alpha}}{\Rightarrow} Q')\, and\, (P' \mathcal{R} Q'))$$
$$\bullet\, (Q \overset{\alpha}{\rightarrow} Q')\, implies\, ((P \overset{\widehat{\alpha}}{\Rightarrow} P')\, and\, (P' \mathcal{R} Q'))$$

Hence, a relation between processes is a bisimulation if it adheres to the *transfer property i.e.,* if each action from one LTS can be paired by a weak action from the other (and vice versa). Based on the bisimulation relation we can now define the notion of *bisimilarity*.

Definition 4.4.3. *(Bisimilarity relation $\approx$) The bisimilarity relation, denoted by $\approx$, is defined to be the union of all bisimulation relations.*

Given that $\approx$ is the union of all bisimulations, $\approx$ is itself a bisimulation relation by the definition of set union. Moreover, it is also the *largest* bisimulation relation, since all other bisimulations are a subset of this relation. Hence, the definition of $\approx$ is a coinductive one, and admits a straightforward proof technique. To show that two systems S_1, S_2 are bisimilar, we simply need to exhibit a bisimulation containing the pair (S_1, S_2) since the bisimulation is guaranteed to be in $\approx$. Relation $\approx$ can also be shown to be contextual, as well as an equivalence relation [46].

Example 4. Consider systems

$$P_6 \triangleq c_1?\langle x\rangle.c_2!\langle\rangle.\mathsf{stop},\ \ P_7 \triangleq c_1?\langle x\rangle.\mathsf{if}\ x = 1\ \mathsf{then}\ (c_2!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (c_2!\langle\rangle.\mathsf{stop})$$

Process P_6 first communicates on c_1, then synchronises on c_2 and eventually terminates. On the other hand, P_7 also receives a value on c_1, but exhibits branching behaviour depending on the value received on c_1. However, on further inspection we notice that both branches exhibit the same behaviour, also synchronising on c_2 before terminating. In conclusion, although P_7 may take an additional internal step to decide which branch to compute, we expect both P_6 and P_7 to externally exhibit identical behaviour. This statement is proven by exhibiting a bisimulation between P_6, P_7, and is defined below.

$$\begin{aligned}\{\ \ (\ c_1?\langle x\rangle.c_2!\langle\rangle.\mathsf{stop}\,,\ \ c_1?\langle x\rangle.\mathsf{if}\ x = 1\ \mathsf{then}\ c_2!\langle\rangle.\mathsf{stop}\ \mathsf{else}\ c_2!\langle\rangle.\mathsf{stop}\ )\,,\\ (\ c_2!\langle\rangle.\mathsf{stop}\,,\ \ c_2!\langle\rangle.\mathsf{stop}\ )\ \}\end{aligned}$$

Hence proving our observation. Note that although we have shown P_6, P_7 to be *behaviourally equivalent*, the rules in *Def$^{\underline{n}}$* 4.3.1 do not define P_6, P_7 as *structurally equivalent i.e.,* $P_6 \not\equiv P_7$. We have therefore seen a witness example of the fact that behavioural equivalence is a more refined notion of equality than that based on structure; we can pair more processes behaviourally than structurally. ■

4.5 Conclusions

This chapter presented an overview of the π-calculus; we introduced its syntax, assigned an action semantics, defined an appropriate form of structural equivalence $\equiv$, and also motivated the use of bisimilarity relation $\approx$ as the most refined form of behavioural equivalence. Moreover, we saw how the π-calculus is the most adept formalism for describing concurrent computations of dynamic systems. This notion of dynamicity partly coincides with our understanding of dynamic architectures presented in section 3.4.4. Most importantly however, the above language serves as a theoretical foundation for the formalism presented in chapter 5.

Part II

The Theory

5. A Calculus of Distributed Monitored Processes

The next chapter proposes a *generalised* framework capturing *varied* distributed monitoring approaches (introduced in chapter 3). To this effect we present mDPI, a π-calculus adaptation with explicit notions of *monitoring* and *distribution*. The reader is firstly acquainted with an informal account of the setting we are out to achieve. We next show how mDPI describes required scenarios through an overview of the language. This involves (i) an interpretation of the language syntax, followed by (ii) an *extensible* LTS semantics allowing for the expression of mDPI term behaviour at different *levels of abstraction*. Later on we also consider different forms of equality, allowing for the comparison of terms on a *structural* and *behavioural* basis. The latter part of the chapter is focused on proving results for our framework, achieved through the use of introduced machinery. Throughout this chapter keep in mind that we restrict our interest to a *non-interfering* form of monitors *i.e.,* monitors which do not affect process computation. Moreover, our interpretation of the monitoring semantics are at a theoretical level, and do not encode practical considerations such as time and memory consumption (*i.e.,* we do not consider system overhead).

5.1 Overview

We are interested in formally studying the *runtime monitoring* of *distributed systems*. Our aim is to *precisely* reason about the scenario's capabilities, as well as its limitations. Chapter 2 saw the introduction of runtime verification, employing *runtime monitors* tasked with verifying system traces (representing effected behaviour) correct with respect to some requirements. We later explored the extension of this approach to *distributed settings* in chapter 3. This chapter highlighted the need for such extension, as well as discussing additional complexities introduced by distribution (which we are still bound by during subsequent analysis). Moreover, the chapter described a broad taxonomy of possible approaches to the monitoring of distributed systems, together with additional considerations regarding their suitability. The following chapter aims to formalise the concepts introduced by distributed monitoring.

To this end we shall present mDPi, a π-calculus adaptation with explicit *distribution* and *monitoring capabilities*. Serving as motivation for our approach, we start by considering the setting described by the standard calculus (chapter 4), and incrementally build up to our scenario of interest. Given the *location agnostic* approach taken by the π-calculus, this implies that distribution is at most an *implicit* concept. In other words, π-calculus terms can be considered as *concurrent* processes $P_1, P_2....P_i$ running independently (at unspecified locations), occasionally synchronising over channels $c_1, c_2...c_j$. An example scenario is shown below.

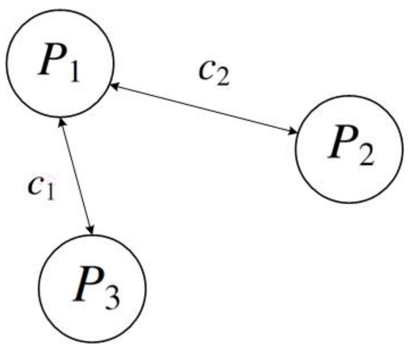


Figure 5.1: Example π-calculus processes.

Process P_1 eventually communicates with P_2 over channel c_2, as well as communicating with P_3 over c_1. At this stage it is unclear whether all three processes are executing locally to one another, whether all three are located remotely, or some arrangement in between. Given our interest in describing *distributed computation*, our next step is to *explicitly* introduce located executions. This is achieved by introducing the notion of *locations*, as highlighted in Fig. 5.2.

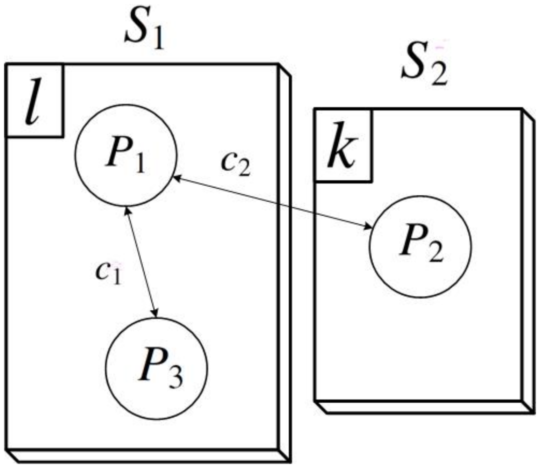


Figure 5.2: Example *located* π-calculus processes.

Distribution is made explicit by assigning P_1 and P_3 a common location k, whereas P_2 is located remotely by residing at l. Hence, the set of *localised processes* collectively make up the distributed system's computational effort. Also notice the introduction of *systems* S_1, S_2. Systems hence serve a *structural* purpose, organising various concurrent processes into computational entities *partitioned* per location. In general, a distributed system can be viewed as a collection of located systems. Communication can now be effected *locally* between two processes at the same location, or *remotely* between processes at differing locations. We argue that the distinction between local and remote communication is a crucial one — implementing the latter is more expensive than the former in a distributed setting, not to mention possible restrictions on remote interactions. The capability of distinguishing between communication forms (*i.e.*, local or remote) will eventually allow us to reason about the operation of both at the calculus level.

We next consider the extension of π with explicit notions of monitoring. For clarity, we first enhance *one* location k with monitoring capabilities, and later extend this setting to distributed systems. The addition of a monitoring semantics requires two further concepts; (i) *traces*, and (ii) *monitors*. A trace serves as a *log* of *past process behaviour*, subsequently verified by independently executing monitors for correctness against a set of requirements. *Order* of logged events is crucial for the monitoring of *temporal properties* (see chapter 2), and is hence something we have to cater for. The addition of said entities gives us the setting in Fig. 5.3.

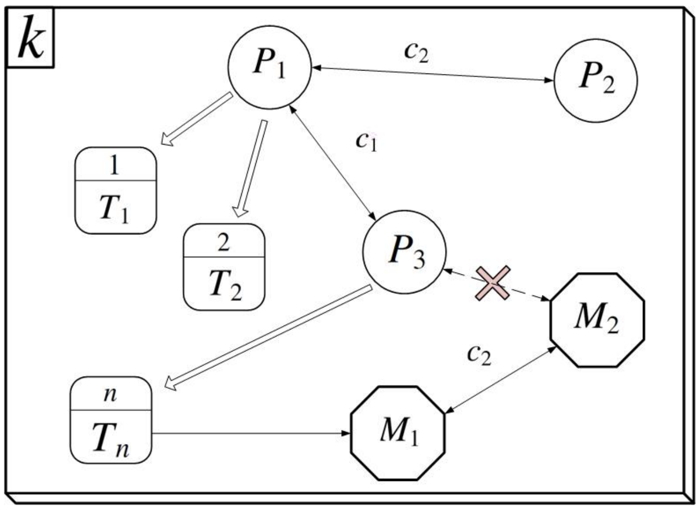


Figure 5.3: A Monitoring Scenario.

The setting in Fig. 5.3 describes three entity categories actively participating in computation; processes, monitors and traces. All three execute independently, but in certain cases can affect, or are the result of, each other's execution. Processes $P_1, P_2...P_i$ entail the system's primary computational aspect, whose behaviour we are interested in verifying. Moreover, said processes generate *trace entities* $T_1, T_2 ... T_n$ as a *side effect* during execution. One trace entity represents a *permanent* log of a single process event, and is generated *dynamically* (during process computation). By permanent, we understand that the log cannot be subsequently consumed or destroyed, only *analysed*. Trace entities are considered passive, incapable of independent computation — their purpose is to bridge the system's computation and monitoring effort. Moreover, each trace entity is assigned an additional *counter value* (shown in Fig. 5.3), representing its position in the local ordering of events. This counter encodes a temporal order on trace entities, with the resulting sequence taken to represent the system's local *trace*.

Monitors $M_1, M_2 ... M_j$ can be considered as a form of augmented processes. Crucially, their purpose is the analysis of traces as a *check* for adherence of process execution to requirements. This is made possible by a special monitor operator which *interfaces* with trace entities (discussed below), allowing for monitors to *analyse* logged trace information. We allow for the execution of multiple monitors for better *separation of concerns* — distinct monitors can be tasked with verifying different properties, increasing modularity. Given traces' non-consuming nature, this in effect allows for monitors to analyse the *same* trace for adherence to different properties. In general, the set of depicted monitors make up the system's global monitoring effort. As evidenced in Fig. 5.3, apart from analysing traces, monitors can *also* communicate over channels to *synchronise* their verification effort. Note that although chapter 2 introduces the possibility of monitors interacting with processes for the purpose of reacting to violations, we disallow this occurrence in our proposed setting. One could explore such avenues in future work, perhaps through a form of direct communication between monitor and process (currently disallowed). Hence, our framework is only interested in *identifying* incorrect process behaviour.

We have so far achieved an *event-based*, *asynchronous* monitoring framework of *monolithic architectures* (see chapters 2 and 3). Concurrently executing processes generate a trace, which is eventually analysed by monitors. Given that we are so far dealing with one location, this implies that the order on trace entities is total. The next step involves extending the above scenario to distributed settings (*i.e.*, admitting more than one location), thus formalising the *monitoring of distributed computations*. The result looks like Fig. 5.4.

Analogously to process distribution, the monitoring of distributed systems involves the partitioning of monitoring functionality across locations. Monitors can (i) analyse traces, and (ii) synchronise their efforts at *both* local and remote levels (relative to the monitors' location). Moreover, the distributed scenario admits a trace per location, with each system imposing its own local ordering on events. In general, the distributed setting admits a set of *remote unsynchronised clocks*, with one clock assigned to each location. We represent this clock through a *counter* which explicitly assigns a counter value to each trace entity, and is incremented on each assignment. We require our tracing semantics to produce *well-formed* traces *i.e.*, no two trace entities at common location k can have the same counter value. The result is a mechanism which allows for a temporal comparison of local trace entities, which however are *mutually*

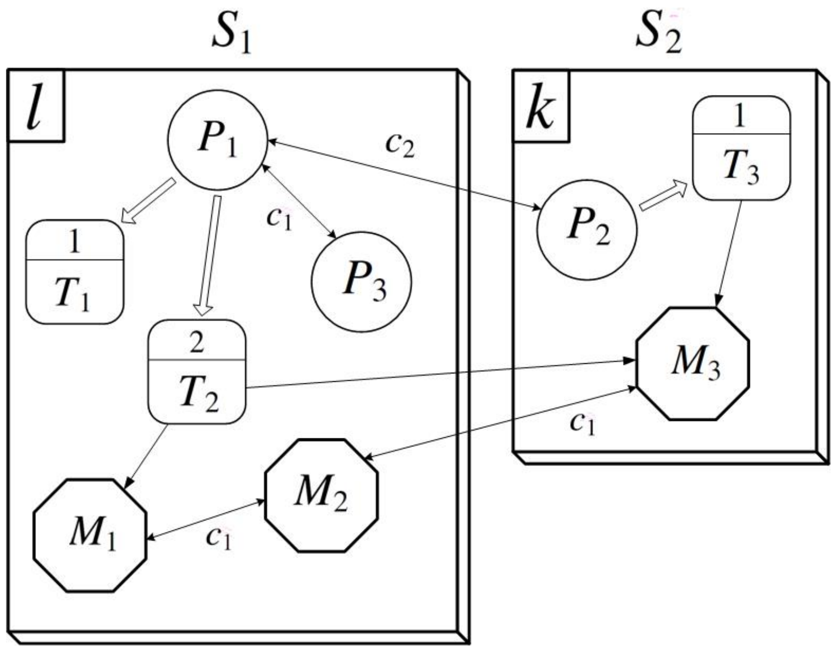


Figure 5.4: An Example Distributed Monitoring Scenario.

incomparable at a remote level.

Can we obtain a globally ordered trace, allowing for the comparison of remote traces? Although such functionality would have been a great asset, chapter 3 highlighted the difficulty (impossibility?) in achieving perfect synchrony amongst remote entities, implying that we cannot synchronise remote clocks. Hence, an implementable algorithm which extracts a global ordering is, in general, unattainable. Issues with synchrony of remote traces highlights our need to study monitoring in a distributed setting — we are faced with physical limitations on what what can be *reasonably* verified when faced with distribution. We also consider said limitations as a generalisation of our need to differentiate between local and remote interactions. mDPI respects said restraints, instead extracting a *partial* (section 3.5) ordering on remote events through the execution of monitors (more below).

Recall the broad taxonomy of monitoring approaches identified in chapter 3. We recognise that the *arrangement* of said monitors throughout the distributed configuration dictates the choice of monitoring approach. For instance, consider Fig. 5.5, depicting a distributed system monitored through an *orchestrated* approach. The monitoring effort is placed at a central location $\mathcal{G}$, with monitors remotely analysing localised traces. The depicted approach is considered *static* if the set of monitors $M_1 ... M_n$ remains unchanged during system execution. However, a *dynamic* orchestrated approach allows for the monitor configuration to change at runtime, allowing for properties to *evolve* / the addition of *new* properties (see chapter 3).

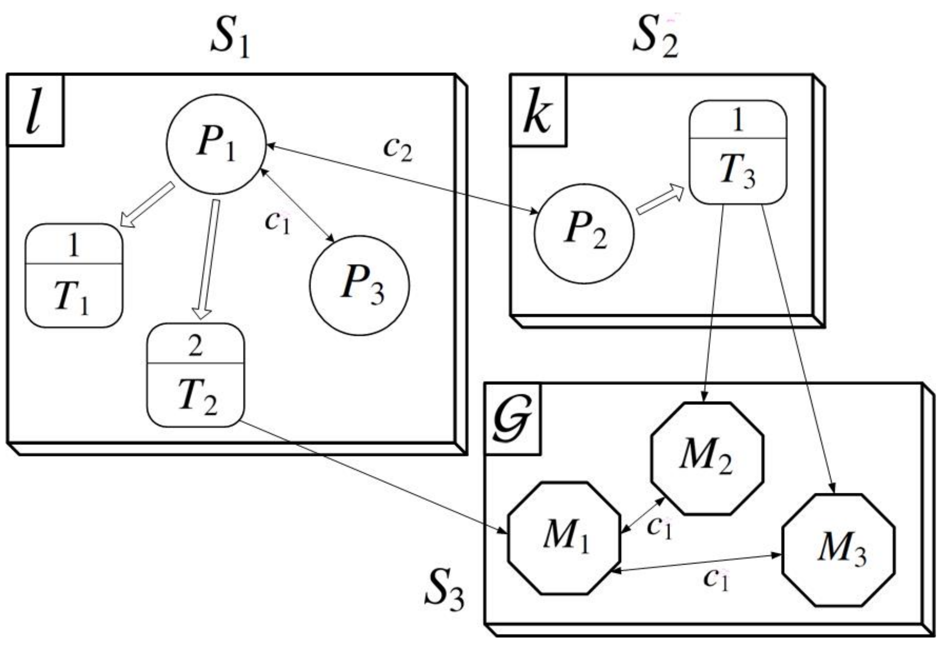


Figure 5.5: An Orchestrated Monitoring Scenario.

The scenario depicted in Fig. 5.6 differs from that in Fig. 5.2 (apart from admitting an extra location h) by *avoiding the analysis of remote traces*, exhibited in the latter scenario with M_3 remotely analysing T_2. Hence, Fig. 5.6 depicts a *static choreography* based approach. Monitors analyse trace entities *locally*, and communicate remotely to *synchronise* the global monitoring effort. However, this implies that processes at location h are *not* monitored in Fig. 5.6. Perhaps knowledge of said processes were not known to other locations at the start of computation (*i.e.*, the global system admits a dynamic configuration). In effect this points to the need for *dynamic choreography*, the final scenario presented in chapter 3. More specifically, we motivated the use of *migrating monitors* as a vehicle for studying dynamic choreography.

The monitors depicted in Fig. 5.7 analyse traces exclusively at a local level, before physically *migrating* to remote locations when the said systems' behaviour becomes pertinent to the distributed system's global correctness. Clearly, this approach does not stop monitors from communicating remotely for synchronisation. Note how the dynamic addition of locations (as is the case for location h, which is now verified) can now be handled, by directing monitors to relocate to new locations at runtime.

As a result, monitors in mDPi admit two additional capabilities (over processes); (i) the ability to *analyse* trace entities, and (ii) the ability to *migrate* between locations. Both capabilities shall be encoded in our prospective calculus. Moreover, the increased modularity of our monitor definitions, coupled with the separation of the monitoring and tracing semantics points to a *property agnostic* approach (chapter 3). In turn, this allows us to *spawn* monitors at runtime

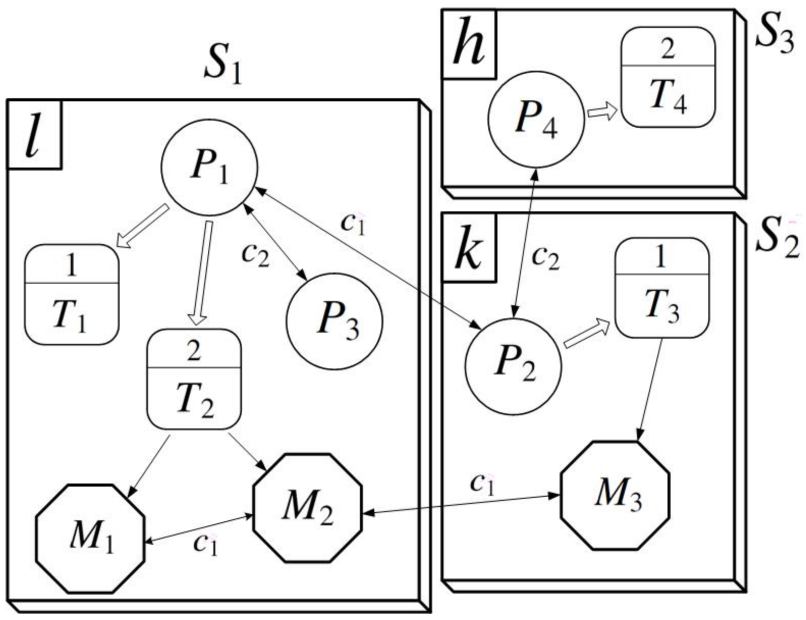


Figure 5.6: A Static Choreography-Based Scenario.

(possibly tasked with verifying new properties), which eventually migrate to different locations as required.

Both processes and monitors have been designed to communicate over a set of channels. However, given our interest in a non-interfering form of monitors (*i.e.,* no reaction to violations), it is key to consider whether monitors can interfere with processes communication (and vice versa). At this stage, we are faced with a design choice. One approach is to *statically* restrict both monitors and channels to use a *disjoint* set of channels. Although this option is potentially viable, it is unclear how the static enforcement of the partitioning of channels would work at runtime, especially when faced with name extrusion during computation (chapter 4). Moreover, this option places unnecessary resource constraints on the use of channel names; a channel can be sometimes used by monitors, and at others used by processes. As a result we might needlessly be restricting the application of valid monitoring approaches. We therefore take an alternate approach, allowing for processes and monitors to use the same channels. To avoid interference, we propose a partitioning of channel communication at the semantic level. In other words, we shall define the *capability* of processes to interact with other processes exclusively, with monitors only interacting with monitors. We argue that through this choice we alleviate the need for static enforcement at each computational step. Moreover, by imposing this requirement at a behavioural level we can produce a proof to ensure that (given the defined semantics) both interaction forms are always disjoint (more below).

We have so far motivated the distributed monitoring setting formalised by mDPI. The next step involves making use of this tool in order to *distill* and *identify* the core aspects of distributed

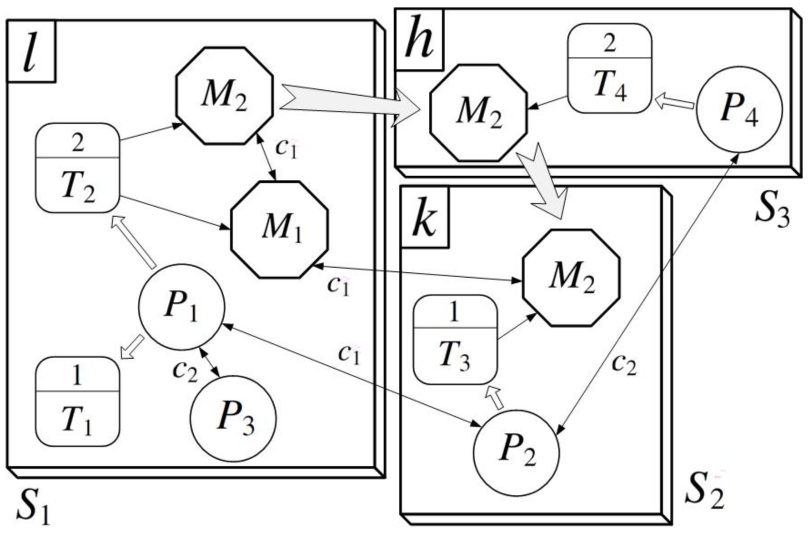


Figure 5.7: A Distributed System Verified By Migrating Monitors.

monitoring at a theoretical level. More specifically, we are primarily interested in proving three statements (in no particular order) identified as fundamental to the motivation of our approach

- *Does monitoring affect computation?* It is imperative that the monitoring approach we put forward does not affect the original process computation. If this were not the case, we would be employing monitors which could potentially alter the same processes' behaviour they are trying to verify. We would like to prove the impossibility of such scenarios, in order to ensure that we avoid said contradictory situations. At a theoretical level, monitors can effect process computation in two ways; (i) by admitting the capability to directly influence processes through some operator, and/or (ii) through interference over channel interactions. The impossibility of (i) should easily follow, since no monitor operator will be defined which directly interacts with processes. On the other hand, assurances that communication interference does not occur requires proof, especially since we have defined this requirement behaviourally. However, once such a result has been proven, we can rest assured of our non-interacting form of monitoring semantics, without the need for static checks. This requirement can be eventually relaxed when considering *controlled* interfering monitoring techniques, such as during runtime enforcement [34].

- *Is choreographed monitoring equivalent to orchestrated monitoring?* Chapter 3 introduced the possibility of monitoring distributed systems either at a local (choreographed) or global (orchestrated) level. The aforementioned chapter also explored advantages and disadvantages of each, with each approach admitting ideal scenarios of application. We conjecture that both approaches are *to some degree* equivalent. In other words, we expect to be able to verify the same property classes using both approaches. Conversely, using either approach should not impact monitors' expressivity *wrt.* the set of verifiable properties. Such considerations require a delicate understanding of the language semantics, since monitoring behaviour for both approaches *do* indeed differ at some level. More specifically, the *location* of orchestrated monitors will invariably differ from that of their choreographed counterparts. This points to the need for considering system behaviour at different *levels of abstraction*; in certain circumstances we shall consider behaviour *extensively* (*i.e.,* considering additional notions such as location information), whereas in others we shall *ignore* certain 'uninteresting' details. We shall return to the consideration of system behaviour at different abstraction layers at a later stage.

- *Do migrating monitors preserve locality?* The migrating monitor approach emerged during our discussion of dynamic choreography. This approach derives its strength from its adoption of monitor migration as a primitive construct. We further conjectured that this approach respects *locality* (chapter 3). The term 'locality' in our framework takes the form of *localized analysis of trace entities i.e.,* trace analysis only occurs at a local level. Clearly, localised trace analysis does not stop monitors from remotely synchronising. However, it forces monitors to avoid exposing confidential trace information at a remote level. This advantage, together with tolerance to dynamic architectures were put forward as migrating monitors' main advantages. We aim to provide assurances of locality for migrating monitors, in the form of a proof. In turn, this could go a long way to providing guarantees of the approach's trustworthiness.

We shall come back to the above questions at a later stage of our consideration of mDPi, after obtaining enough mathematics allowing for the formalisation of the required statements.

5.2 The mDPi Calculus

The next section is dedicated to describing the calculus in detail; at first by introducing the language operators and necessary concepts, followed by the definition of a language semantics. In addition, we introduce machinery which allows for the analysis of system behaviour at different *levels of abstraction*, in turn facilitating eventual consideration of the three questions posed of our framework at the end of section 5.1.

5.2.1 Syntax

We next present the mDPi language. The syntax assumes denumerable sets of channel names c, d, $b \in$ Chans, location names $l, k \in$ Locs, basic values $a, b \in$ BV and variables x, $y \in$ Vars; identifiers u, v range over Idents $=$ Chans $\cup$ Locs $\cup$ Vars and lists of identifiers $v_1, \ldots, v_n$ are denoted as $\bar{v}$.

Channels names act as communication links, referred to by processes and monitors alike. However, although both processes and monitors have access to the same channels, they do not immediately interact. Moreover, channels are location agnostic, and can be referred to by any located entity privy to knowledge of their existence. Location names represent some located *environment* where entities can execute. Each entity is hence assigned a location. Importantly, location information allows us to statically determine which entities are local and remote to some particular entity under consideration. The set of basic values BV represents some unspecified collection of strings, booleans, integers, doubles *etc.*. Finally, variables act as information placeholders *i.e.,* containing either a basic value or a channel name.

The syntax in mDPi mirrors the above scenario, adopting four syntactic categories; (i) *systems*, (ii) *processes* , (iii) *monitors* and (iv) *traces*. We give a brief informal account of the intended interpretation of each class, together with their corresponding syntax.

Systems

The set of systems $S, V \in \textsc{Sys}$ is inductively defined in Fig. 5.8 (with P, Q ranging over processes).

$$S, V \quad ::= \quad k[\![P]\!] \;\mid\; S \parallel V \;\mid\; \mathsf{new}\, c.S$$

Figure 5.8: System Syntax

- Located processes are represented through $k[\![P]\!]$, denoting that process P is located at k. Hence, each process is syntactically tagged with its location. Moreover, any computation emanating from P must be performed whilst in its located form.
- $S \parallel V$ represents two systems executing in parallel. Both systems might reside at the same location (hence marked with the same k), or they might be remote (*i.e.,* tagged with differing locations). Both S and V execute independently, and can communicate by exchanging information (more below).
- Knowledge of channel c extended to system S is written as $\mathsf{new}\, c.S$. The *scope* of c is said to be restricted to S. Hence, c can be used internally for communication within S, but is unavailable for use with other systems. Clearly, the scope of c may change as a result of interaction through scope extrusion.

Hence systems serve a structural purpose by assigning location to processes, arranging them in parallel and demarcating channel scope. Note that knowledge of locations *cannot* be scoped. Instead, the presence of *located processes* can be learnt at runtime through name extrusion, by exporting channel names linking to new processes. This process may happen to reside at (i) a *known* location, or (ii) at a *previously unseen* location. As a result, the dynamic addition of new locations is at most implicit in mDPi. At this stage the reader might also question the lack of monitor and trace entities in the above definition. This is borne out of the design choice to consider both as special forms of processes, for reasons of expression succinctness. We shall explicitly differentiate between processes, monitors and traces through appropriate syntax.

Processes

Processes $P, Q \in$ Proc represent the system's computational aspect, with Proc representing the (infinite) set of syntactically valid processes inductively defined in Fig. 5.9

$$\begin{aligned} P, Q \quad ::= \quad & \mathsf{stop} \mid u!\bar{v}.P \mid u?\bar{x}.P \mid \mathsf{new}\, c.P \mid \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P \| Q \mid *P \mid \\ & \{M\}^n \mid T \end{aligned}$$

Figure 5.9: Process Syntax

- stop represents the terminal process, which does nothing.
- Process $u!\bar{v}.P$ represents a process which transmits list $\bar{v}$ on channel u, and continues as P. The contents of $\bar{v}$ can be either identifiers or basic values. We shall use notation $u!\langle\rangle.P$ when an empty list is sent on u. In this case process output is used for synchronisation purposes.
- Input of $\bar{v}$ on channel u is represented by process $u?\bar{x}.P$. Assuming that $\bar{v}$ is well defined (*i.e.*, of the same arity) with respect to $\bar{x}$, each entry $v_i \in \bar{v}$ is *substituted* for each instance of $x_i \in \bar{x}$ in P, with the process' computation resuming through the resulting process after substitution of each variable. Notation $u?\langle\rangle.P$ is used when no data is expected on u.
- Process $\mathsf{new}\, c.P$ serves an analogous purpose to that exhibited at the system level, this time restricting knowledge of c to process P. This construct hence serves as a form of channel scoping at the process level. We shall eventually see how to *elevate* channel scope from the process to the system level, thus allowing for scope extrusion across remote processes.
- $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ serves as a test of identifiers and basic values. If the test for equality returns true the computation resumes through P, else we execute Q. This operation is also elevated as a test of tuple equality in the standard manner.
- Overloaded operator $\|$ represents concurrency at the process level. Much like parallel systems, $P \| Q$ describes processes P and Q executing independently, and possibly synchronise through some interaction. We shall later see how mDPi takes the approach of elevating processes (along with monitors and traces) to the system level prior to affecting computation.

- $*P$ represents recursive computation, with the composite process denoting an unbounded number of copies of P executing in parallel.
- Syntax $\{M\}^n$ represents a monitor M tagged with counter value n. We shall elicit the structure of M below. However, for now note that at the process level we can syntactically determine which entities are acting as monitors. This is provided by identifying which entities are encapsulated in $\{\,\}^n$ parenthesis. Counter value n is used by M during monitor execution to keep track of trace entities analysed at the *current* location under consideration, as well as to extract partial orderings at a *remote* level. Counter n is usually initialised to 1.
- Trace entity T is dynamically generated during process execution, logging process behaviour pertinent to the monitoring process. We shall expand on the structure of T in the next section. Note that trace entities are permanent, in that once generated they cannot be altered or consumed. Moreover, monitors are the only structures capable of interfacing with traces for analysis.

This concludes our exposition of processes. Processes encode located turing-complete computation, and is hence capable of representing any distributed implementation. Moreover, we believe the syntax to be straightforward, minimally extending the calculus seen in chapter 4. Monitors and traces can be immediately identified through appropriate syntax (with monitors encapsulated within parenthesis, as well as using an additional a counter).

Traces

Traces serve the purpose of recording *past process computation* of interest in a runtime verification setting. In other words, traces encapsulate the necessary information which allow us to determine whether processes are behaving as required. We shall take an *event-based approach*, recording *individual* process activities. Logs of said activities are then organized into *sequences*, thus encoding the temporal ordering of events in order of occurrence. Each event is represented through *trace entity* $T \in \textsc{Trc}$, whose syntax is described in Fig. 5.10. Informally, we consider the encoding of process events through *process output*. This choice simplifies our language by avoiding the necessity of an additional operator representing an event. Hence, channel output $e!\bar{v}$ (effected by a process) can be taken to encode the occurrence of event e, with event parameters $\bar{v}$. For minimality we shall therefore only consider the logging of process output.

$$T \quad ::= \quad \mathbf{t}(c, \bar{d}, n)$$

Figure 5.10: Trace Entity Syntax

One trace entity represents a record of the occurrence of some process output on c, while transmitting $\bar{d}$. Moreover, this event occurred *locally* at (logical) timestamp n. We encode a temporal ordering of trace entities *per location*, implying that the interpretation of each n is bound to the entity's location. In other words, since each trace entity is located *i.e.,* is of the form $k[\![\mathbf{t}(c, \bar{d}, n)]\!]$, this implies that $\mathbf{t}(c, \bar{d}, n)$ is the n^{th} trace entity generated at k. Moreover, we cannot infer the location of T *wrt.* other trace entities located remotely to k.

Trace entities serve as the individual building blocks for trace sequences. Hence, whereas a trace entity records a log of an individual event, a trace represents a system's *execution*; entailing the sequence of effected events.

Definition 5.2.1. *(Traces) A trace is defined as a set of trace entities of the form* $k[\![\mathbf{t}(c, \bar{v}, n)]\!]$.

A trace is a hence a set of located trace entities. Order on said entities is inferred from counter value n. However, n is only ordered per location, implying that this sequence is *partitioned* on a per location basis. Hence, a trace can be interpreted as a set of sequences, with each sequence representing a sub-trace at a distinct location. Given the notion of well-formed traces, we shall semantically define the generation of traces such that no two trace entities admit the same values for k and n. Moreover, we conjecture that our semantics should preserve well formed-ness of traces at each step. We shall expand on the generation of distributed traces at a later stage.

Monitors

We next explore monitors $M, N \in$ Mon whose syntax is described in Fig. 5.11. Monitors execute independently of processes in the global system configuration. Moreover, they *do not* contribute to the system's execution in the traditional sense. Rather, their purpose is to *analyse traces* generated by processes at runtime in order to discern whether process behaviour is correct with respect to requirements. Recall that monitors are encapsulated within $\{M\}^n$, with n representing a counter value used for two purposes; (i) to keep track of the trace analysed *sequentially* (thus avoiding the re-analysis of the same entities), and (ii) to extract a partial ordering on trace entities at a *remote* level. We shall use notation $k\{\![M]\!\}^n$ to denote a process containing just monitor M with *counter value n* at location k *i.e.,* as syntactic sugar for $k[\![\{M\}^n]\!]$.

$$
\begin{array}{rcl}
M, N & ::= & \mathsf{go}\ u.M \mid u!\bar{v}.M \mid u?\bar{x}.M \mid \mathsf{new}\, c.M \mid \mathsf{if}\ u = v\ \mathsf{then}\ M\ \mathsf{else}\ N \mid M \| N \mid *M \mid \\
 & & \mathsf{setC}(u).M \mid \mathsf{ok} \mid \mathsf{fail} \mid \mathsf{stop} \mid \mathbf{m}(c, \bar{x}, k).M
\end{array}
$$

Figure 5.11: Monitor Syntax

- ok and fail are both terminal monitors, which hence do nothing. The former is used to *flag* that a property has been satisfied, whereas the latter flags property failure. Hence, both terminal monitors are used to signify the *outcome* of the runtime monitoring process.
- Overloaded operators stop, $u!\bar{v}.M$, $u?\bar{x}.M$, $\mathsf{new}\, c.M$, $\mathsf{if}\ u = v\ \mathsf{then}\ M\ \mathsf{else}\ N$, $M \,\|\, N$, and $*M$ take an analogous interpretation to their process equivalent. These operators are fundamental in giving monitors sufficient expressive power, subsequently used to encode the monitoring of high-level properties. We can for instance encode branching of monitoring behaviour, and even specify channel scope amongst monitors (which can later be extruded). Parallel composition through $\|$ takes an augmented meaning at the level of monitors; it allows for (i) the separation of concerns, (ii) localisation of monitoring, and (iii) concurrent verification. $*M$ gives monitors the ability to verify properties over infinite process computations, and is hence crucial when translating certain specification languages (including regular expressions). Finally monitors can use channels to synchronise their monitoring effort. Take for instance system $k[\![P_1 \,\|\, \{M_1\}^1]\!] \,\|\, l[\![P_2 \,\|\, \{M_2\}^1]\!]$, whereby we want to verify that P_1 first generates an event e_1, which is followed by event e_2 at P_2. Without using monitor migration, how could we go about monitoring such a property? Assume monitors M_1 and M_2. What we require is M_1 to start listening for a trace entity denoting the occurrence of e_1. Once the required trace entity is analysed, M_1 synchronises with M_2, thus triggering M_2 to start checking for e_2. Once confirmed, M_2 proceeds to ok.
- Monitor operator $\mathbf{m}(c, \bar{x}, k).M$ is the only operator in mDPI which interfaces with trace entities. This operator describes a monitor's request to *analyse* trace entity $\mathbf{t}(c, \bar{d}, n)$ located at k, substituting $d_i \in \bar{d}$ for $x_i \in \bar{x}$ in M (assuming $\bar{x}$ is well defined *wrt.* $\bar{d}$). Intuitively, a monitor of the form $\{\mathbf{m}(c, \bar{x}, k).M\}^n$ requests the n^{th} trace entity at location k, as long as it logs process output on channel c. If this log (at k) records some output on a different channel, the trace entity is ignored.
- Operator $\mathsf{go}\ u.M$ instructs monitor M to migrate to the location specified by u, re-aligning its counter in the process. Identifier u can either refer to a specific location k, or some variable x which is later instantiated during monitor execution. The use of variables allows us to *dynamically redirect* monitor location on the fly. In general, a located monitor of the form $l\{\![\mathsf{go}\ k.M]\!\}^n$ evolves to $k\{\![M]\!\}^{n'}$, where n' is the value of the counter assigned to *last* trace entity generated at k, incremented by 1. Although the necessity of the counter update is admittedly unclear at this stage, we will return to its use at a later stage. For now, it suffices to keep in mind that monitor re-alignment serves to extract a partial ordering on remote traces, in this case by exploiting migration's natural sequential semantics.
- On the other hand, we also require a mechanism which extracts an ordering on remote trace entities *without* resorting to migration. This need is borne out of the three alternative methods (discounting migrating monitors) for monitoring distributed architectures (see chapter 3). This mechanism is provided by operator $\mathsf{setC}(u).M$, which re-aligns M's

counter to the *last* counter value assigned to some trace entity at u, plus 1. We shall expand on the extraction of remote partial orderings in the next section. Similarly to the use of identifiers for go, u is either a location, or a variable which is instantiated during execution.

Notation $\mathbf{m}(c, \langle\rangle, k).M$ shall be used when we do not care about logged information of data transferred on c. This concludes our exposition of the mDPι syntax, expressed through an informal interpretation of the language operators. However, although we have seen each *individual* operator, there is still much to be said about the *collective* operation of the monitoring and tracing semantics, which enable the runtime verification of (distributed) process behaviour. To this end, the next section presents an overview of the mDPι tracing semantics for generating distributed traces, followed by a discussion of monitor operation for the extraction of partial orderings on traces (both locally and remotely).

Note that although monitors are special processes, the syntax disallows the definition of some entity which is part monitor, part process. In other words, one cannot define some entity which sometimes behaves as a process, and at others behaves as a monitor — it either behaves completely as one or the other. This property shall be useful during our formalisation of the language semantics.

Trace Generation

We have so far described traces as a sequence of trace entities, ordered by counter values. Consider example system

$$S_1 = k[\![P \parallel \mathbf{t}(c_1, \bar{d}, 1) \parallel \mathbf{t}(c_1, \bar{e}, 3) \parallel \mathbf{t}(c_2, \bar{v}, 2)]\!]$$

we can infer that process P first output $\bar{d}$ on c_1, then $\bar{v}$ on c_2, and finally $\bar{e}$ on c_1. The order of trace entities at the level of the expression is unimportant; order is rather inferred from the assigned counter values (and location). Due to restrictions on remote synchronisation (chapter 3), we opt for a local ordering on events. Hence, given system

$$S_2 = k[\![P_1 \parallel \mathbf{t}(c_1, \bar{d}, 1) \parallel \mathbf{t}(c_2, \bar{e}, 2)]\!] \parallel l[\![P_2 \parallel \mathbf{t}(c_3, \bar{v}, 1) \parallel \mathbf{t}(c_2, \bar{z}, 2)]\!]$$

we infer that at location k, P_1 first output $\bar{d}$ on c_1 then $\bar{e}$ on c_2, whereas at location l process P_2 first output $\bar{v}$ on c_3 then output $\bar{z}$ on c_2. Although we have a total ordering on trace entities per location, we have partitioned said traces *across* locations. In other words, given any trace entity pair from two remote locations, their order is *mutually incomparable* — there is no way how we can statically discern which event occurred first. Referring to the above example, event 1 at k could have occurred before event 1 at l, after event 2 (at l) or in between events 1 and 2 (at l). Analogous reasoning could be effected for any pair of remote trace entities.

The problem of trace generation hence lies with the extraction of a temporal ordering per location. Consider next system $S_3 = k[\![c_1!\bar{v_1}.c_2!\bar{v_2}.\mathsf{stop}]\!]$. Clearly, S_3 will eventually evolve to $k[\![\mathsf{stop} \parallel \mathbf{t}(c_1, \bar{v_1}, 1) \parallel \mathbf{t}(c_2, \bar{v_2}, 2)]\!]$, where information regarding both process output operations have been logged. However, had S_3 been something of the form $k[\![c_1!\bar{v_1}.c_2!\bar{v_2}.\mathsf{stop} \parallel \mathbf{t}(d, \bar{v}, 1)]\!]$, the system instead evolves to $k[\![\mathsf{stop} \parallel \mathbf{t}(d, \bar{v}, 1) \parallel \mathbf{t}(c_1, \bar{v_1}, 2) \parallel \mathbf{t}(c_2, \bar{v_2}, 3)]\!]$. Note how the latter two trace entities generated by effecting two output operations now have different counter values, due to the occurrence of a prior trace entity (at that location). This trace could have been generated during some prior computation. Had this trace been located at a different location (such as in $k[\![c_1!\bar{v_1}.c_2!\bar{v_2}.\mathsf{stop}]\!] \parallel l[\![\mathbf{t}(d, \bar{v}, 1)]\!]$), its presence would not have affected the subsequent generation of trace entities at k.

In general, generating trace entities on the fly requires knowledge of the *last* assigned counter value *per location*. Moreover, this information is to be regularly updated on the assignment of a counter value to *each* trace entity generated. We identify two ways for extracting required counter information. The first involves *static analysis* of the system, identifying the trace entity at k with the largest counter value. Although this approach avoids introducing additional machinery in our calculus, we consider it to be inelegant and impractical. This is especially true since such analysis is potentially expensive (especially with larger system implementations), and would have to take place on the generation of *each* trace entity. We instead adopt an alternate approach, by introducing the *counter state* acting as a *monotonically increasing logical clock maintained per location*. This state is modeled through a function $\delta \in \Delta :: \text{Locs} \rightarrow \mathbb{N}$ mapping each location to a natural number. Informally, we exploit this mapping in order for δ to keep track of a counter value per location. In other words, δ encodes a logical clock per location, used for extrapolating localised temporal orderings on trace entities.

The counter value at location k is obtained through function application *i.e.*, $\delta(k)$. Updating the counter state is also encoded in straightforward fashion through *function overriding* [82], written $f \oplus g$. The result is an *update* of function f, whose mapping is updated according to the values of function g. Given counter state δ and location k, we define *inc* (over counter states) which increments the counter value assigned to k by one.

Definition 5.2.2. *(Counter state update) Given counter state δ, and location $k \in dom(\delta)$, we increment the counter value assigned to k through operator inc* $:: (\Delta \times \text{Locs}) \rightarrow \Delta$*, defined as follows*

$$inc(\delta, k) \triangleq \delta \oplus \{(k, (\delta(k) + 1))\}$$

We use *inc* to update the counter state on the generation of *each* trace entity. The *initial* counter state δ_i prior to system computation is assumed to map all locations to the value 1. Hence, the first trace entity generated at each location will be assigned the first position in the local trace. Counter state δ is updated accordingly through *inc* on the generation of each trace entity at k, by incrementing the counter value for that location. Therefore, δ contains the next counter value to be assigned at each location throughout system execution. Finally, we assume

δ to be total, thus ensuring that the next counter value assigned to each location is always known.

As we shall see in the next section, the counter state is also of use during monitor re-alignment. For instance, a monitor shall update its counter upon migration to the value held by the counter state for the monitor's new location. Analogously, operator $\mathsf{setC}(k)$ queries the counter state for the value of the latest counter value for k, updating its monitor's counter accordingly. Hence, monitors will have the capability to *directly* interface with the counter state.

It is clear that we require knowledge of the counter state in order to fully determine how systems compute. Moreover, different counter states can generate alternate event orderings, implying that the counter state can also *indirectly* affect the system's monitoring effort. With the importance of the counter state in mind, the runtime semantics needs to be defined in terms of *configurations* C, R which take the clock into account *i.e.,*

$$C, R \in \textsc{Conf} :: \Delta \times \textsc{Sys}$$

where notation $\delta \triangleright S$ specifies that system S has counter state δ. We shall later consider the effect of the counter state on system equality. Does the counter state contribute to the system's structure? Should behavioural equality consider behaviour of the counter state? We shall return to the issue of system equality in section 5.3.

Trace Analysis

The mechanism adopted by mDPi monitors during trace analysis is tightly bound with the use of adjacent counters. As previously discussed, said counters are used for two purposes; (i) to keep track during *sequential* trace analysis, and (ii) during *monitor re-alignment*. This latter use of counters is crucial in obtaining a partial ordering on remote trace entities. More specifically, monitor counters are analogous to Lamport Timestamps [58] *i.e.,* a logical clock adapted to instead extract the *monitored-before relation* (section 3.5). Consider Fig. 5.12, describing two event sequences at locations k and l together with their corresponding trace entities (*i.e.,* depicting an example trace).

Suppose we want to monitor that a process output on c_2 at location k, followed by an output on c_1 at l never occurs. An appropriate monitor implementing an orchestrated approach entails

$$M = \mathcal{G}\{\![\mathbf{m}(c_2, \langle\rangle, k).\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1$$

Note how this monitor analyses the sub-traces at k and l remotely from central location $\mathcal{G}$. Let us consider M's execution on the above trace. M starts off by analysing the trace entity at location k with counter value 1, as requested by operator $\mathbf{m}$ and the current counter value. In general, in order for monitor operator $\mathbf{m}(c, \bar{x}, k)$. to analyse some trace entity it has to match on (i) the *channel under consideration* (ensuring that the monitor is analysing information on the

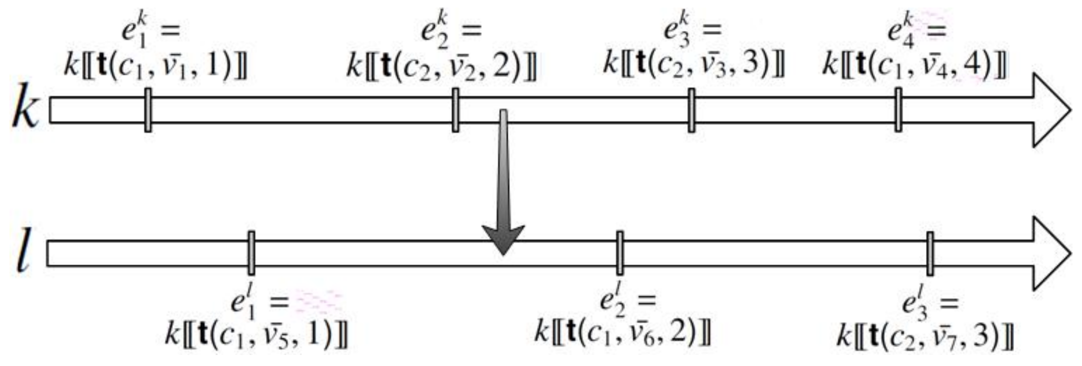

$e^k_1 = k[\![\mathbf{t}(c_1, \bar{v_1}, 1)]\!]$
$e^k_2 = k[\![\mathbf{t}(c_2, \bar{v_2}, 2)]\!]$
$e^k_3 = k[\![\mathbf{t}(c_2, \bar{v_3}, 3)]\!]$
$e^k_4 = k[\![\mathbf{t}(c_1, \bar{v_4}, 4)]\!]$
k
l
$e^l_1 = k[\![\mathbf{t}(c_1, \bar{v_5}, 1)]\!]$
$e^l_2 = k[\![\mathbf{t}(c_1, \bar{v_6}, 2)]\!]$
$e^l_3 = k[\![\mathbf{t}(c_2, \bar{v_7}, 3)]\!]$

Figure 5.12: An Example Trace.

event/channel it is interested in), (ii) the *counter value* (ensuring that the monitor is analysing the *next* trace entity in the temporal order defining the the trace), and (iii) the monitor's location of interest k. In this case, since M is interested in an output on channel c_2, and the first trace entity at k logs some process output on c_1, the monitor ignores the first entity by incrementing its counter. Therefore, M evolves to $\mathcal{G}\{\![\mathbf{m}(c_2, \langle\rangle, k).\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^2$.

The second trace entity in the sub-trace at k however records an output on c_2. This implies that all three conditions are satisfied for monitor trace input to occur; this second trace entity (i) exposes some output on c_2, the same channel which M is interested in, (ii) is next in line for analysis by M (*i.e.,* counter values match), and (iii) is located at k, the same location where M is currently interested in monitoring. Hence, M next computes to $\mathcal{G}\{\![\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^3$. The counter is also incremented upon trace input, forcing the monitor to avoid re-analysing the same trace entity. Note that $\mathbf{t}(c_2, \bar{v_2}, 2)$ is *not* consumed even after analysis by M — the trace is left untouched for verification by other monitors if so required.

We have thus seen the operation of sequential trace analysis when dealing with one location. However, the monitor has reached a stage in its computation where its' interest lies with a sub-trace at alternate location l. The monitor is hence instructed to *dynamically re-align* its counter value when switching its analysis of the trace at k to that at l, depicted in Fig. 5.12 by a vertical arrow. More specifically, if we wish to re-align a monitor to location l, then the act of *re-alignment* refers to an update of the monitor's counter to the *next* trace counter value at l. For instance, if we update the counter value to the next value assigned at l at the instant shown in Fig. 5.12, we infer that e^k_1, e^k_2 must have occurred prior to e^l_2, e^l_3. In general, counter re-alignment implies that any event which occurs at the new location post re-alignment must have occurred *after* events which the monitor analysed at the prior location, thus extracting a *temporal ordering* on remote trace entities. Clearly, this extracted ordering is however partial.

In general, monitor re-alignment represents our adopted mechanism for the extraction of a monitored-before relation (section 3.5). The act of re-alignment can be effected in one of two ways: (i) during *migration*, effectively exploiting the sequential aspect of migration, or (ii) *explicitly* through monitor operator $\mathsf{setC}(k).M$, whereby if a monitor wants to start analysing the trace at an alternate location it can request its counter to be re-aligned. In both cases, the next trace counter value at l is obtained through the counter state. Given that we are exploiting dynamic monitor execution to extract remote temporal orderings at runtime, we hence require our asynchronous monitors to, in a sense, *keep up* with the rate of trace generation in order to extract a *best-effort temporal ordering* between remote trace entities. By best-effort we mean that the *precision* of the extracted underlying temporal ordering across remote locations depends on the monitor's efficacy; the faster the monitor, the more remote trace entities we can recognise as temporally ordered. After re-alignment, the monitor starts operating as if it is local to the new location's trace (*i.e.,* sequentially analysing trace entities as shown above). In case of migration, the monitor is truly local to the new location's trace. On the other hand, when re-aligning through operator SetC the monitor is *physically remote*, but *logically re-aligned* to act as if a local monitor. Through this mechanism we achieve a concise solution for the monitoring of both local and remote traces in uniform fashion.

Given that M is an orchestrated monitor, it achieves a temporal ordering on remote traces through SetC. At that point in the system's computation, the only event to have occurred at l is e^l_1. M hence computes to $\mathcal{G}\{\![\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^2$, updating its counter to 2. Through re-alignment we hence infer that all trace entities henceforth generated at l with counter value ≥ 2 must have happened *after* events seen at k. Trace entity e^l_2 is eventually generated at l, which happens to record information of process output on c_1 (implying that the monitored property has been violated). This trace is input by the monitor, thus evaluating to terminal system $\mathcal{G}\{\![\mathsf{fail}]\!\}^3$ flagging that the property has been violated.

Another Example

We briefly consider the implementation for an alternate monitoring approach for the scenario depicted in Fig. 5.12. Consider systems $S_1 = k[\![P_1]\!]$, $S_2 = l[\![P_2]\!]$ as generic systems responsible for the generation of the above trace. We next consider an implementation of the same property through a static choreography-based approach. To this effect, we shall adopt two monitors $M_1 = k\{\![\mathbf{m}(c_2, \langle\rangle, k).c!\langle\rangle.\mathsf{stop}]\!\}^1$ and $M_2 = l\{\![c?\langle\rangle.\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1$; one each at each location k and l. Hence, the global system takes the form

$$k[\![P_1]\!] \parallel \mathsf{new}\, c.(k\{\![\mathbf{m}(c_2, \langle\rangle, k).c!\langle\rangle.\mathsf{stop}]\!\}^1 \parallel l\{\![c?\langle\rangle.\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1) \parallel l[\![P_2]\!]$$

Note the use of scoped name c for monitor synchronisation. Let us consider the above system's execution during trace generation and analysis. At some point, event e^k_1 occurs at process P_1, generating trace entity $k[\![\mathbf{t}(c_1, \bar{v}_1, 1)]\!]$. M_1 is the only monitor interested in analysing traces at k (statically identified through the use of operator m). However, as before, M_1 is uninterested in process output on c_1 and hence ignores the trace entity. The system therefore collectively evolves to

$$k[\![P_1']\!] \parallel k[\![\mathbf{t}(c_1, \bar{v_1}, 1)]\!] \parallel \mathsf{new}\, c.(k\{\![\mathbf{m}(c_2, \langle\rangle, k).c!\langle\rangle.\mathsf{stop}]\!\}^2 \parallel$$
$$l\{\![c?\langle\rangle.\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1) \parallel l[\![P_2]\!]$$

M_1 waits for the generation of the required trace entity, whereas M_2 waits for the go-ahead from M_1 before starting its computation. At some point event e^l_1 also occurs. However, although the generated trace is eventually of interest to M_2 (since it logs output on c_1), this monitor is still waiting for synchronisation on c, and hence does nothing. Event e^k_2 eventually occurs at P_1, generating trace entity $k[\![\mathbf{t}(c_2, \bar{v_2}, 2)]\!]$, which is of interest to M_1. The monitor hence analyses the trace, and upon confirmation synchronises with M_2 on c resulting in

$$k[\![P_1'']\!] \parallel k[\![\mathbf{t}(c_1, \bar{v_1}, 1)]\!] \parallel k[\![\mathbf{t}(c_2, \bar{v_2}, 2)]\!] \parallel k\{\![\mathsf{stop}]\!\}^3 \parallel$$
$$l\{\![\mathsf{setC}(l).\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1 \parallel l[\![P_2']\!] \parallel l[\![\mathbf{t}(c_1, \bar{v_5}, 1)]\!]$$

M_1 hence computes to a terminal process. It has completed its verification at k, and has signalled to M_2 to start verifying at l. M_2 immediately forces an update to its counter value through operator SetC, thus extracting a temporal ordering on events through synchronisation. Any trace entity at l analysed by M_2 after the counter update is guaranteed to temporally succeed all trace entities encountered by M_1 prior to synchronising. Since e^l_1 has already occurred at l, M_2's counter takes the value of 2 by order of SetC. Upon completion, M_2 waits for a trace entity logging some output on c_1 at l. Event e^l_2 eventually generates $l[\![\mathbf{t}(c_1, \bar{v_6}, 2)]\!]$, breaking the monitored property. This property violation is identified by M_2, which analyses the latter trace and signals failure by evaluating to fail *i.e.,*

$$k[\![P_1'']\!] \parallel k[\![\mathbf{t}(c_1, \bar{v_1}, 1)]\!] \parallel k[\![\mathbf{t}(c_2, \bar{v_2}, 2)]\!] \parallel k\{\![\mathsf{stop}]\!\}^3 \parallel$$
$$l\{\![\mathsf{fail}]\!\}^3 \parallel l[\![P_2']\!] \parallel l[\![\mathbf{t}(c_1, \bar{v_5}, 1)]\!] \parallel l[\![\mathbf{t}(c_1, \bar{v_6}, 2)]\!]$$

Systems S_1 and S_2 proceed in their execution unaware of property failure, with events e^k_3, e^k_4 and e^l_3 occurring after the violation. In other words, our monitors are limited to flagging errors. The above example concludes our informal tour of the mDPI semantics. A more formal treatment shall be provided in section 5.2.2.

We have seen two approaches so far; implementing orchestrated and statically choreographed monitors. Whereas the former approach used central monitors which analysed traces remotely, the latter adopted monitors which analysed locally, and synchronised remotely. Note how this latter approach avoided the exposure of trace information across locations by monitoring locally. This was not the case for orchestrated monitors, which exposed traces by remotely analysing their contents. We can also define a migrating monitor implementation for the same property, taking the form

$$k\{\![\mathbf{m}(c_2, \langle\rangle, k).\mathsf{go}\ l.\mathbf{m}(c_1, \langle\rangle, l).\mathsf{fail}]\!\}^1$$

The monitor starts its computation at k, before migrating to l to complete verification of the property. Counter re-alignment is effected during migration, exploiting the operator's sequential nature. Note the similarities in the definition of the orchestrated and migrating monitors. This is achieved through *elevated encapsulation* provided by migrating monitors; defining necessary monitoring functionality through one monitor while achieving a choreographed approach. This is in contrast with the statically choreographed approach, which required two monitors. The advantage with migrating monitors is highlighted further when dealing with dynamic architectures. However, the cost of migrating monitors is the adoption of an expensive go operator as a language primitive.

In general, we consider monitors as an *executable form* of system properties written in some *high-level specification language*. Hence, verifying properties written in some language simply requires us to define a *translation* from the language to an mDPɪ monitor. One can go further, by defining *multiple translations* of the same language; one for each distributed monitoring approach. The above examples show that, at least for certain properties, mDPɪ monitors are sufficiently expressive to monitor alternate distributed monitoring approaches (chapter 3). Clearly, the expressivity afforded to our monitors dictate what specification languages we can encode, and which approach fits best. As a result, we do not focus our attention to one specification language, but rather focus on the monitoring semantics. If we prove that mDPɪ monitors admit desirable characteristics, this would imply that any property verified through our monitors would also adhere to said characteristics.

Further Considerations

We extend standard notions of *variable binders*, *substitution*, and *α-equivalence* (presented in chapter 4) for mDPɪ. Note that monitor operator $\mathbf{m}(c, \tilde{x}, k).M$ acts as an *additional* variable binder (on top of $c?\tilde{x}.P$ and $c?\tilde{x}.M$), binding $x \in \tilde{x}$ in M. The use of binders implies the notion of variable substitution. See appendix B for the full definition of $S\,\sigma$ *i.e.*, substitution in its most general form. Crucially, since trace entities are closed terms ($\mathsf{fv}(T) = \emptyset$) this implies that substitution does not affect trace information — one would not want to alter permanent logged event information. The definition of $S\,\sigma$ assumes renaming of bound variables in case of variable capture. Variable renaming through substitution presents the preliminary form of α-equivalence, elevated to configurations in case of mDPɪ; two configurations are deemed α equivalent if they are the same, except in their naming of bound variables [47]. Since locations and counter values admit concrete values (*i.e.*, are not contained within variables), configurations have to match on both location and counter values in order to be α equivalent. This preliminary notion of equality makes sense, since the naming of variable placeholders should not effect system behaviour. Section 5.3 shall provide more refined notions of equality, pairing configurations on more involving concepts such as equality based on *structure*, and finally a notion of equality based on exhibited *behaviour*. We shall also apply the *barendregt convention* [47] to mDPɪ terms. In other words, we consider mDPɪ terms up to α-equivalence, by ensuring that all bound identifiers are distinct, and are chosen to be different from present free identifiers within the *context* the term is being used.

The notion of contexts is also crucial in mDPi — we shall often be interested in considering mDPi terms as part of some larger environment consisting of other systems. Reasoning about configurations-in-context give us the most profound understanding of their operation in its most general form, by considering how a system may also interact with its external surroundings (section 4.2.1). A context in mDPi is considered a 'configuration with a hole' of the form $\delta \triangleright S \parallel \_$ or $\delta \triangleright \mathsf{new}\, c.\_$, where $\_$ can be substituted with any valid system. We next elevate the property of *contextuality* over configurations (as opposed to processes in case of π; *Def$^{\underline{n}}$*4.2.4).

Definition 5.2.3. *(contextual relation) A relation $\mathcal{R}$ :: $\textsc{Conf} \leftrightarrow \textsc{Conf}$ is said to be contextual if it is preserved by operators $\parallel$ and* new*, hence adhering to the following two properties:*

- $(\delta \triangleright S_1 \;\mathcal{R}\; \delta \triangleright S_2) \Rightarrow ((\delta \triangleright (S_1 \parallel V) \;\mathcal{R}\; \delta \triangleright (S_2 \parallel V)) \wedge (\delta \triangleright (V \parallel S_1) \;\mathcal{R}\; \delta \triangleright (V \parallel S_2)))$
- $(\delta \triangleright S_1 \;\mathcal{R}\; \delta \triangleright S_2) \Rightarrow (\delta \triangleright \mathsf{new}\, n.S_1 \;\mathcal{R}\; \delta \triangleright \mathsf{new}\, n.S_2)$

Contextuality is either included as part of a relation's definition, or subsequently proven as a relation property.

This concludes our informal introduction of the mDPi language. However, before moving on we consider the issue of dynamic architectures in mDPi. The specification of dynamicity is derived from our interest in studying certain monitoring approaches when faced with dynamic architectures. You may recall chapter 4 introducing two possible forms of dynamicity in the π-calculus; dynamicity of channel links, and process mobility [47]. The former comes part and parcel with the π-calculus (and its extensions) due to inherent name extrusion. mDPi is no different, allowing for channel names to be extruded and used at runtime. The latter form of dynamicity refers to (i) the dynamic creation of new locations, and/or (ii) *process* migration. However, given that location information is not scoped in mDPi, this implies that the addition of new locations on the fly is at most implicit. Moreover, we do not allow for processes to migrate between locations, implying that a process is tied down to its location for the duration of its computation. Note that *monitor* migration should not be taken as a form of dynamic architecture, since monitors do not actively participate in system computation (in the traditional sense) — their purpose is rather to *verify* system behaviour. In conclusion, process mobility is not explicitly studied in mDPi. Nevertheless, it would be interesting to explore the effectiveness of monitoring approaches when faced with said mobility, and is left as future work.

5.2.2 A Semantics For mDPi

Our next task is to assign an appropriate formal semantics to the mDPi language. However, although we have an informal understanding of the language operators, there still are unresolved challenges. As section 5.1 at the beginning of the chapter hinted, we are often required to consider system behaviour at different *levels of abstraction*. More specifically, this section proposed the need to *expose different aspects* of system behaviour, depending on the statement we are required to prove. Consider systems S_1 and S_2 below.

$$S_1 = k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c, \bar{x}, k).\mathsf{ok}]\!\}^1, \quad S_2 = k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel l\{\![\mathbf{m}(c, \bar{x}, k).\mathsf{ok}]\!\}^1$$

Both systems admit a simple process outputting values $\bar{d}$ on c at k. However, S_1 and S_2 are not perfectly identical, due to differences in their monitoring effort. Whereas the former performs runtime monitoring at a local level, the latter monitors remotely due to being situated at different location l. Identifying such differences are hence crucial in certain scenarios — the former might represent a choreographed approach whereas the latter represents an orchestrated monitoring effort. Therefore, we require our language semantics to be capable of identifying differences in computation location. However, in other scenarios location may be deemed to be not pertinent. For example, we may *not* be interested in monitors' location, as long as both systems are monitored for the same properties. In such cases, we want our semantics to ignore/abstract over information of entities' runtime location. Clearly, we have so far only shown one example exposing the need for abstraction on system behaviour (over location information). In fact, our need is more *varied*, as evidenced by a further three possible scenarios below.

Scenario 5.1. *Do S_1 and S_2 admit the same behaviour, factoring in location?*

Scenario 5.2. *Do S_1 and S_2 admit the same behaviour, disregarding location?*

Scenario 5.3. *Do S_1 and S_2 admit the same process behaviour, disregarding monitor behaviour?*

Scenario 5.1 requires that we match systems based on both their processing and monitoring efforts, as well as considering the *located* aspect of computation. Hence, the first scenario requires the *finest* possible analysis of system behaviour, matching on additional monitoring and located aspects of computation (apart from the standard matching of process computation). As discussed above, we expect to be able to distinguish between S_1 and S_2 at a formal level.

On the other hand, Scenario 5.2 requires a *coarser* analysis of system behaviour. In this case, we are not interested in *where* computation takes place, as long as both processes and monitors otherwise exhibit the same behaviour. In case of S_1 and S_2 both would be considered identical, since the difference in location for monitoring computation would be ignored. Similar analysis shall be useful when considering whether different monitoring approaches are equally expressive; as long as our monitoring approaches can verify the same property classes, we do not care where the monitoring is effected.

The third scenario 5.3 exposes another form of information abstraction, this time ignoring a particular *aspect* of system execution. In this case we are uninterested in monitor behaviour, as long as process computation matches. Such analysis would once more render S_1 and S_2 as equivalent. We shall perform similar analysis when proving that the monitoring semantics do not affect process computation. This is achieved by proving that any system and its unmonitored counterpart produce the same computation, disregarding monitor behaviour.

More scenarios can be thought of. We may for instance consider *localised monitors* only, or *process computation* disregarding *location*. Even more scenarios are introduced when considering the effect of traces on system behaviour. Throughout this dissertation we shall however limit ourselves to focusing on scenarios which allow us to formalise the required statements outlined in section 5.1.

Keeping the above challenges in mind, we next explore how to, at a mathematical level, encode system behaviour. To this effect, we opt for an *action semantics* due to its well-documented advantages [47, 67, 76];

- To obtain standard *coinductive equational reasoning* through bisimilarity relation $\approx$ (*Def*$^{\underline{n}}$ 4.4.3),
- while also allowing for the *compositional* analysis of system behaviour.

More specifically, we expose system behaviour by describing a system's capability to *perform* actions, either *internally* or *externally* with its context. Although *internal actions* are undetectable to a system's environment, *external actions* are used to describe possible interactions with outside entities. In effect, this gives us a more general view of system behaviour, describing computation when in isolation, as well as within some larger computational endeavour. This view of processes is given in terms of a *Labelled Transition System* (see *Def*$^{\underline{n}}$ 4.2.5), where transition labels represent the system's action capabilities at each stage in its computation.

The next question lies as to how we shall go about extracting different LTS representations at various levels of abstraction (exemplified above). In general, each scenario requires *different* LTS semantics, which however admit only *slight* modifications in terms what actions we allow. These modifications dictate the *degree* to which we choose to abstract information (for that scenario). For instance, whereas scenario 5.1 would require its LTSs to record location information within their labels, scenario 5.2 would opt for LTSs which ignore location details. A naïve approach would require the definition of a transition system for *each* alternate view of system operation. Hence, analysing system behaviour under a new scenario requires us to define an alternate transition system exposing pertinent information in each case. Although correct, we view this approach as inefficient and inelegant, which is why opt for an alternate mechanism.

Instead, we exploit the underlying similarities inherent to all LTSs, giving rise to a *preliminary* LTS representation of system behaviour, termed a *pre-LTS*. Given that mDPI introduces notions of *monitoring* and *distribution*, this preliminary representation extends standard LTS views by decorating transition labels with

(i) *location information,* specifying locations that participated in the action;

(ii) *action modality,* specifying whether it is a process, monitor or trace action.

A pre-LTS representation is hence enriched with *extensive knowledge* of system behaviour. This implies that obtaining a *particular* view of system behaviour (as is required in scenarios 5.1, 5.2 and 5.3) simply involves *abstracting* unnecessary detail from the pre-LTS labels. Information abstraction is formally achieved through *filter function* Ω over labels, encapsulating logic of *how* abstraction is performed. Informally, we consider a pre-LTS to contain *too* much information of how systems behave for any *particular* scenario. Hence, we define a way how to *abstract non-pertinent information* (encapsulated in a filter function definition), in order to obtain an LTS containing *just enough* information to suit our needs. Simply put,

$$\textit{pre-LTS} \ + \ \Omega \ = \ \textit{LTS} \tag{5.1}$$

Given that Ω operates exclusively on transition labels, the result after filtering is another transition system. Notice the simplicity in achieving an *LTS* representation; given a system's pre-LTS (extracted through a standard rule set), in order to obtain an LTS we simply need to define *how* Ω filters information. In other words, obtaining different views of system behaviour involves the reuse of *existing derivation mechanics*, instantiated for each valid definition of Ω. By extension, obtaining a *particular* view simply requires the motivation of an appropriate Ω. Our setting hence becomes

$$\begin{tikzcd}
 & \boxed{\text{pre-LTS}} \arrow[dl, "\Omega_1"'] \arrow[d, "\Omega_2"] \arrow[dr, "\Omega_n"] & & \\
\boxed{\text{LTS}_1} & \boxed{\text{LTS}_2} & \cdots & \boxed{\text{LTS}_n}
\end{tikzcd}$$

Such that we can obtain various LTS representations from a pre-LTS.

However, although the above approach is convenient *wrt.* summarising LTS semantics definitions, it admits undesirable repercussions. More specifically, by encoding extended information on transition labels, a pre-LTS exhibits a rather *intensional* view of system behaviour *i.e.*, it exposes information of *both* the system's internal and external operation. The addition of tags to τ actions effectively results the *partitioning* of τ. In other words, added tags force us to *distinguish* between different forms of internal actions. For instance, although actions τ_p and τ_m represent internal actions, we can now deduce that the former was performed by a process, whereas the latter was performed by a monitor. Moreover, although this example considers action modality, τ is partitioned further when also considering location tags. The loss of the silent action is undesirable; we cannot, at a pre-LTS level, consider behavioural equality which is weak up to silent actions. In truth, comparing behaviour of pre-LTSs results in something akin to strong bisimilarity [76], *also* requiring the matching of τ tags. However, weak bisimilarity (section 4.4) is seen as a more natural form of bisimilarity as opposed to its strong counterpart [47]. It is for this reason that we shall endeavour to re-obtain an *extensional* view at the level of the LTS, abstracting over internal actions by reobtaining the silent action.

The reader might at this stage question why we tag τ actions in the first place. Wouldn't it be simpler to generate tag-less τs in our transition rules? Although the above issue would be resolved, we argue that, at a preliminary level, we *do* require the capability to distinguish between internal actions. This necessity is borne out of subsequent filtration based on label tags. In other words, there may be scenarios which are not interested in certain types of τs. The added tags hence allow us to decide which τs to keep, and which shall be ignored. Consider an example scenario where we are not interested in monitor behaviour. Certainly this implies that all monitor behaviour, both external and internal, is to be ignored from the pre-LTS. Had we not tagged τs prior to filtration, we would have had no way to distinguish between say, a process or a monitor internal action.

In conclusion, apart from benefits *wrt.* the use of an action semantics, we believe our approach to be additionally advantageous on two fronts;

1. It avoids repetition, by presenting a modular way for extracting different LTSs. Each LTS focuses on different aspects of system computation *i.e.*, representing behaviour at various levels of abstraction.
2. We identify a *hierarchy* on LTSs, depending on the filter functions used for their derivation. This hierarchy shall be exploited to prove statements at a particular level, which are then automatically valid for *coarser* abstraction levels.

The second advantage is admittedly vague at present; we shall elucidate this point throughout the forthcoming section. For now it suffices to say that we shall exploit the relationship between a pre-LTS and its various LTS views in order to obtain a hierarchy *wrt.* the level of encoded information. We subsequently make use of this hierarchy at the proof level, thus avoiding the need to re-prove results for each LTS. Simply put, if we prove some result when considering more information, the result should still hold when dealing with less. The next section presents a general transition rule system for the extraction of a pre-LTS, and is followed by an in-depth treatment of filter functions. We subsequently define the extraction of an LTS through the use of filter functions on a pre-LTS. Section 5.3 formalises configuration equivalence at a structural and behavioural level. Finally, section 5.4 presents our results *wrt.* the statements we set out to reason about (presented in section 5.1). Section 5.5 concludes the chapter.

The Pre-LTS

We next present the transition rules of our pre-LTS, defined over *closed* configurations *i.e.*, configurations of the form $\delta \triangleright S$ where $\mathsf{fv}(S) = \emptyset$. The transition labels denote *judgements* on a system's action capabilities. We employ *five* distinct actions:

- $C \xrightarrow{c?\bar{d}} R$; the ability of configuration C to receive tuple $\bar{d}$ on c, evolving to residual configuration R in the process.

- $C \xrightarrow{(\bar{b})c!\bar{d}} R$; the ability of configuration C to transmit tuple $\bar{d}$ on channel c, evolving to R after effecting the action. Moreover, this judgement encodes the ability of C to *export* knowledge of bound names $\bar{b}$ *during* channel output, such that $\bar{b} \subseteq \bar{d}$ and $c \notin \bar{b}$. In other words, we restrict the exporting of names to those which are part of the transferred tuple in order to avoid unintended capture of free names.

- $C \xrightarrow{(\bar{b})\mathrm{t}(c,\bar{d},n)} R$; the ability of C to *expose* the n^{th} trace entity information of some logged process output of $\bar{d}$ on c. The addition of the counter value to judgements shall be used to *synchronise* trace analysis based on the extracted ordering of events. Similarly to the output action, trace exposure can simultaneously export names, also bound by $\bar{d}$ as before.

- $C \xrightarrow{\mathrm{m}(c,\bar{d},k,n)} R$; denoting the ability of C to *read* information logged by the n^{th} trace entity at location k recording process output of $\bar{d}$ on c, as well as evolving to R in the process.

- $C \xrightarrow{\tau} R$; denoting an *internal* action executed by C, evolving to R in the process. Both forms of communication (*i.e.,* process and monitor comm.), trace analysis, as well as other computational steps (such as migration) shall be represented by an unobservable τ action.

An mDPi term can, at any stage of its computation, perform one of the above five actions. As motivated above, for generality we extend all actions with additional information, including (i) location information, denoting which location(s) participated in the action, and (ii) modality $\mu \in (\textsc{Mod} = \{p, m, t\})$, recording whether it was a *process*, *monitor* or *trace* action.

For instance, action $\tau_{\langle p:l,k\rangle}$ denotes a *process* internal action involving locations l and k whereas $\tau_{\langle m:l,l\rangle}$ denotes a *monitor* internal action, local to (both produced and consumed at) location l; the explanations for actions $(\bar{b})c!\bar{d}_{\langle\mu:l\rangle}$ (*resp.* $c?\bar{d}_{\langle\mu:l\rangle}$) are similar; here the location tag l denotes the source (*resp.* destination location) of the communication action. We restrict modality of actions $\mathrm{t}(c, \bar{d}, n)_{\langle t:k\rangle}$ and $\mathrm{m}(c, \bar{d}, l, n)_{\langle m:k\rangle}$ to t and m respectively, since only traces and monitors are capable of effecting the aforementioned actions.

Notation 5.1. *We shall take $\alpha_{\mu,l} \in Act_{\mu,l}$ to represent the set of actions tagged with modality and location information. Through this notation we elide other sets; for instance $\alpha_l \in Act_l$ represents the set of actions tagged with information location (only). Overloaded notation $\mathsf{n}(\alpha)$, $\mathsf{fn}(\alpha)$ and*

$\mathsf{bn}(\alpha)$ *is taken to represent the set of names, free names and bound names in action* α*, such that* $\mathsf{n}(\alpha) = \mathsf{fn}(\alpha) \cup \mathsf{bn}(\alpha)$*. Taking action* $\alpha = (\bar{b})c!\bar{d}_{\langle p:k\rangle}$ *as an example;*

$$\begin{array}{rcl} \mathsf{fn}((\bar{b})c!\bar{d}_{\langle p:k\rangle}) & = & \{c, \bar{d}\}/\bar{b} \\ \mathsf{bn}((\bar{b})c!\bar{d}_{\langle p:k\rangle}) & = & \bar{b} \end{array}$$

Note that tagged information is neither free nor bound. We now move on to an exposition of the rules. As we shall see mDPi admits an extensive list of transition rules, which is why we present the rules in smaller digestible chunks. See appendix A for a complete listing of the calculus' syntax and semantics.

$$\textsc{Out}_p\ \frac{}{\delta \rhd k[\![c!\bar{d}.P]\!] \xrightarrow{c!\bar{d}_{\langle p:k\rangle}} \mathsf{inc}(\delta, k) \rhd k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{d}, \delta(k))]\!]}$$

$$\textsc{In}_p\ \frac{}{\delta \rhd k[\![c?\bar{x}.P]\!] \xrightarrow{c?\bar{d}_{\langle p:k\rangle}} \delta \rhd k[\![P\{^{\bar{d}}/_{\bar{x}}\}]\!]} \qquad \textsc{Out}_m\ \frac{}{\delta \rhd k\{\![c!\bar{d}.M]\!\}^n \xrightarrow{c!\bar{d}_{\langle m:k\rangle}} \delta \rhd k\{\![M]\!\}^n}$$

$$\textsc{In}_m\ \frac{}{\delta \rhd k\{\![c?\bar{x}.M]\!\}^n \xrightarrow{c?\bar{d}_{\langle m:k\rangle}} \delta \rhd k\{\![M\{^{\bar{d}}/_{\bar{x}}\}]\!\}^n}$$

$$\textsc{Com}_1\ \frac{\delta \rhd S \xrightarrow{(\bar{b})c!\bar{d}_{\langle \mu:k\rangle}} \delta' \rhd S' \qquad \delta \rhd V \xrightarrow{c?\bar{d}_{\langle \mu:l\rangle}} \delta \rhd V'}{\delta \rhd S \parallel V \xrightarrow{\tau_{\langle \mu:k,l\rangle}} \delta' \rhd \mathsf{new}\,\bar{b}.(S' \parallel V')}\ [\bar{b} \cap \textsf{FN}(V) = \emptyset]$$

Figure 5.13: Communication in mDPi

Rules ($\textsc{Out}_p$) and ($\textsc{In}_p$) describe the ability of configurations to perform process output and input. The former interacts with the latter to describe a *synchronous* form of process communication with *side effects*. Rule ($\textsc{Out}_p$) describes the ability of system $k[\![c!\bar{d}.P]\!]$ to evolve to $k[\![P]\!]$ while effecting action $c!\bar{d}_{\langle p:k\rangle}$. The novel aspect of this rule is that process output also generates trace entity $k[\![\mathbf{t}(c, \bar{d}, \delta(k))]\!]$ as a side effect, *permanently* logging information of output of $\bar{d}$ on c at k. The generated trace is assigned the *next* counter value to be assigned at location k, obtained from the counter state (through $\delta(k)$). The value mapped by the counter state for k is subsequently *incremented* (through inc) in the residual configuration. Rule ($\textsc{Out}_p$) is hence responsible for formalising the generation of a temporal order on trace entities *per location*. On the other hand, process input is described through rule ($\textsc{In}_p$) is standard [47, 76, 76], with $d \in \bar{d}$ received on c being substituted for $x \in \bar{x}$ in P. Both actions are tagged with $(p : k)$, denoting that the action was effected by a *process* at k.

Monitor communication is (i) *synchronous*, and (ii) *side effect free*. Clearly, we are not interested in verifying monitor behaviour in the same way we are for processes, which is why we do not log monitor actions. We opt for synchronous communication to facilitate synchronisation between monitors. Monitor output and input operations are described through rules ($\textsc{Out}_m$) and ($\textsc{In}_m$) in the standard manner [47, 76, 76]. In this case, both actions are tagged with $(m : k)$, describing a *monitor action* at k.

Rule ($\textsc{Com}_1$) is central to our semantics as it describes *both* process communication as well as monitor communication (we elide symmetric rule ($\textsc{Com}_2$); see appendix A). For communicating sub-systems to synchronise, the rule requires that the corresponding input and output labels agree on the channel of communication, c, the values communicated $\bar{d}$ (these two are standard) but also on their *modality*, μ: this way monitor actions having modality m should not interfere/interact with normal process communication carrying labels with modality p. Note that ($\textsc{Com}_1$) does not require location information of input/output actions to match, thereby permitting cross-location communication for both processes and monitors. Importantly though, it aggregates source and destination location information relating to communication as part of the τ label from these premise labels. Note the side condition on exported names $\bar{b}$ in order to avoid unwanted capture of free names.

$$\textsc{Trace}_e\ \frac{}{\delta \rhd k[\![\mathbf{t}(c,\bar{d},n)]\!] \xrightarrow{\mathrm{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta \rhd k[\![\mathbf{t}(c,\bar{d},n)]\!]}$$

$$\textsc{MonTr}_i\ \frac{}{\delta \rhd k\{\![\mathbf{m}(c,\bar{x},l).M]\!\}^{n} \xrightarrow{\mathrm{m}(c,\bar{d},l,n)_{\langle m:k\rangle}} \delta \rhd k\{\![M\{^{\bar{d}}/_{\bar{x}}\}]\!\}^{n+1}}$$

$$\textsc{Mon}_1\ \frac{\delta \rhd S \xrightarrow{(\bar{b})\mathrm{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta \rhd S \qquad \delta \rhd V \xrightarrow{\mathrm{m}(c,\bar{d},k,n)_{\langle m:l\rangle}} \delta \rhd V'}{\delta \rhd S \parallel V \xrightarrow{\tau_{\langle t:k,l\rangle}} \delta \rhd \mathsf{new}\,\bar{b}.(S \parallel V')}\ [\bar{b} \cap \textsf{fn}(V) = \emptyset]$$

Figure 5.14: Trace Analysis

We next take a look at formalising *trace analysis* in mDPi. This form of communication is *asynchronous* and *persistent*, such that the contributing trace entity is *never* consumed. Rule ($\textsc{Trace}_e$) describes the capability of trace entity $k[\![\mathbf{t}(c,\bar{d},x)]\!]$ to *expose* logged information by performing action $\mathrm{t}(c,\bar{d},n)_{\langle t:k\rangle}$. This label exposes that the trace entity logged some process output of $\bar{d}$ on c. Importantly, the action also includes counter value n encoding the position of the trace entity in the local trace at k. This is counter is used to synchronise monitors and traces with respect to the extracted temporal ordering of events (more below). The above label is tagged with $(t : k)$, denoting an external *trace action* at k. From this we can also infer that the process action logged by the trace entity also occurred at k.

Rule (MonTr$_i$) describes a monitor's capability to read trace entity information through operator **m**. Monitor trace input exhibits analogous behaviour to process and monitor input capabilities, with the exception of alternate judgement $\mathrm{m}(c, \bar{d}, l, n)_{\langle m:k\rangle}$. Apart from standard information including the communication channel and the values communicated, this label also adds a *snapshot* of the monitor's counter value n for location l at the instant of trace input. Its purpose is to record the *next* trace entity (located at l) required by the monitor at that instant during its computation, also used for synchronisation purposes (below). Clearly, given that the action is performed by a monitor (at k) it is tagged with $(m : k)$.

The actions derived from the last two rules interact to perform *trace analysis*, formalised by rule (Mon$_1$) (and symmetric rule (Mon$_2$); see appendix A). Once created, trace entities *repeatedly* expose logged information (giving the act an asynchronous flavour), which is eventually input by monitors. The rules once more require the corresponding trace exposure and monitor trace input labels to agree on (i) the communication channel c, (ii) value tuple $\bar{d}$ (as is standard), but also on (iii) the counter value n and (iv) location of trace exposure k matched with the location whose trace the monitor intends to analyse. Intuitively, matching said actions additionally on their counter value and intended trace location forces monitors to input the n^{th} trace element (located at required location k) during each stage in its computation. It is important to note that we are *not* required to match actions on their effected location (*i.e.*, $k \neq l$ in certain cases), implying that remote trace analysis is allowed. Moreover, we require the input and output actions to be of modality m and t respectively. The resulting trace analysis act is represented by an internal τ action aggregating location information, and is assigned modality t. This choice allows us to differentiate between monitor communication and trace analysis. Had we tagged the action with modality m, this would not have been possible. Note the presence of the side condition, enforced to avoid unwanted capture of free names in the residual.

$$\textsc{Go}_m\ \frac{}{\delta \rhd k\{\!\{\mathsf{go}\ l.M\}\!\}^n \xrightarrow{\tau_{\langle m:k,l\rangle}} \delta \rhd l\{\!\{M\}\!\}^{\delta(l)}}$$

$$\textsc{SetC}_m\ \frac{}{\delta \rhd k\{\!\{\mathsf{setC}(l).M\}\!\}^n \xrightarrow{\tau_{\langle m:k\rangle}} \delta \rhd k\{\!\{M\}\!\}^{\delta(l)}}$$

$$\textsc{Inc}_m\ \frac{\delta \rhd S \xrightarrow{(\bar{b})\mathrm{t}(c_1,\bar{d},n)_{\langle t:l\rangle}} \delta \rhd S \qquad \delta \rhd k\{\!\{M\}\!\}^n \xrightarrow{\mathrm{m}(c_2,\bar{e},l,n)_{\langle m:k\rangle}} \delta \rhd k\{\!\{M'\}\!\}^{n+1}}{\delta \rhd S \parallel k\{\!\{M\}\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \rhd S \parallel k\{\!\{M\}\!\}^{n+1}}\ [c_1 \neq c_2]$$

Figure 5.15: Extraction Of Temporal Orderings During Monitor Execution

Rule (Go$_m$) describes a monitor's ability to migrate from locations k to l through operator go, and is represented through monitor internal action $\tau_{\langle m:k,l\rangle}$ (also aggregating source and destination information). Upon migration, the monitor's counter is re-aligned to $\delta(l)$, the value mapped by the counter state for the new location. (SETC$_m$) describes operator SetC's behaviour, forcing the monitor counter's re-alignment to a specific location passed as a parameter. Finally, rule (INC$_m$) describes monitor behaviour when the trace entity at current counter value n (at location l) does not concern the channel currently of interest to the monitor (represented through side condition $c_1 \neq c_2$). In such cases, the monitor simply increments its counter value, thus ignoring the trace entity. This behaviour is represented by a monitor internal action, tagged with the same source and destination location (since only one location contributed to its execution).

$$\textsc{Rec}_p\ \frac{}{\delta \triangleright k[\![*P]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P \parallel *P]\!]} \qquad \textsc{Rec}_m\ \frac{}{\delta \triangleright k\{\![*M]\!\}^n \xrightarrow{\tau\langle m:k,k\rangle} \delta \triangleright k\{\![M \parallel \mathsf{setC}(k).\,*M]\!\}^n}$$

$$\text{EQ}_p\ \frac{}{\delta \triangleright k[\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P]\!]}\ [u = v]$$

$$\text{NEQ}_p\ \frac{}{\delta \triangleright k[\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![Q]\!]}\ [u \neq v]$$

$$\text{EQ}_m\ \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^n \xrightarrow{\tau\langle m:k,k\rangle} \delta \triangleright k\{\![M]\!\}^n}\ [u = v]$$

$$\text{NEQ}_m\ \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^n \xrightarrow{\tau\langle m:k,k\rangle} \delta \triangleright k\{\![N]\!\}^n}\ [u \neq v]$$

Figure 5.16: Other Operators

The above rules describe the operation of the *recursion* and *branching* operators; both exhibit almost identical behaviour for both processes and monitors. Rules (REC$_p$) and (REC$_m$) describe the unraveling of a recursive call by creating a copy of the program (or monitor) in parallel. In case of processes, the new copy is not bound by operator $*$, and can compute freely. However, in case of monitor recursion the new monitor copy is forced to update its counter through SetC before executing. Had this not been the case *each* copy would inherit counter value n assigned to $*M$. Thus, each copy would start its analysis from the same point in the trace. However, although we want to describe recursive computation for monitors, we want each copy to progress in its analysis of the generated trace, which is why we use SetC. Both rules hence allow for the generation of an arbitrary amount of copies of the original entity. Rules (EQ$_p$), (NEQ$_p$), (EQ$_m$) and (NEQ$_m$) are straightforward, allowing for the analysis of value tuples through the conditional operator. Given branches P, Q (or M, N in the case of monitors), the composite process (or monitor) chooses the next branch depending on the outcome of $u\!=\!v$.

$$\textsc{Open}_s \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})c!\bar{d}_{\langle\mu:k\rangle}} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\, b.S \xrightarrow{(b,\bar{b})c!\bar{d}_{\langle\mu:k\rangle}} \delta' \triangleright S'} \; [b \in \bar{d}]$$

$$\textsc{Open}_t \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\, b.S \xrightarrow{(b,\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta' \triangleright S'} \; [b \in \bar{d}]$$

$$\textsc{Split}_p \; \frac{}{\delta \triangleright k[\![P \parallel Q]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright k[\![P]\!] \parallel k[\![Q]\!]}$$

$$\textsc{Split}_m \; \frac{}{\delta \triangleright k\{\![M \parallel N]\!\}^n \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright k\{\![M]\!\}^n \parallel k\{\![N]\!\}^n}$$

$$\textsc{Exp}_p \; \frac{}{\delta \triangleright k[\![\mathsf{new}\, c.P]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright \mathsf{new}\, c.k[\![P]\!]}$$

$$\textsc{Exp}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{new}\, c.M]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright \mathsf{new}\, c.k\{\![M]\!\}^n}$$

Figure 5.17: Structural Re-organisation

Rules ($\textsc{Open}_s$) and ($\textsc{Open}_t$) handle scope extrusion in the standard way [47, 76, 76], by allowing for new names to be exported over output or trace exposure actions. Knowledge of these names is eventually made known to residual systems after communication/trace analysis. Both rules require exported names to be part of the value tuple being communicated/exported, in order to avoid unintended variable capture. Rules ($\textsc{Split}_p$), ($\textsc{Split}_m$), ($\textsc{Exp}_p$) and ($\textsc{Exp}_m$) serve housekeeping purposes, by pushing out parallel composition and scoping to the system level.

$$\textsc{Cntx}_1 \; \frac{\delta \triangleright S \xrightarrow{\alpha} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\, b.S \xrightarrow{\alpha} \delta' \triangleright \mathsf{new}\, b.S'} \; [b \notin \textsf{FN}(\alpha)]$$

$$\textsc{Cntx}_2 \; \frac{\delta \triangleright S \xrightarrow{\alpha} \delta' \triangleright S'}{\delta \triangleright S \parallel V \xrightarrow{\alpha} \delta' \triangleright S' \parallel V} \; [\textsf{BN}(\alpha) \cap \textsf{FN}(V) = \emptyset]$$

$$\textsc{Cntx}_3 \; \frac{\delta \triangleright S \xrightarrow{\alpha} \delta' \triangleright S'}{\delta \triangleright V \parallel S \xrightarrow{\alpha} \delta' \triangleright V \parallel S'} \; [\textsf{BN}(\alpha) \cap \textsf{FN}(V) = \emptyset]$$

Figure 5.18: Systems In Context

Finally, rules ($\textsc{Cntx}_1$), ($\textsc{Cntx}_2$) and ($\textsc{Cntx}_3$) allow for system behaviour to be deduced when placed in a context. ($\textsc{Cntx}_1$) and ($\textsc{Cntx}_2$) describe system behaviour when placed in parallel, whereas ($\textsc{Cntx}_3$) describes system behaviour when scoped. In all three cases side conditions ensure no unwanted capture of free names as a byproduct.

Example 5. We can now formally express the execution of example system $S_1 = k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$ introduced at the beginning of the chapter. By rules $\textsc{Out}_p$ and $\textsc{Cntx}_2$ we infer transition

$$\{(k,1),(l,1)\} \triangleright k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \xrightarrow{c!\bar{d}_{\langle p:k\rangle}}$$
$$\{(k,2),(l,1)\} \triangleright k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$$

Note the introduction of trace entity $k[\![\mathbf{t}(c,\bar{d},1)]\!]$ recording the process output of $\bar{d}$ on c at k. The counter state has also been updated, incrementing the value mapped to k by 1. Hence, the next trace entity created at k will be assigned the value 2, encoding a temporal ordering on traces. At this stage the trace entity can repeatedly expose its information, represented by transition

$$\{(k,2),(l,1)\} \triangleright k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \xrightarrow{\mathrm{t}(c,\bar{d},1)_{\langle t:k\rangle}}$$
$$\{(k,2),(l,1)\} \triangleright k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$$

by rule $\textsc{Trace}_2$, allowing for the trace to be analysed by various monitors. Moreover, monitor $k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$ exhibits the capability to *read* a trace entity *i.e.*, by $\textsc{MonTr}_i$,

$$k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \xrightarrow{\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}} k\{\![\mathsf{ok}]\!\}^2$$

Hence the monitor requests a trace entity (i) at location k, (ii) at trace order location 1, (iii) regarding output on c. All three conditions happen to be satisfied by the generated trace entity. By rules MON and CNTX$_3$ we derive transition

$$\{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \xrightarrow{\tau_{\langle t:k,k\rangle}} \{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathsf{ok}]\!\}^2$$

thus representing trace analysis as an internal τ action. Moreover, the action is tagged with modality t, and locations k, k since both the monitor and the trace resided at the same location during analysis. The full pre-LTS for S_1 is presented in Fig. 5.19.

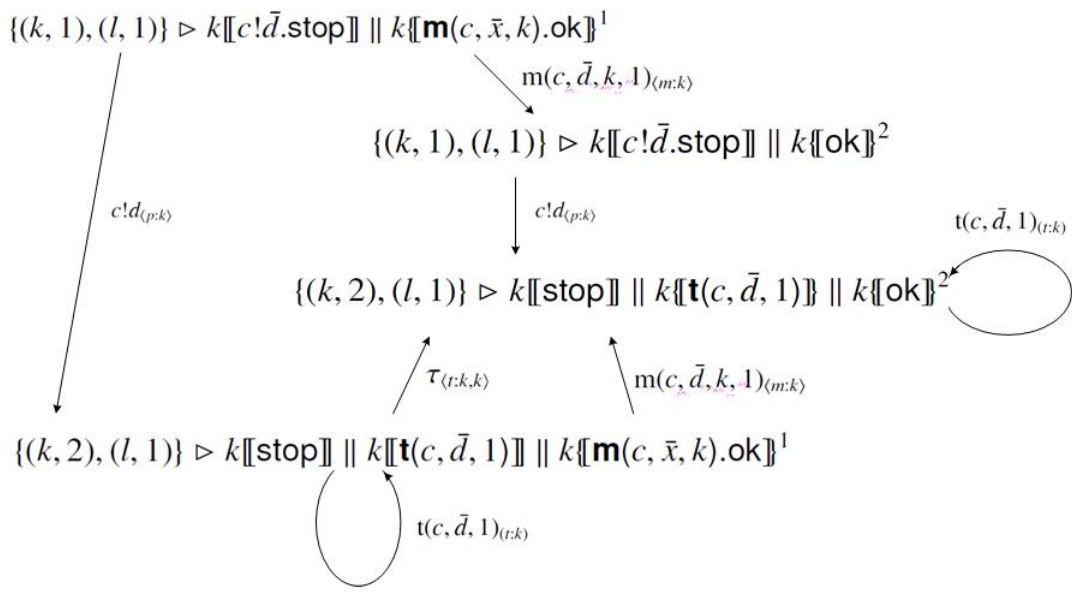

$\{(k,1),(l,1)\} \rhd k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$
$\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}$
$\{(k,1),(l,1)\} \rhd k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathsf{ok}]\!\}^2$
$c!d_{\langle p:k\rangle}$
$c!d_{\langle p:k\rangle}$
$\mathrm{t}(c,\bar{d},1)_{\langle t:k\rangle}$
$\{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathsf{ok}]\!\}^2$
$\tau_{\langle t:k,k\rangle}$
$\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}$
$\{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$
$\mathrm{t}(c,\bar{d},1)_{\langle t:k\rangle}$

Figure 5.19: Pre-LTS representation for S_1

■

Can we, at a mathematical level, identify differences in the behaviour of S_1 and system $S_2 = \{(k,1),(l,1)\} \rhd k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1$? Consider the pre-LTS for S_2 in Fig. 5.20.

Although the pre-LTSs for S_1 and S_2 are very similar, they admit minute differences. Consider the difference in location information for labels $\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}$ and $\mathrm{m}(c,\bar{d},k,1)_{\langle m:l\rangle}$, effected by respective monitors to exhibit their capability of *externally* analysing a trace (*i.e.*, within the context). Since both monitors are at differing locations k and l, the added tags reflect this difference. Another difference lies with actions $\tau_{\langle t:k,k\rangle}$ and $\tau_{\langle t:k,l\rangle}$, representing some internal trace analysis. The difference in location information reflects the fact that the monitor in S_1 reads the trace *locally*, whereas that in S_2 reads the same trace *remotely*. On the other hand,

$$\begin{tikzcd}
\{(k,1),(l,1)\} \rhd k\llbracket c!\bar{d}.\mathsf{stop}\rrbracket \parallel k\{\!|\mathbf{m}(c,\bar{x},k).\mathsf{ok}|\!\}^1 \arrow[dr, "\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}"] \arrow[ddd, "c!d_{\langle p:k\rangle}"'] & \\
& \{(k,1),(l,1)\} \rhd k\llbracket c!\bar{d}.\mathsf{stop}\rrbracket \parallel k\{\!|\mathsf{ok}|\!\}^2 \arrow[d, "c!d_{\langle p:k\rangle}"] \\
& \{(k,2),(l,1)\} \rhd k\llbracket \mathsf{stop}\rrbracket \parallel k\{\!|\mathbf{t}(c,\bar{d},1)|\!\} \parallel k\{\!|\mathsf{ok}|\!\}^2 \arrow[loop right, "\mathrm{t}(c,\bar{d},1)_{\langle t:k\rangle}"] \\
\{(k,2),(l,1)\} \rhd k\llbracket \mathsf{stop}\rrbracket \parallel k\llbracket\mathbf{t}(c,\bar{d},1)\rrbracket \parallel k\{\!|\mathbf{m}(c,\bar{x},k).\mathsf{ok}|\!\}^1 \arrow[ur, bend left, "\tau_{\langle t:k,k\rangle}"] \arrow[ur, bend right, "\mathrm{m}(c,\bar{d},k,1)_{\langle m:k\rangle}"'] \arrow[loop below, "\mathrm{t}(c,\bar{d},1)_{\langle t:k\rangle}"] &
\end{tikzcd}$$

Figure 5.20: Pre-LTS representation for S_2

if we *ignore* tagged location information and only consider actions and their modality, we can *abstract* over differences in the above pre-LTSs. Hence, we can consider S_1 and S_2 to be behaviourally equivalent, as long as we abstract over location information (more below).

This concludes our overview of the pre-LTS semantics. Through the above rules we are able to extract a preliminary LTS representation of system behaviour, whose judgements are tagged with additional information. Clearly, what information is considered pertinent depends on the scenario we wish to consider. In general, the volume of added information results in the semantics being *too discriminating* in most cases — it forces us to distinguish behaviourally between terms which for a certain scenario we want to consider as identical. We therefore require a means which allows for the removal of non-pertinent information in such cases, and is presented below.

Filter Functions

We have so far informally motivated filter functions as a *mechanism* which allows for the consideration of system behaviour at various abstraction levels. This mechanism operates on pre-LTSs, filtering transitions by removing non-pertinent information. Consider system S_1 and its pre-LTS representation in Fig. 5.19. Suppose we want to consider's the subset of S_1's behaviour relating to process actions, as required in scenario 5.3. The resulting LTS looks like Fig. 5.21.

This LTS is derived by ignoring labels tagged with modality m or t *i.e.,* monitor and trace actions. Moreover, since we are only interested in viewing *one* modality of system operation, the modality tags for the remaining process actions are also filtered. The result is an LTS which

$$\begin{tikzcd}
\{(k,1),(l,1)\} \rhd k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \arrow[dr, "c!d_{(k)}"] & \\
& \{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1
\end{tikzcd}$$

Figure 5.21: LTS representation for S_1 obtained by ignoring monitor and trace behaviour.

exhibits process behaviour only. Although mention of traces and monitors are still syntactically present, by filtering their behaviour we give the *illusion* that they do not compute.

Other scenarios may however dictate the need to abstract over different aspects of system behaviour. We may for instance want to define some Ω which filters remote trace analysis, or another Ω which only allows for monitor behaviour. To this effect, the following section focuses on the precise definition of appropriate filter functions. We later make use of these functions in the next section, formalising a *generalised form* of the above derivation process. In other words, we do not tie down to a particular definition of Ω, but rather define a *general derivation mechanism* which we can apply to various filter functions. In general, Ω shall serve two purposes within our calculus:

- To formalise the process of *behaviour information abstraction*, thus obtaining an LTS (from a pre-LTS) encoding a *coarser* view of system behaviour.
- To *reobtain* an extensional view of system behaviour, by reintroducing the silent action at the level of the LTS.

We hence introduce the notion of a *well-formed filter function*, filtering label information in a desirable way. Mirroring the above requirements, Ω is considered well-formed if it adheres to *two* properties *i.e.,*

Definition 5.2.4. *(Well-formed Ω) Filter function Ω is s.t.b. well-formed iff it adheres to the properties of*

- *Visibility Restriction, and;*
- *Action Preservation.*

Our notion of function well formed-ness hence depends on the definition of these two properties. The former intuitively forces filter functions to either (i) remove tags associated to τ, or (ii) *prohibit/ignore* certain τs completely. Hence, the result after filtering a pre-LTSs transitions is the reobtaining of unobservable actions, removing all tagged τs in the process.

Definition 5.2.5. *(Visibility restriction) Function $\mathcal{F}$ is said to be visibility restricting if, for all $\mu : \textsc{Mod}, k, l : \textsc{Locs}$,*

$$\mathcal{F}(\tau_{(\mu:k,l)}) = \tau \ \vee \ \tau_{(\mu:k,l)} \notin dom(\mathcal{F})$$

Informally, the action preserving property forces filter functions to, *at most*, filter information. Given a pre-LTS label $\alpha_{\mu,l}$, we can either (i) filter nothing, or (ii) remove its tagged *modality*, or (iii) remove its *location* information, or (iv) remove *both* modality and tagged information, or (vi) *prohibit* it. Hence, we can define a filter function which maps $c?\bar{d}_{\langle m:k\rangle}$ to $c?\bar{d}_{\langle m\rangle}$ or to $c?\bar{d}_{\langle k\rangle}$, but we *cannot* define Ω to convert $\mathrm{m}(c, \bar{d}, l, n)_{\langle m:k\rangle}$ to $c?\bar{d}_{\langle m:k\rangle}$, since in doing so we would have illegally altered system behaviour.

Definition 5.2.6. *(Action preservation) Function $\mathcal{F}$ is said to be action preserving if, for all $\alpha_{(\mu:k,l)} \in Act_{\mu,l}$*

$$(\mathcal{F}(\alpha_{(\mu:k,l)}) = (\alpha_{(\mu:k,l)} \ \vee \ \alpha_{(\mu)} \ \vee \ \alpha_{(k,l)} \ \vee \ \alpha)) \ \vee \ (\alpha_{(\mu:k,l)} \notin dom(\mathcal{F}))$$

By declaring that certain actions can have an undefined output we are effectively saying that Ω can be a partial function. In the case when we mark actions (either external and internal) as having *undefined* output, we are implying our *disinterest* in considering said behaviour. Hence, behaviour emanating from uninteresting actions is *pruned* from the resulting LTS; both at an internal level, as well as removing how pre-LTSs interact with their context through some *prohibited* external action. We argue this choice to be better than converting unwanted behaviour to silent actions, since by doing so we would still be unnecessarily factoring in interaction with the context. Note that we shall henceforth only consider well-formed filter functions. To avoid repetition, the term 'well-formed' is henceforth assumed.

We next consider a few example definitions of filter functions, starting from Ω_F. This function filters the *least* possible information, by mapping tagged τs to silent actions (only). In doing so, Ω_F returns LTSs containing the *most* information. The result of using Ω_F is hence twofold; (i) comparison of system behaviour using LTSs generated through this filter function is the *most discriminating* (since all *external* actions are *also* matched according to their tagged information), and (ii) we nonetheless *reobtain an extensional view* by abstracting from additional internal action information.

Definition 5.2.7. *(Filter function Ω_F) We define filter function Ω_F as follows*

$$\Omega_F\,\alpha \triangleq \begin{cases} \tau & \textit{if}\, (\alpha = \tau_{\langle m:k,l\rangle}) \vee (\alpha = \tau_{\langle p:k,l\rangle}) \vee (\alpha = \tau_{\langle t:k,l\rangle}) \\ \alpha & \textit{otherwise} \end{cases}$$

Clearly, Ω_F is *well-formed*, since it adheres to the above two properties. Ω_F also happens to be *total*, by mapping each action to another.

Consider next filter function Ω_{RT}, which abstracts all *remote* trace analysis behaviour from a pre-LTS. In general, function Ω_{RT} is useful when we want to consider the *subset* of system execution which does not perform remote trace analysis. Hence, if for example we consider a system's behaviour in full (through Ω_F), and the same system's behaviour through Ω_{RT} and prove that the resulting LTSs are bisimilar, then we would have proven that the system does not perform remote tracing. Given that rules (Mon$_1$) and (Mon$_2$) represent trace analysis through a silent action tagged with modality t, we can easily identify which of these actions involved remote locations.

Definition 5.2.8. *(Filter function Ω_{RT}) We define filter function Ω_{RT} as follows*

$$\Omega_F\,\alpha \triangleq \begin{cases} \tau & \textit{if}\,(\alpha = \tau_{\langle m:k,l\rangle}) \vee (\alpha = \tau_{\langle p:k,l\rangle}) \vee (\alpha = \tau_{\langle t:k,k\rangle}) \\ \textit{undefined} & \textit{if}\,(\alpha = \tau_{\langle t:k,l\rangle}) \\ \alpha & \textit{otherwise} \end{cases}$$

Marking the output for $\tau_{\langle t:k,l\rangle}$ as *undefined* is equivalent to the statement that $\tau_{\langle t:k,l\rangle} \notin dom(\Omega_{RT})$. This filter function once more converts tagged τs to their un-tagged counterpart (in order to satisfy the visibility restricting property). However, note that although trace τ actions of the form $\tau_{\langle t:k,k\rangle}$ (*i.e.*, analysed locally) are filtered, the remainder trace τs have an undefined mapping (and are hence pruned).

Yet another filter function Ω_P shall be defined to remove monitor and trace behaviour from a pre-LTS *i.e.*, allowing only process actions. Thus if for instance we prove that a monitored system's processing behaviour (extracted through Ω_P) is equivalent to the same monitor-less system, then we would have proven that monitoring behaviour does not effect the system's processes. This approach shall be crucial when proving the result that the monitoring semantics do not effect process computation. Clearly, the removal of all monitor and trace behaviour renders modality information redundant (since only process actions are permitted), and is hence also removed.

Definition 5.2.9. *(Filter function Ω_P) We define filter function Ω_P as follows*

$$\Omega_P\,\alpha \triangleq \begin{cases} \tau & \textit{if}\,(\alpha = \tau_{\langle p:k,l\rangle}) \\ c!d_{\langle k\rangle} & \textit{if}\,(\alpha = c!d_{\langle p:k\rangle}) \\ c?d_{\langle k\rangle} & \textit{if}\,(\alpha = c?d_{\langle p:k\rangle}) \\ \textit{undefined} & \textit{if}\,(\alpha = \alpha_{\langle m,k:l\rangle}) \\ \textit{undefined} & \textit{if}\,(\alpha = \alpha_{\langle t,k:l\rangle}) \end{cases}$$

Ω_P handles process actions in straightforward fashion, removing process τ tags, as well as external process actions' modalities. However, all monitor and trace actions of the form $\alpha_{\langle m,k:l\rangle}$ or $\alpha_{\langle t,k:l\rangle}$ respectively are ignored, hence pruning all pre-LTS behaviour concerning either monitors or traces. We have informally seen the use of Ω_P on the pre-LTS for S_1 at the beginning of the section. Moreover, scenario 5.3 argued that systems S_1 and S_2 should admit identical behaviour, disregarding monitor behaviour. Can we confirm this observation on LTSs? Consider Fig. 5.22, which depicts the LTS representation obtained for S_2 through Ω_P.

$$\begin{tikzcd}
\{(k,1),(l,1)\} \rhd k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel l\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1 \arrow[d, "c!d_{\langle k\rangle}"] \\
\{(k,2),(l,1)\} \rhd k[\![\mathsf{stop}]\!] \parallel k[\![\mathbf{t}(c,\bar{d},1)]\!] \parallel l\{\![\mathbf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1
\end{tikzcd}$$

Figure 5.22: LTS representation for S_2 obtained through Ω_P.

By comparing Fig. 5.21 and Fig. 5.22 we can intuitively see that both systems admit the same (filtered) behaviour. Both S_1 and S_2 exhibit the capability for action $c!\bar{d}_{\langle p:k\rangle}$ after filtering, implying that processes in S_1 and S_2 behave in identical fashion. In other words, by ignoring monitoring behaviour we *abstracted over* differences in S_1 and S_2's monitoring efforts. Clearly, the above LTSs show a *subset* of the information encoded by the previous pre-LTSs. A more formal treatment of behaviour comparison (for both LTSs and pre-LTSs) is presented in section 5.3.2.

Given that filtered actions contain varying levels of information, we can deduce an *ordering* on said actions. For instance, actions tagged with both modality *and* location information are more informative than those tagged with either modality *or* location only. Moreover, actions tagged with either modality *or* location only are mutually incomparable; it doesn't make sense to say who has more information. The result is a partial order on action labels, depicted in Fig. 5.23.

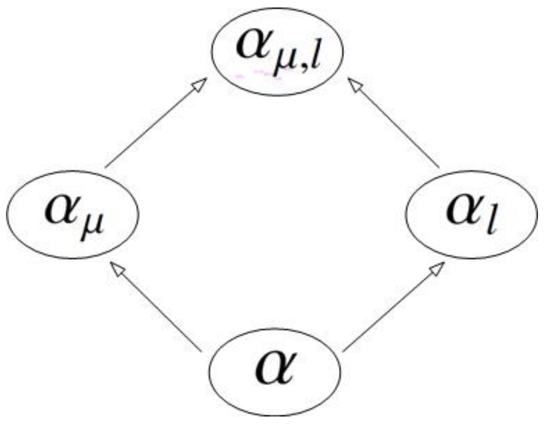


Figure 5.23: Order $\leq$ on actions

We next extend $\leq$ to filter functions. In general, filter functions either pair tagged actions (from a pre-LTS label) with a less informative equivalent, or prohibit certain actions completely. Hence, if $\Omega_1 \leq \Omega_2$ this implies that the former *consistently* filters more information than the latter; see *Def$\underline{^n}$* 5.2.10.

Definition 5.2.10. *(Ordering On Filter Functions) Given well-formed filter functions* Ω_1, Ω_2*, we say* $\Omega_1 \leq \Omega_2$ *iff*

$$(\alpha_1, \alpha_2) \in \Omega_1 \Rightarrow (\alpha_1, \alpha_3) \in \Omega_2 \textit{ s.t. } \alpha_2 \leq \alpha_3$$

The above definition also allows for Ω_2 to pair more actions than Ω_1. What happens if $\Omega_1 \nleq \Omega_2$, and vice versa? This implies that Ω_1, Ω_2 filter *different kinds* of information, implying that we have a partial order on filter functions. An example of mutually incomparable filter functions involve some Ω_1 which filters modality, and Ω_2 which filters location tags. We consider the following ordering on the filter functions defined above

$$\Omega_P \leq \Omega_{RT} \leq \Omega_F$$

Ω_F is the largest by filtering the *least* information, followed by Ω_{RT} (which ignores remote trace analysis), followed by Ω_P ignoring all monitor and trace activity. Although the use of filter function ordering is admittedly unclear at this point, its utility becomes apparent when comparing LTSs in the next section. More specifically, the extraction of a hierarchy on LTSs allows for (i) a hierarchy on their comparison (section 5.3.2), and (ii) its exploitation for proof reuse (exemplified in results 5.3.1 and 5.3.2).

The LTS

We next formalise the extraction of an LTS, representing *behaviour* for given system S at a particular level of abstraction. As equation 5.1 suggests, we require (i) a *pre-LTS* representation for S, and (ii) a *filter function* Ω. Filter functions operate at the semantic level, by altering transition labels from the pre-LTS. Intuitively, we obtain an LTS by filtering *each* pre-LTS transition of the form $C_1 \xrightarrow{\alpha} C_2$; thereby converted to $C_1 \xrightarrow{\alpha'} C_2$ such that $\Omega(\alpha) = \alpha'$. We shall use notation $C \xrightarrow{\alpha}_{\Omega} C'$ to refer to *filtered transitions*, signifying that transition $C \xrightarrow{\alpha'} C'$ is filtered through Ω. By extension, LTS_{Ω} refers to an LTS obtained through Ω.

Definition 5.2.11. *(Filtered Transitions) Given transition $C_1 \xrightarrow{\alpha} C_2$ s.t. $\alpha \in Act_{\mu,l}$, well-formed filter function Ω, the corresponding filtered transition is defined as*

$$\textsc{F-Tran}\ \frac{C_1 \xrightarrow{\alpha} C_2}{C_1 \xrightarrow{\alpha'}_{\Omega} C_2}\ [\Omega(\alpha) = \alpha']$$

Extracting an LTS from a pre-LTS is thereby a straightforward process, highlighting the attractiveness of our approach. In order to obtain an LTS representation, we simply filter the pre-LTS' transitions, and associated action labels.

Definition 5.2.12. *(Extraction of an LTS) Given pre-LTS $(C, \textsc{Act}, \rightarrow)$ and well-formed filter function Ω, we define an LTS $(C', \textsc{Act}', \rightarrow')$ as follows*

- $C' \triangleq C$
- $\textsc{Act}' \triangleq \{x \in \textsc{Act} \bullet \Omega(x)\}$
- $\rightarrow' \triangleq \{C_1 \xrightarrow{\alpha} C_2 \in \rightarrow \mid \Omega(\alpha) = \alpha' \bullet C_1 \xrightarrow{\alpha'}_{\Omega} C_2\}$

The result after filtering is another transition system, implying that standard LTS considerations (including $\approx$) can be applied. The use of well-formed filter functions give rise to the notion of a *well-behaved LTS*.

Definition 5.2.13. *(Well-behaved LTS) An LTS is s.t.b. well behaved iff*

- *The LTS is faithful to the original pre-LTS, at most filtering information i.e., for all LTS transitions $P \xrightarrow{\alpha}_{\Omega} Q$ this implies that $P \xrightarrow{\alpha'} Q$ and $\alpha \leq \alpha'$.*
- *It removes tagged τ actions.*

We argue that the resulting LTS should abide by the above two properties in order to make sense. If an LTS is not faithful to its pre-LTS, this implies that (a subset of) the former's transitions express different behaviour than its pre-LTS, thus breaking the bond between the two. Also, the second property promotes an extensional view of system behaviour by abstracting over details of internal behaviour. Fortunately, both properties are safeguarded through the application of well-formed Ω.

Section 5.2.2 motivated orderings on both actions and filter functions. We argue that abstracting more information leads to *coarser* forms of behaviour — the more information we ignore the more systems we *cannot* distinguish. In general, given system S we can extract a hierarchy on its LTS representations. This hierarchy ranges over all valid definitions of Ω. Assuming filter functions Ω_1, Ω_2 *s.t.* $\Omega_1 \leq \Omega_2$, LTSs generated by the former are therefore *smaller* than those generated by the latter. In other words, the transitions of LTS_{Ω_1} for S contain less than or an equal amount of information than LTS_{Ω_2} for the same S. Moreover, the former LTS might even be a *pruned* version of the latter, since filter functions can choose to ignore transitions. This gives rise to an ordering on LTSs.

Definition 5.2.14. *(Ordering on LTSs) Given* LTS_{Ω_1}, LTS_{Ω_2} *s.t.* $\Omega_1 \leq \Omega_2$*, then for all transitions* $C_1 \xrightarrow{\alpha_1}_{\Omega_1} C_2$*,* $C_1 \xrightarrow{\alpha_2}_{\Omega_2} C_2$ *we say* $LTS_{\Omega_1} \leq LTS_{\Omega_2}$ *iff*

$$C_1 \xrightarrow{\alpha_1}_{\Omega_1} C_2 \textit{ implies } C_1 \xrightarrow{\alpha_2}_{\Omega_2} C_2 \textit{ and } \alpha_1 \leq \alpha_2$$

This observation can be extended to pre-LTSs, which will always contain the *most* information. The pre-LTS hence serves as a *greatest fixed point* in the above hierarchy. We believe that the partial ordering on filter functions (section 5.2.2) maps directly to a *hierarchy* on LTSs. In other words, in order to determine whether an LTS is more informative than another (assuming they represent the same system's behaviour), we simply need to compare their filter functions. Using the above example hierarchy on a selection of filter functions, we deduce that $\textit{Pre-LTS} \geq LTS_{\Omega_F} \geq LTS_{\Omega_{RT}} \geq LTS_{\Omega_P}$. Let us assume another filter function Ω_M, which generates LTSs admitting only monitoring behaviour (defined in section 5.4). Clearly, Ω_P is mutually incomparable to Ω_M, since both functions admit a disjoint set of action pairs (the former admits process actions exclusively, whereas the latter admits monitor actions). However, $\Omega_{RT} \geq \Omega_M$ since although monitor behaviour is unaltered by Ω_{RT}, trace behaviour is removed by Ω_M. This gives rise to the hierarchy in Fig. 5.24.

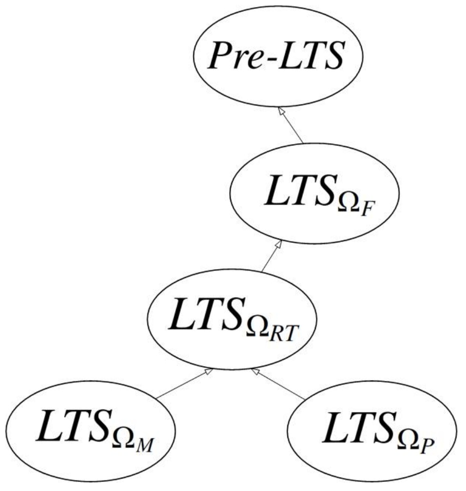


Figure 5.24: An Example Hierarchy On LTSs

The hierarchy on LTSs can be exploited when proving results. More specifically, we make use of the observation that higher abstraction levels depict a *subset* of behaviour exhibited at a lower level. Hence, if we prove a statement when considering *more* information, this statement should also hold when faced with *less* (information). Finally, given that a pre-LTS always contains the *most* information, then proving a statement at the level of the pre-LTS implies truth of the statement for *any* LTS, irrespective of the filter function used. This technique is exemplified in results 5.3.1 and 5.3.2. We first prove the former result at the pre-LTS level, and then show that the result holds irrespective of the filter function used.

5.3 Configuration Equality

The mDPi syntax introduced in section 5.2.1 is too *discriminating* — it distinguishes between terms which we may, at least in certain situations, choose to consider the same. Certainly, reasoning about configuration equivalence is a vital tool allowing for a formal analysis of the language properties. Therefore, we require a precise way how to identify equivalent configurations. What configurations we consider to be equivalent can take multiple forms. We have so far introduced α-equivalence, a more basic form of equality pairing identical systems, except in their naming of bound variables. The following section introduces two additional forms; the first which considers configuration equivalence based on *structure*, followed by a more refined form of *behavioural* equality.

5.3.1 Structural Equivalence

We now define structural equivalence $\equiv$, abstracting over inessential details of system structure without affecting its meaning.

Definition 5.3.1. *(Structural Equivalence $\equiv$) Structural equivalence $\equiv :: \textsc{Sys} \leftrightarrow \textsc{Sys}$ is defined as the least relation which (i) extends α-equivalence, (ii) is an equivalence relation, (iii) is contextual, and (iv) satisfies the following equalities*

(S-Extr)	$\mathsf{new}\, c.(S_1 \parallel S_2)$	$\equiv$	$S_1 \parallel \mathsf{new}\, c.S_2$	*if* $c \notin fn(S_1)$
(S-Com)	$S_1 \parallel S_2$	$\equiv$	$S_2 \parallel S_1$	
(S-Assoc)	$S_1 \parallel (S_2 \parallel S_3)$	$\equiv$	$(S_1 \parallel S_2) \parallel S_3$	
(S-Stop$_1$)	$S \parallel k[\![\mathsf{stop}]\!]$	$\equiv$	S	
(S-Stop$_2$)	$\mathsf{new}\, c.k[\![\mathsf{stop}]\!]$	$\equiv$	$k[\![\mathsf{stop}]\!]$	
(S-Flip)	$\mathsf{new}\, c_1.\mathsf{new}\, c_2.S$	$\equiv$	$\mathsf{new}\, c_2.\mathsf{new}\, c_1.S$	

The above equalities are an extension of *Def$^{\underline{n}}$* 4.3.1, applied to the standard calculus. Rules (S-Com) and (S-Assoc) define commutativity and associativity of parallel composition (at the system level). Rules (S-Stop$_1$) and (S-Stop$_2$) serve as a form of garbage collection, removing terminal processes and unnecessary bound names. Rule (S-Flip) negates the order of defined bound channels. Finally rule (S-Extr) describes scope extrusion at a structural level. If name c

bound in S_2 is not free in S_1, then knowledge of c can be extruded to the latter. Clearly, if c is free in S_1, exporting its name should be disallowed to avoid inadvertent name capture.

Structural equivalence has so far been defined at the system level, which is however only part of a configuration's contents. What about the counter state? Clearly, the counter state can be considered part of a system's structure, due to its role in guiding system execution. Therefore, we argue that it does *not* make sense for structurally equivalent systems to admit different counter states. As a result of this design choice, we can later prove that computation emerging from structurally equivalent systems always matches (with the residual systems also equivalent), a desirable property of $\equiv$. We elevate the definition of structural equivalence over configurations *i.e.,* $\equiv :: \textsc{Conf} \leftrightarrow \textsc{Conf}$, such that

$$S_1 \equiv S_2 \text{ implies } \delta \triangleright S_1 \equiv \delta \triangleright S_2$$

thus matching δ on both sides. Given that δ is a function, proving equality of counter states can be *inferred* from transition rules, or *achieved* through standard theory of relations [82].

We next present our first set of results, relating behaviour of structurally equivalent configurations. These intermediate results shall prove useful when reasoning about more involving proofs in section 5.4. Lemma 5.3.1 proves that structurally equivalent configurations can immediately match each other's computation, with the residual configurations also being structurally equivalent. This result has important repercussions; if structurally equivalent configurations always match in their computation, then switching one configuration for the latter should not affect overall behaviour (see lemma 5.3.5). Given that we are dealing with unfiltered transitions, the proof proceeds at the level of the pre-LTS.

Lemma 5.3.1. $((C_1 \equiv C_2) \wedge (C_1 \xrightarrow{\alpha} C')) \Rightarrow ((C_2 \xrightarrow{\alpha} C'') \wedge (C' \equiv C''))$

Proof. The proof proceeds by induction on the derivation of $(C_1 \equiv C_2)$. In each case, we consider possible transitions from C_1, and prove that (i) at least one matching transition from C_2 exists, and (ii) the residual configurations C', C'' are structurally equivalent. Given that the proof is at the level of the pre-LTS, matching of transition labels involves matching (i) the action (including the communication channel and data tuple where applicable), (ii) the action modality, and (iii) location information. See appendix E for a complete listing of the proof, admitting eight base cases (one for each equality in *Def$^{\underline{n}}$* 5.3.1, and another implicit case due to reflexivity of $\equiv$), and four inference cases; two resulting from symmetry and transitivity of $\equiv$, and another two due to $\equiv$ being contextual. We present an example case below.

• (S-Com) : $\delta \triangleright S_1 \parallel S_2 \equiv \delta \triangleright S_2 \parallel S_1$

Hence, this case considers the situation where $C_1 = \delta \triangleright (S_1 \parallel S_2)$. We are therefore required to prove that $((\delta \triangleright S_1 \parallel S_2 \equiv \delta \triangleright S_2 \parallel S_1) \wedge ((\delta \triangleright S_1 \parallel S_2) \xrightarrow{\alpha} C')) \Rightarrow (((\delta \triangleright S_2 \parallel S_1) \xrightarrow{\alpha}$

$C'') \wedge (C' \equiv C''))$. The structure of configuration $\delta \rhd S_1 \parallel S_2$ indicates the possible use of one of six rules for the derivation of $(\delta \rhd S_1 \parallel S_2) \xrightarrow{\alpha} C'$. We shall be considering each case.

–(Cntx₂): The rule dictates that in this case $C' = \delta' \rhd S_1{}' \parallel S_2$, such that $(\delta \rhd S_1 \parallel S_2) \xrightarrow{\alpha} (\delta' \rhd S_1{}' \parallel S_2)$ because $\delta \rhd S_1 \xrightarrow{\alpha} \delta' \rhd S_1{}'$.

Using rule Cntx₃ we can hence infer $(\delta \rhd S_2 \parallel S_1) \xrightarrow{\alpha} (\delta' \rhd S_2 \parallel S_1{}')$ which gives us the required matching move, since the transitions match and $\delta' \rhd S_2 \parallel S_1{}' \equiv \delta' \rhd S_1{}' \parallel S_2$.

– (Cntx₃): Analogous to the above argument.

– (Com₁): Hence $C' = \delta \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}')$, $\alpha = \tau_{(\mu:k,l)}$ such that $(\delta \rhd (S_1 \parallel S_2)) \xrightarrow{\tau_{(\mu:k,l)}} (\delta' \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}'))$. By Com₁ we can infer transitions *(i)* $\delta \rhd S_1 \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S_1{}'$, *(ii)* $\delta \rhd S_2 \xrightarrow{c?d_{\langle p:k\rangle}} \delta \rhd S_2{}'$.

By rule Com₂ we can hence infer $(\delta \rhd (S_2 \parallel S_1)) \xrightarrow{\tau_{(\mu:k,l)}} (\delta' \rhd \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}'))$ which gives us the required matching move, since the transitions match and $\delta' \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}') \equiv \delta' \rhd \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}')$.

– (Com₂): Analogous argument to that given in case of Com₁.

The last two cases deal with the case with monitor trace analysis between S_1 and S_2. Proof of these cases is analogous to that given for system communication, with one difference; counter state remains unaffected during trace import.

– (Mon₁): Hence $C' = \delta \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}')$, $\alpha = \tau_{(t:k,l)}$ such that $(\delta \rhd (S_1 \parallel S_2)) \xrightarrow{\tau_{(t:k,l)}} (\delta \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}'))$. By Mon₁ we can infer transitions *(i)* $\delta \rhd S_1 \xrightarrow{(\bar{b})\mathsf{t}(c,d,n)_{\langle t:k\rangle}} \delta \rhd S_1{}'$, *(ii)* $\delta \rhd S_2 \xrightarrow{\mathsf{m}(c,d,k,n)_{\langle m:l\rangle}} \delta \rhd S_2{}'$.

By rule Mon₂ we can hence infer $(\delta \rhd (S_2 \parallel S_1)) \xrightarrow{\tau_{(t:k,l)}} (\delta \rhd \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}'))$ which gives us the required matching move, since the transitions match and $\delta \rhd \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}') \equiv \delta \rhd \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}')$.

– (Mon₂): Analogous argument to that given in case of Mon₁.

Other cases are equally straightforward, with the exception of one *i.e.,* proof of the case for S-Extr, which follows the argument presented in [46]. □

We next extend this result to configurations whose transitions are filtered. Since we have proven lemma 5.3.1 on unfiltered transitions, the result should hold when dealing with less information. This implies veracity of the result *irrespective* of the filter function used, since by *Def$^{\underline{n}}$* 5.2.4 a filter function can at most filter (*i.e.*, never add) information. Nevertheless, given $S_1 \equiv S_2$, we require the transitions emanating from S_1, S_2 to be filtered through the same Ω. This is necessary since if transitions from both systems are filtered *differently*, then filtered transitions from S_1 might not match those from S_2.

Lemma 5.3.2. $((C_1 \equiv C_2) \wedge (C \xrightarrow{\alpha}_{\Omega} C')) \Rightarrow ((C_2 \xrightarrow{\alpha}_{\Omega} C'') \wedge (C' \equiv C''))$

Proof. We hence know *(i)* $C_1 \equiv C_2$ and *(ii)* $C_1 \xrightarrow{\alpha}_{\Omega} C'$ to be true, and are required to prove *(iii)* $C_2 \xrightarrow{\alpha} C''$ such that *(iv)* $C' \equiv C''$

By the definition of filtered transitions (*Def$^{\underline{n}}$* 5.2.11) we infer that (ii) was derived from a transition of the form $C_1 \xrightarrow{\alpha'} C'$ *s.t.* $\Omega(\alpha') = \alpha$ *...(v)* .

By (v), (i) and lemma 5.3.1 we infer that there exists some C'' such that $C_2 \xrightarrow{\alpha'} C''$ and $C' \equiv C''$ *...(vi)* .

Since Ω is a function, then applying α' to Ω will return the same result α. Hence, we obtain filtered transition $C_2 \xrightarrow{\alpha}_{\Omega} C''$ *...(vii)*.

The result follows by (vi), (vii). □

An immediate corollary of lemma 5.3.2 is given below, instantiating the result for filter function Ω_P. We shall henceforth often consider our results *wrt.* this filter function due to its use in result 5.4.

Corollary 5.3.1. $((C_1 \equiv C_2) \wedge (C_1 \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((C_2 \xrightarrow{\alpha}_{\Omega_P} C'') \wedge (C' \equiv C''))$

Truth of the above statement follows from lemma 5.3.2; instantiating Ω to Ω_P renders the above corollary immediately true. Notice the efficient proof reuse in the above lemmas. We first exhaustively proved truth of the required statement (*i.e.*, transitions from structurally equivalent configurations match) at the level of the pre-LTS (lemma 5.3.1). Subsequent proof of this statement for filtered transitions easily followed; we were not required to provide another exhaustive proof (lemma 5.3.2). Moreover, extending this result to transitions filtered through Ω_P was immediate, suiting our particular needs (see results 5.3.4, 5.3.5, and 5.2). In other words, we successfully exploited the hierarchy on LTSs (including the pre-LTS) to summarise our proofs, as discussed in section 5.2.2. Lemma 5.3.2 can be further instantiated to any filter function definition, as necessary.

5.3.2 Bisimilarity

The definition of a labelled transition system semantics for mDPi yields a straightforward co-inductive proof technique for formal reasoning about system behaviour through bisimilarity relation $\approx$. We are interested in a form of bisimilarity which is *weak up to internal actions* (*i.e., weak bisimilarity*), since we argue it is a form of behavioural equivalence which best fits our requirements (see section 4.4). Weak bisimilarity is based on the notion of the weak action (*Def*$^{\underline{n}}$ 4.4.1); given that transitions can be filtered in case of LTS representations, we define the *weak filtered action* below.

Definition 5.3.2. *(Weak Filtered Action $C_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega} C_2$) Given action α filtered by Ω s.t. $\Omega(\alpha') = \alpha$, a weak filtered action $C_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega} C_2$ is derived as follows.*

$$C_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega} C_2 \triangleq \begin{cases} C_1 \ (\overset{\tau}{\rightarrow}_{\Omega})^* \ C_2 & \alpha = \tau \\ C_1 \ (\overset{\tau}{\rightarrow}_{\Omega})^* \ C_1{}' \ \overset{\alpha}{\rightarrow}_{\Omega} \ C_2{}' \ (\overset{\tau}{\rightarrow}_{\Omega})^* \ C_2 & \alpha \textit{ is an external action} \end{cases}$$

Hence, a weak filtered action is derived from either (i) a sequence of filtered τ actions, or (ii) an external filtered action α (derived from α' through Ω) interspersed by an arbitrary amount of (filtered) silent actions. Clearly, since all transitions have already been filtered, we may go further by considering how each transition was filtered. For instance, an untagged τ must have been derived from its tagged equivalent; same goes for the external actions. The weak filtered action allows for the abstraction over internal actions in case of LTSs.

Relation $\mathcal{R}$ over LTS states is s.t.b. a bisimulation if adheres to the transfer property below.

Definition 5.3.3. *(Bisimulations on LTSs) Given LTSs $(C_1, \textsc{Act}_1, \rightarrow_1)$, $(C_2, \textsc{Act}_2, \rightarrow_2)$ whose transitions are filtered by Ω_1, Ω_2, we say a relation $\mathcal{R} :: C_1 \leftrightarrow C_2$ is a bisimulation if, whenever $(C \ \mathcal{R} \ R)$,*

$$\bullet (C \overset{\alpha}{\rightarrow}_{\Omega_1} C') \textit{ implies } ((R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_2} C'') \textit{ and } (C' \mathcal{R} C''))$$
$$\bullet (R \overset{\alpha}{\rightarrow}_{\Omega_2} C') \textit{ implies } ((C \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_1} C'') \textit{ and } (C' \mathcal{R} C''))$$

In general, we consider LTSs whose transitions are filtered using distinct Ω_1, Ω_2. Hence, the above definition states that an action (filtered through Ω_1) from the former LTS must match a weak filtered action (obtained through Ω_2) from the latter (and vice versa). In other words, we do not care about the filter functions used for deriving our transitions, as long as the action labels match.

We next extend the definition of bisimilarity to LTSs. Given that transitions emanating from configuration C are filtered in the case of LTSs, we shall use notation $\Omega\langle C\rangle$ to refer to configuration C, whose outgoing transitions are filtered through Ω. Informally, given two configurations

$\Omega\langle C_1\rangle$, $\Omega\langle C_2\rangle$, we say that $\Omega\langle C_1\rangle$ is bisimilar to $\Omega\langle C_2\rangle$ (written $\Omega\langle C_1\rangle \approx \Omega\langle C_2\rangle$) if we can exhibit a bisimulation containing the pair $(\Omega\langle C_1\rangle, \Omega\langle C_2\rangle)$.

Definition 5.3.4. *(Bisimilarity relation $\approx$) The bisimilarity relation, denoted by $\approx$, is defined to be the union of all bisimulation relations.*

Hence, relation $\approx$ is taken to be the *largest* bisimulation relation, since all other bisimulations are a subset of $\approx$ (irrespective of the filter functions used for obtaining the LTSs). Conversely, $\approx$ can be considered to range over all combinations of well-formed Ω definitions. We use notation $\approx_\Omega$ to denote the use of $\approx$ while dealing with LTSs whose transitions are filtered by Ω *i.e.*, as shorthand for $\Omega\langle C_1\rangle \approx \Omega\langle C_2\rangle$. Crucially, the reader should not misinterpret this notation into thinking we have defined a novel bisimilarity relation; we are still applying the standard $\approx$. This notation is rather meant for situations when we want to refer to $\approx$ applied to LTSs, without referring to particular $\Omega\langle C\rangle$. Analogously, notation $\Omega\langle C\rangle$ does not imply manipulation of C at a structural level, this notation is simply used to specify that C's transitions are filtered through Ω.

We have so far defined bisimilarity on LTSs by considering their filtered transitions. What about bisimilarity on pre-LTSs? Since their transitions are unfiltered we apply standard definitions of bisimulation (*resp.* bisimilarity); see *Def$^{\underline{n}}$*4.4.2 and 4.4.3. However it so happens that, over pre-LTSs, weak bisimilarity *coincides* with a strong form of bisimilarity $\sim$ [47], since pre-LTSs do not admit silent τ moves. In other words, we are required to match tagged τ actions *also* on their additional tagged information. This side effect can be seen as the unintended consequence of the rather intensional view adopted by the pre-LTS semantics.

Consider systems S_1, S_2 presented in section 5.2.2; We informally motivated their equivalence, disregarding monitor behaviour through filter function Ω_P. The required statement can now be formalised as

$$\{(k,1),(l,1)\} \triangleright S_1 \approx_{\Omega_P} \{(k,1),(l,1)\} \triangleright S_2$$

In order to prove this statement correct, we therefore have to exhibit a bisimulation containing the pair $(\Omega_P\langle\{(k,1),(l,1)\} \triangleright S_1\rangle, \Omega_P\langle\{(k,1),(l,1)\} \triangleright S_2\rangle)$. LTSs representations for both systems can be seen at Fig. 5.21 and Fig. 5.22 respectively. As these figures imply, both systems admit identical behaviour after filtering, implying that the bisimulation can be defined as follows. For brevity we shall refer to $T = \{(k,1),(l,1)\}$ and $T' = \{(k,2),(l,1)\}$.

$$\begin{aligned}\{\ &(T \triangleright k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel k\{\![\mathsf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1, T \triangleright k[\![c!\bar{d}.\mathsf{stop}]\!] \parallel l\{\![\mathsf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1\ ),\\ &(T' \triangleright k[\![\mathsf{stop}]\!] \parallel k[\![\mathsf{t}(c,\bar{d},1)]\!] \parallel k\{\![\mathsf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1\ ,\\ &\qquad\qquad T' \triangleright k[\![\mathsf{stop}]\!] \parallel k[\![\mathsf{t}(c,\bar{d},1)]\!] \parallel l\{\![\mathsf{m}(c,\bar{x},k).\mathsf{ok}]\!\}^1\ )\ \}\end{aligned}$$

Moreover, if we had to also consider *full behaviour* for S_1, S_2 then

$$\{(k,1),(l,1)\} \triangleright S_1 \not\approx_{\Omega_F} \{(k,1),(l,1)\} \triangleright S_2$$

since we cannot exhibit a bisimulation which matches both systems' behaviour. This is due to differences in monitoring location, exhibited in labels $\mathrm{m}(c, \bar{d}, k, 1)_{\langle m:k\rangle}$ and $\mathrm{m}(c, \bar{d}, k, 1)_{\langle m:l\rangle}$. Given that Ω_F (*Def*$^{\underline{n}}$ 5.2.7) does not filter monitoring location, these differences persist at the LTS level. Analogously, at the pre-LTS level (prior to filtering transitions) these differences are more apparent *i.e.*,

$$\{(k, 1), (l, 1)\} \rhd S_1 \not\approx \{(k, 1), (l, 1)\} \rhd S_2$$

See Fig. 5.19 and Fig. 5.20 for a graphical depiction of these differences on transition labels.

We have successfully formalised a *modular* approach for the comparison of system behaviour at various abstraction levels. Moreover, although the above examples have compared systems at the same level (*i.e.*, filtered by the same Ω), we are also often interested in comparing behaviour at different levels of abstraction. Consider $\{(k, 1), (l, 1)\} \rhd S_1$ once more; by proving that

$$\Omega_F\langle\{(k, 1), (l, 1)\} \rhd S_1\rangle \;\approx\; \Omega_{RT}\langle\{(k, 1), (l, 1)\} \rhd S_1\rangle$$

we can conclude that S_1 does not effect trace analysis at a remote level. Informally we are comparing S_1's behaviour in full, and the same S_1's behaviour while prohibiting remote tracing. If the resulting LTSs are bisimilar, then we have proven that no remote tracing could have occurred. Hence, the above statement is intuitively true, since the monitor at S_1 analyses the generated trace locally. However, this also implies that

$$\Omega_F\langle\{(k, 1), (l, 1)\} \rhd S_2\rangle \;\not\approx\; \Omega_{RT}\langle\{(k, 1), (l, 1)\} \rhd S_2\rangle$$

since S_2's monitor analyses the generated trace from $l \neq k$.

Finally, we present some results on the bisimilarity relation $\approx$. Consider first the effect of filter function ordering on $\approx$. We have already established that if $\Omega_1 \subseteq \Omega_2$ then the resulting LTSs from the former are *less* informative that those extracted from the latter. Hence, we can pair *more* configurations through Ω_1 than we can with Ω_2; since LTSs from Ω_2 admit more information, then pairing configurations through the former is *more discriminating*. In other words, by considering more informative LTSs we have to pair transitions on more information. However, the more information we consider, the less likely will their transitions match. This observation can be formalised as a predicate over $\approx$, and is proven below.

Lemma 5.3.3. $(\Omega_1 \leq \Omega_2) \;\Rightarrow\; (\approx_{\Omega_1} \supseteq \approx_{\Omega_2})$

Proof. We know that (i) $(\Omega_1 \leq \Omega_2)$ and are required to prove (ii) $(\approx_{\Omega_1} \supseteq \approx_{\Omega_2})$. Consider relation

$$\mathcal{R} \triangleq \{(C_1, C_2) \mid \Omega_2\langle C_1\rangle \approx \Omega_2\langle C_1\rangle\}$$

The result follows by coinduction if we prove that $\mathcal{R} \subseteq \approx_{\Omega_1}$. In other words, if we prove that a bisimulation exists between configurations $(C_1, C_2) \in \mathcal{R}$ after their transitions have been filtered by Ω_1, then the proof is complete. We are hence required to prove the following two statements:

$$\textit{(1)}\ ((C_1\ \mathcal{R}\ C_2) \wedge (C_1 \xrightarrow{\alpha}_{\Omega_1} C')) \Rightarrow ((C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_1} C'') \wedge (C' \mathcal{R} C''))$$
$$\textit{(2)}\ ((C_1\ \mathcal{R}\ C_2) \wedge (C_2 \xrightarrow{\alpha}_{\Omega_1} C')) \Rightarrow ((C_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_1} C'') \wedge (C' \mathcal{R} C''))$$

Proof of (1):

We know (iii) $C_1\ \mathcal{R}\ C_2$ and (iv) $C_1 \xrightarrow{\alpha}_{\Omega_1} C'$ to be true, and are required to prove (v) ($C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_1} C''$), and (vi) $C' \mathcal{R} C''$.

By (iv) and F-Tran (*Def<u>n</u>* 5.2.11) we infer $C_1 \xrightarrow{\alpha'} C'$ *s.t.* $(\alpha', \alpha) \in \Omega_1$. Since $\Omega_1 \leq \Omega_2$ we can hence infer $(\alpha', \alpha'') \in \Omega_2$ *s.t.* $\alpha \leq \alpha''$ by *Def<u>n</u>* 5.2.10. Applying rule F-Tran to $C_1 \xrightarrow{\alpha'} C'$ we obtain $C_1 \xrightarrow{\alpha''}_{\Omega_2} C'$ *...(v)*.

By (iii) and the *Def<u>n</u>* of $\mathcal{R}$ we infer that $\Omega_2\langle C_1\rangle \approx \Omega_2\langle C_2\rangle$, implying that there exists a bisimulation containing the pair (C_1, C_2). Therefore, by (iii), (v) and the *Def<u>n</u>* of $\mathcal{R}$ we obtain

- $C_2 \overset{\widehat{\alpha''}}{\Rightarrow}_{\Omega_2} C''$ *...(vi)*
- $\Omega_2\langle C'\rangle \approx \Omega_2\langle C''\rangle$ *...(vii)*

By (vii) and the *Def<u>n</u>* of $\mathcal{R}$ we also immediately infer that $C' \mathcal{R} C''$ *...(viii)*

Let us elaborate on the derivation of (vi). By *Def<u>n</u>* 5.3.2 we know that (vi) was derived from a sequence of transitions filtered by Ω_2. However, given knowledge of $(\alpha', \alpha'') \in \Omega_2$ we know that $C_2 \overset{\widehat{\alpha''}}{\Rightarrow}_{\Omega_2} C''$ was derived from a sequence of unfiltered transitions of the form

$$C_2 \xrightarrow{\alpha_1} C_i \xrightarrow{\alpha_2} C_{ii} \xrightarrow{\alpha_3} ...C_j \xrightarrow{\alpha'} C_{j+1} \xrightarrow{\alpha_{j+1}} ... \xrightarrow{\alpha_n} C''$$

s.t. $\Omega_2(\alpha_1...\alpha_n) = \tau$. Due to the properties of visibility restriction (*Def<u>n</u>*5.2.5) and action preservation (*Def<u>n</u>*5.2.6) which Ω_2 adheres to, we infer that $\alpha_1...\alpha_n$ are tagged τ actions.

Our next step is to filter the above transition sequence using Ω_1. Filtering transition $C_j \xrightarrow{\alpha'} C_{j+1}$ is straightforward, since $(\alpha', \alpha) \in \Omega_1$ *i.e.*, resulting in $C_j \xrightarrow{\alpha}_{\Omega_1} C_{j+1}$ (by F-Tran). What about the filtering of tagged τs? Since Ω_1 is well-formed, this implies that the tagged τs are once more converted to their tagless version. This implies that the transition sequence takes the form

$$C_2 \xrightarrow{\alpha_1}_{\Omega_1} C_i \xrightarrow{\tau}_{\Omega_1} C_{ii} \xrightarrow{\tau}_{\Omega_1} ...C_j \xrightarrow{\alpha}_{\Omega_1} C_{j+1} \xrightarrow{\tau}_{\Omega_1} ... \xrightarrow{\tau}_{\Omega_1} C''$$

which collapses to $C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_1} C''$ *...(ix)*

The result follows by (viii) and (ix).

Proof of (2):

Analogous to the first case.

□

Informally, the above result proves that if we know two that LTSs are bisimilar, then smaller LTSs (according to ordering $\leq$ on filter functions) are also bisimilar. This is a powerful concept; by comparing filter functions we can infer an ordering on LTSs, and by extension an ordering on their comparison through $\approx$.

The next two results prove useful in section 5.4, hence the consideration of Ω_P. More specifically, lemmas 5.3.4 and 5.3.5 relate relations $\equiv$ and $\approx$. The first result proves that $\equiv$ is a subset or equal to relation $\approx_{\Omega_P}$. In other words, we are required to show that equivalence at the structural level is *equally or more discriminating* than pairing filtered configurations (through Ω_P) according to their behaviour. Conversely, we are able to pair more configurations when comparing filtered behaviour rather than when comparing structure. Moreover, filtered configurations which are structurally equivalent must *also* exhibit the same behaviour.

Lemma 5.3.4. $\equiv\ \subseteq\ \approx_{\Omega_P}$

Proof. We define relation $\mathcal{R} \triangleq \{(C_1, C_2) \mid C_1 \equiv C_2\}$. The desired result is achieved by coinduction if we prove $\mathcal{R} \subseteq \approx_{\Omega_P}$.

To prove $\mathcal{R} \subseteq \approx_{\Omega_P}$, we are required to prove statement:

(1) $((C_1\ \mathcal{R}\ C_2) \wedge (C_1 \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$
(2) $((C_1\ \mathcal{R}\ C_2) \wedge (C_2 \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((C_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$

Proof of (1):

We know *(i)* $C_1\ \mathcal{R}\ C_2$, *(ii)* $C_1 \xrightarrow{\alpha}_{\Omega_P} C'$ to be true, and are required to prove *(iii)* $(C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C'')$.

By (i) and *Def$^{\underline{n}}$* of $\mathcal{R}$, $C_1\ \mathcal{R}\ C_2$ implies that $C_1 \equiv C_2$. Given that $C_1 \equiv C_2$ and (ii), by corollary 5.3.1 we infer *(iv)* $C_2 \xrightarrow{\alpha}_{\Omega_P} C''$, and *(v)* $C' \equiv C''$.

By (v) and *Def$^{\underline{n}}$* of $\mathcal{R}$, we infer $C'\ \mathcal{R}\ C''$*...(vi)*. Moreover, knowledge of (v) implies $C_2 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C''$ *...(vii)*.

The result follows by the conjunction of (vi) and (vii).

Proof of (2):

Analogous to the first case.

□

The next lemma is crucial, simplifying subsequent reasoning about $\approx_{\Omega_P}$. More specifically, our aim is to prove that $\approx_{\Omega_P}$ is weak up to structural equivalence. In other words, when checking residual configuration pairs to be in $\approx_{\Omega_P}$ it is sufficient to work up to structurally equivalent terms. We look at $\approx_{\Omega_P}$ in particular due to its use in result 5.2.

Lemma 5.3.5. *$\approx_{\Omega_P}$ is a bisimilarity up to structural equivalence.*

The proof takes a coinductive approach, and can be seen in full in appendix F. We first define relation $\mathcal{R}$ which pairs residual configurations (whose incoming transitions are filtered by Ω_P) up to structurally equivalent terms, and later prove that $\mathcal{R}$ is a bisimulation. From this it follows that $\mathcal{R}$ is a subset of $\approx$, proving that if we pair up to structurally equivalent terms we will remain in $\approx$.

5.4 Results

The following section presents our achievements with respect to the fundamental statements presented in section 5.1. These achievements come in two forms; (i) we have successfully formalised the required statements, and (ii) we also present a proof for the first statement *i.e.,* proving that the monitoring semantics do not effect process computation. Certainly, the first achievement is testament to the creation of a sufficiently rich calculus for our requirements.

Monitoring Does Not Affect Computation

Any given system contains process, monitor and trace components. Given system S we say that its *process projection*, written S_P, extracts/projects the system's processing effort. See appendix C for more details regarding projection, as well as the formal definition of S_P. For now, it suffices to say that by projecting, we syntactically extract the process components within S. This implies that any monitoring behaviour exhibited by S is absent from S_P, since monitors are no longer present after projection.

The definition of process projection gives us a straightforward approach for encoding the required statement. We proceed by comparing the computation of a system and its monitorless equivalent (*i.e.,* its process component). Given system S, if we prove that S and its process projection S_P are behaviourally equivalent (while disregarding monitoring behaviour), then the statement is proven. In other words, if a system and its monitorless version exhibit identical *processing behaviour*, this implies that the system's monitoring effort does not effect process computation. This statement can be written as

$$(\delta \rhd S) \approx_{\Omega_P} (\delta \rhd S_P) \tag{5.2}$$

Proof of the above statement is presented in appendix G and proceeds by coinduction, while inductively exploiting the structure of S (and resultant S_P). We hence define relation $\mathcal{R} = \{S : \textsc{Sys} \mid S' = S_P \bullet (\delta \rhd S, \delta \rhd S')\}$ pairing each system with its process projection, and go on to prove that $\mathcal{R} \subseteq \approx_{\Omega_P}$. However, we are faced with an additional challenge *wrt.* the effect of traces on the above setting. More specifically, traces are generated as a side effect of process computation of both S and S_P. Although we successfully manage to ignore trace behaviour through Ω_P, we are still faced with the problem of handling traces syntactically generated at runtime. In general, their creation during computation imply that residual configuration $S_P{}'$ is not the projection of S' *i.e.,* $(S', S_P{}') \notin \mathcal{R}$. As a consequence, $\mathcal{R}$ is *not* a bisimulation due to its failure to adhere to the transfer property.

As a solution to this issue, we choose to, in some way, ignore traces generated by S and S_P. However, given that we want to prove that process computation is equivalent, and trace entities represent persistent logs of events, traces generated by a system and its projection must be identical. The required statement to prove therefore becomes

$$(\delta \rhd (S \parallel T)) \approx_{\Omega_P} (\delta \rhd (S_P \parallel T))$$

In other words, we are allowed to ignore generated traces, as long as they match on both sides. To this effect, the definition of $\mathcal{R}$ is updated to

- $S' = S_P$ implies $(\delta \rhd S)\ \mathcal{R}\ (\delta \rhd S_P)$
- $(\delta \rhd S)\ \mathcal{R}\ (\delta \rhd S_P)$ implies $(\delta \rhd (S \parallel T))\ \mathcal{R}\ (\delta \rhd (S_P \parallel T))$

In order to prove $\mathcal{R} \subseteq \approx_{\Omega_P}$ we hence have to prove validity of the transfer property for both clauses defining $\mathcal{R}$. We here consider proof of the first clause; see the appendix for the full proof. In order to prove the transfer property for the first clause, we consider each S and prove that if $\delta \rhd S\ \mathcal{R}\ \delta \rhd S_P$, then the following two properties hold:

$$\textit{(1)}\ (\delta \rhd S \xrightarrow{\alpha}_{\Omega_P} C') \Rightarrow ((\delta \rhd S_P \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$$
$$\textit{(2)}\ (\delta \rhd S_P \xrightarrow{\alpha}_{\Omega_P} C') \Rightarrow ((\delta \rhd S \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$$

We next present an example proof of the above statements in case of $S = k[\![c!\bar{x}.P]\!]$, $S_P = k[\![c!\bar{x}.P_P]\!]$.

Proof. **Proof of (1):**

We know *(i)* $(\delta \rhd k[\![c!\bar{x}.P]\!])\ \mathcal{R}\ (\delta \rhd k[\![c!\bar{x}.P_P]\!])$, *(ii)* $\delta \rhd k[\![c!\bar{x}.P]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \rhd k[\![c!\bar{x}.P_P]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$.

Given the structure of configuration $\delta \rhd k[\![c!\bar{x}.P]\!]$, the only valid transition at this stage is

$$\begin{aligned}
&\delta \rhd k[\![c!\bar{x}.P]\!] \xrightarrow{c!v_{\langle p:k\rangle}} \mathsf{inc}(\delta,k) \rhd (k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]) && \text{by rule (O}\textsc{ut}_P\text{)}\\
&\delta \rhd k[\![c!\bar{x}.P]\!] \xrightarrow{c!\bar{v}_{\langle k\rangle}}_{\Omega_P} \mathsf{inc}(\delta,k) \rhd (k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]) && \text{by } Def^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}
\end{aligned}$$

We can derive a matching transition for system $k[\![c!\bar{x}.P_P]\!]$ *i.e.*,

$$\begin{aligned}
&\delta \rhd k[\![c!\bar{x}.P_P]\!] \xrightarrow{c!v_{\langle p:k\rangle}} \mathsf{inc}(\delta,k) \rhd (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]) && \text{by rule (O}\textsc{ut}_P\text{)}\\
&\delta \rhd k[\![c!\bar{x}.P_P]\!] \xrightarrow{c!\bar{v}_{\langle k\rangle}}_{\Omega_P} \mathsf{inc}(\delta,k) \rhd (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]) && \text{by } Def^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}
\end{aligned}$$

which satisfies (iii), since the transitions match, and $(\mathsf{inc}(\delta,k) \rhd (k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]), \mathsf{inc}(\delta,k) \rhd (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!])) \in \mathcal{R}$ since $(k[\![P]\!])_P = k[\![P_P]\!]$, with the counter states and generated traces matching.

Proof of (2):

Analogous to the first case. □

We emphasise the necessity of the second clause defining $\mathcal{R}$ in order to pair residual configurations. Had the second clause not been available, it would have been impossible to prove that the pair $(\mathsf{inc}(\delta,k) \rhd (k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]),\ \mathsf{inc}(\delta,k) \rhd (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!])) \in \mathcal{R}$. An analogous proof is presented for each S. Moreover, we also present two additional cases to show that the result holds when systems are placed in context. Finally, the last case entails verifying the required result for the second clause defining $\mathcal{R}$. Note that the above proof makes use of lemma $(P\sigma)_p = P_p\sigma$, presented in appendix D. This lemma proves that projection is invariant under substitution, and is required during proof of the above statement in case of $S = k[\![c?\bar{x}.P]\!]$.

Global Monitoring Is Equivalent To Local Monitoring

Other than the location of communication, we expect orchestrated monitoring to behave identically to choreographed monitors. In other words, we expect to be able to verify the same property classes using *either* approach, with monitoring location being the only discernible difference. Hence, if we abstract over monitoring locations, resulting system behaviour from both approaches should be identical. To this effect, we make use of filter function Ω_{ML} which filters monitor tags appropriately, and is defined below.

Definition 5.4.1. *(Filter function Ω_{ML}) We define filter function Ω_{ML} as follows*

$$\Omega_{ML}\,\alpha \triangleq \begin{cases} \tau & \textit{if}\,(\alpha = \tau_{\langle m:k,l\rangle}) \vee (\alpha = \tau_{\langle p:k,l\rangle}) \vee (\alpha = \tau_{\langle t:k,l\rangle}) \\ c?\bar{v}_{\langle m\rangle} & \textit{if}\, c?\bar{v}_{\langle m:k\rangle} \\ c!\bar{v}_{\langle m\rangle} & \textit{if}\, c!\bar{v}_{\langle m:k\rangle} \\ \alpha & \textit{otherwise} \end{cases}$$

Hence, well-formed Ω_{ML} abstracts over monitoring location, leaving other behaviour unaffected (apart from filtering τ tags). Given monitorless system S, if we prove that S monitored in orchestrated fashion is equivalent to S orchestrated in choreographed fashion, then the proof is complete. In general, an orchestrated monitoring approach is represented through a monitor of the form $\mathcal{G}\{\![M_{\mathcal{G}}]\!\}^1$, where $M_{\mathcal{G}}$ implements the monitoring logic executed at global location $\mathcal{G}$. Analogously, statically choreographed monitors are implemented through local monitors $k_1\{\![M_{C_1}]\!\}^1 \parallel k_2\{\![M_{C_2}]\!\}^1 \parallel ... \parallel k_n\{\![M_{C_n}]\!\}^1$. Finally, migrating monitors take the form $\mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1$. We arbitrarily choose to start migrating monitors at central monitoring location $\mathcal{G}$. The required statement to prove hence becomes

$$\begin{aligned}(\delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{G}}]\!\}^1) \;&\approx_{\Omega_{ML}}\; (\delta \triangleright S \parallel k_1\{\![M_{C_1}]\!\}^1 \parallel k_2\{\![M_{C_2}]\!\}^1 \parallel ... \parallel k_n\{\![M_{C_n}]\!\}^1) \\ &\approx_{\Omega_{ML}}\; (\delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel k_j\{\![M_{\mathcal{M}_n}]\!\}^1) \qquad (5.3)\end{aligned}$$

Thus implying that all three monitoring approaches are equivalent, ignoring monitoring location. Clearly, the above considerations are based on systems of certain structure. We can ensure required structure by defining an initial static check, in the form of a predicate on systems. Proof of the above statement is left as future work.

Migrating Monitors Preserve Locality

We are interested in proving that remote trace analysis does not happen when using a migrating monitor approach. The pre-LTS semantics allows for the immediate identification of remote trace analysis, by identifying actions of the form $\tau_{\langle t:k,l\rangle}$ *s.t.* $k \neq l$. Hence, our task becomes that of ensuring that migrating monitors never perform said actions. If we consider monitorless system S and a migrating monitor framework $\mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1$, the required statement can be written as

$$\begin{aligned}&\Omega_F\langle \delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1\rangle \;\approx \\ &\Omega_{RT}\langle \delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1\rangle \qquad (5.4)\end{aligned}$$

Whereas $\Omega_F\langle \delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1\rangle$ considers (monitored) system behaviour in full, $\Omega_{RT}\langle \delta \triangleright S \parallel \mathcal{G}\{\![M_{\mathcal{M}_1}]\!\}^1 \parallel \mathcal{G}\{\![M_{\mathcal{M}_2}]\!\}^1 \parallel ... \parallel \mathcal{G}\{\![M_{\mathcal{M}_j}]\!\}^1\rangle$ considers the same system's behaviour while prohibiting remote trace analysis. If both LTSs are behaviourally equivalent, then we would have proven that no remote tracing could have been performed by the migrating monitors. Statement 5.4 is an example consideration of system behaviour at different levels of abstraction.

5.5 Conclusions

Throughout this chapter we presented mDPI, a distributed π-calculus with explicit monitoring capabilities. The framework does not focus on one particular monitoring approach, but rather encodes a generalised framework allowing for the formalisation (and comparison) of *various* techniques. We defined an *extensible* LTS semantics allowing for the definition of behaviour at different abstraction levels, considered structural equivalence and also adopted the bisimilarity relation as a measure of behavioural equivalence. Here we saw an application of the pre-LTS semantics, allowing us to focus on different aspects of a system's computation as necessary. With the necessary mathematics at hand, we later produced results regarding our framework's semantics. Importantly, we have proven that our monitoring semantics do not effect process computation, thus providing added assurances of our monitoring framework. It is worth noting that our approach also gives us an elegant approach for categorising LTS representations, which we showed to have desirable repercussions. More specifically, this hierarchy was exploited for proof reuse. Moreover, it was also proven that by comparing more informative LTSs, said comparisons hold when dealing with their less informative equivalent.

To the best of our knowledge, the formalisation of a general monitoring semantics through a π-calculus adaptation is novel. However, the above framework should not be taken as a final study to distributed monitoring — we recognise the need to explore the framework further. For instance, although our monitors obtain temporal orderings through a mechanism analogous to Lamport Timestamps [58], other approaches have been shown to be more fruitful in certain scenarios (including Vector Clocks [35]). We believe our framework to be sufficiently extensible to encode alternate approaches, for instance by enriching the monitor counter by adopting more powerful data structures. Other future work includes the generation of more results (especially proof of the remaining two statements), and also an investigation of our monitoring framework when faced with mobile systems. We shall return to these issue when discussing future work (see section 9.2.1).

6. Distributed Regular Expression Monitoring

The following chapter illustrates the instrumentation of different monitoring approaches in mDPɪ, expressed through the conversion of regular expressions to mDPɪ monitors. In other words, temporal properties on remote process events are *specified* through some regular expression, from which we *synthesise* a monitor capable of verifying said properties. An overview of the proposed approach is presented in section 6.1. Moreover, we shall provide *multiple* conversions in order to illustrate different *monitoring strategies*; converting regular expressions to orchestrated (section 6.2), statically choreographed (section 6.3) and migrating monitors (section 6.4). Finally, section 6.5 concludes the chapter.

6.1 Overview

Regular expressions allow for the *intuitive* specification of temporal orderings on events [74]. We shall therefore adopt their use as a *simple logic* allowing for the illustration of different monitoring strategies (presented in chapter 3) in the mDPɪ calculus. Given system S, whose behaviour we are interested in verifying against a set of properties $\varphi_i \in \textsc{Prop}$, our proposed framework proceeds as follows

(i) Each property φ_i is first encoded through regular expression E_i;

(ii) An mDPɪ monitor M_i is next adopted as an implementation for E_i;

(iii) Monitor M_i is responsible for analysing trace T generated by S;

(iv) The *satisfaction* of E_i by T is considered a *violation* of property φ_i.

The above setting hence employs *numerous* monitors for the analysis of the runtime behaviour describing S, increasing modularity. Moreover, each M_i might refer to a monitoring configuration which employs additional (possibly distributed) monitors, depending on the adopted monitoring approach (more below). The design choice of admitting persistent traces hence comes to fruition, allowing for multiple monitors to analyse the same trace generated by S against various expressions E_i. Crucially, note that our interpretation of expression satisfaction as a property violation implies that we are *monitoring for failure*. Also, given that our synchronous monitors operate on finite traces (*i.e.,* traces seen so far), this implies that regular expressions specify properties of *safety*. The following regular expression syntax shall be assumed.

$$E \quad ::= \quad (e, \bar{v})\,@\,k \mid E.\,E \mid E^{*} \mid E + E$$

Figure 6.1: Regular Expression Syntax

Expression $(e, \bar{v})\,@\,k$ serves as an *atomic proposition* in our language, denoting the occurrence of event e with parameters $\bar{v}$ at location k. The addition of $\bar{v}$ gives us a basic form of conditions over event parameters. Concatenation $E.\,E$ denotes sequential order. Union $E + E$ specifies that either expression must be satisfied in order for the compound expression to trigger. Finally, for E^{*} to trigger we require E to be repeatedly satisfied for an arbitrary number of times. We shall use notation $(e, \langle\rangle)\,@\,k$ when we do not wish to impose a condition on the event parameters.

Notation 6.1. *We make use of notation* $E \triangleq \Sigma k : \mathrm{Locs}.\, E'(k)$ *to specify that, in order for* E *to trigger, intermediate expression* E' *must trigger at one or more locations* $k \in \mathrm{Locs}$ *i.e., as shorthand for* $E = E'(k_1) + E'(k_2) + ... + E'(k_n)$ *(assuming set of locations* $\mathrm{Locs} = \{k_1,\ k_2,\ ...,k_n\}$*).*

Even through the above language is rather basic, we can still encode certain interesting properties. To this effect, consider the hospital management system scenario depicted in Fig. 3.2. Clearly, the system handles confidential information relating to the patients' medical records. This implies that the system is to adhere to a set of mission-critical requirements, both to *ensure* its correct operation in order to avoid the misplacement of medical records, as well as to *avoid* leaking confidential information to unauthorised entities. We assume the following events in order to specify the required properties; (i) *req() @ p*, denoting request of the personal medical record by patient p, (ii) *resp(p,b) @ d*, indicating boolean response b by doctor d for the request placed by patient p, (iii) *sendRec(*p_1*,*$\{p_2$*,*$r\}$*) @ be* referring to *release* of record $\{p_2, r\}$ (entailing patient name p_2 and her diagnosis r) to patient p_1 at backend location *be*. Keeping these events in mind, we express the following *three* properties (introduced in section 3.3) which the system is to adhere to.

1. *No patient is to be given another patient's record as a response* — Conversely, this statement can be restated as a failure property of the form "If a patient request is given another patient's record as a response, then the property is violated". Assuming a set of patients Pat, this can be encoded as follows

$$\Sigma\, p : \textsc{Pat} \bullet\ (req, \langle\rangle)\,@\,p.\,(sendRec, \langle p, \{p', r\}\rangle)\,@\,be$$

This first statement is an example temporal property across remote locations. The property extends across all patient locations, such that if just one patient makes a request and receives record $\{p', r\}$ *s.t.* $p \neq p'$ as a response, then the regular expression is satisfied (and the property, violated).

2. *Multiple patient requests should be responded by at most one medical record release* — For security, the hospital management wishes to minimise exposure of medical records. To this effect, they adopt the policy of granting patients at most one chance to release their records through the system. This statement is formalised as

$$\Sigma\, p : \textsc{Pat} \bullet\ ((req, \langle\rangle) \mathbin{@} p)^{*}.\,(sendRec, p, \{p, r\}) \mathbin{@} be.\,(sendRec, p, \{p, r\}) \mathbin{@} be$$

If an arbitrary amount of patient requests are given as response at least two record releases, then the property is violated.

3. *The release of a patient's record must be approved by supervising doctors* — This property can be restated as "If a patient's medical record is released regardless of a doctor's disapproval, this implies that the property is violated". Assuming set Doc of registered doctors, the statement is written as

$$\Sigma\, p : \textsc{Pat} \bullet\ (req, \langle\rangle) \mathbin{@} p.\,(\Sigma\, d : \mathrm{Doc} \bullet\ (resp, \langle p, \mathit{false}\rangle) \mathbin{@} d.\,(sendRec, \langle p, \{p, r\}\rangle) \mathbin{@} be)$$

This property involves three locations; if a patient request is rejected by one or more doctors, and the patient's record is released nonetheless by the backend, then the property is violated.

We shall consider these three example properties throughout the remainder of this chapter, as well as during chapter 7 during our discussion of the case study.

As highlighted by point (ii) above, properties φ_i expressed as regular expressions E_i are to be implemented through mDPi monitors M_i. In other words, we exploit the monitors' operational semantics (section 5.2.2) in order to *execute* necessary verification on traces. We specify monitor implementations through *conversion strategy* ψ

$$\psi :: (E \times \textsc{Chans} \times \textsc{Chans}) \ \rightarrow\ \textsc{Sys}$$

which takes (i) a regular expression, (ii) a *start* and (iii) *finish* channel, and returns a (set of) *located monitor(s)*. Synchronisation on the start channel instructs the monitor/s (in M_i) to start its/their verification process. On the other hand, output on the finish channel signifies that a sequence of analysed trace entities is included in the language expressed by E_i. Through ψ, we can encode the verification of S for various expressions E_i, $1 \leq i \leq n$.

Definition 6.1.1. *(Monitoring of Expressions) Given conversion strategy ψ, the monitoring of system S wrt. expressions E_i, $1 \leq i \leq n$, assuming corresponding start channels c_i, $1 \leq i \leq n$ becomes*

$$S \parallel \mathsf{new}\, c_o.(\psi(E_1, c_1, c_o) \parallel ... \parallel \psi(E_n, c_n, c_o) \parallel G\{\![c_o?\langle\rangle.\mathsf{fail}]\!\}^1)$$

Each expression E_i is converted to its corresponding located monitor/s through ψ. Moreover, we make use of an additional monitor $c_o?\langle\rangle.\mathsf{fail}$ arbitrarily placed at location $\mathcal{G}$, whose purpose is that of reporting failure if *any* of the resulting monitors identify a trace satisfying E_i. Note the use of channel c_o, whose scope is shared exclusively amongst monitors, and is used to report failure. Also, the start of each monitor's execution requires synchronisation on each start channel c_i.

The *arrangement* of the monitors in M_i dictate the monitoring approach taken by the resulting monitors. Given that chapter 3 introduced numerous broad approaches to the monitoring of distributed systems, we shall hence provide multiple *conversion strategies*. More specifically, three conversions ψ_G, ψ_C and ψ_M shall be provided. The former converts regular expressions to an *orchestrated monitor implementation*. Moreover, given that mDPi describes a property agnostic approach (section 3.4.4) by differentiating between its monitoring and tracing semantics, ψ_G can be applied for both *static* and *dynamic* orchestration. In other words, ψ_G describes necessary verification through a central monitor. We can either choose to start this monitor in conjunction with the system (*i.e.*, static orchestration), or we may spawn the resulting monitor at runtime (dynamic orchestration). On the other hand ψ_C implements a *static choreographed based approach* by generating a set of distributed monitors which analyse traces locally. Finally, ψ_M monitors the required regular expression through a *migrating monitor approach*. Note that although ψ_C and ψ_M both implement choreographed approaches, we are forced to distinguish between their implementations due to the elevated *encapsulation* achieved through migrating monitors, as well as the additional use of the go operator by the latter.

Although we do not encode practical considerations in mDPi (such as memory and time consumption), we still require an element of *efficiency* from the aforementioned conversions. More specifically, we expect the conversion process to create monitors whose size is linear *wrt.* the property specification. It is for this reason that we do not consider conjunction and negation in Fig. 6.1, due to implied exponential blowup.

Sokolsky *et al.* [74] recognise a set of design issues pertinent to the application of regular expressions to a runtime verification setting. We next consider these issues with respect to our work.

- **Do we automatically include events in a regular expression into a relevant set?** This issue deals with the identification of pertinent events *wrt.* the expression under consideration. We assign an *implicit interpretation* to our regular expression events (as opposed to *explicit* event declarations). In order to expose the difference between approaches, consider expression $E = a.b$. An explicit interpretation of E would not be satisfied by sequence *'a, c, b'*, since the expression makes no explicit reference to 'c'. On the other hand, an implicit interpretation ignores non-pertinent events, as long as the *sequence projection* of pertinent events satisfies E. The above sequence would hence satisfy E through an implicit interpretation. This interpretation of implicit events happens to match the monitoring semantics, where uninteresting trace entities are ignored by the monitor

- **If there are more than one place to start evaluating a regular expression, when should we start? Earliest or latest?** — Given that the start of monitor execution is manually specified through synchronisation on the input channel, this choice is left to the user. However, synchronous mDPɪ monitors are *typically* expected to start as soon as possible in order to keep up with the processes' execution. More specifically, we require our monitors to match the rate of trace generation in order to extract temporal orderings on remote trace entities.

- **If there are more than one place to end evaluating a regular expression, when should we end? Earliest (shortest sequence), latest (longest sequence), or report all?** The semantics assigned to runtime monitoring usually assumes the identification of the *shortest satisfying trace*, for three reasons. Firstly, waiting for the longest trace may be problematic, due to possibly infinite traces. Moreover, we are typically interested in the *first* property violation; reporting each subsequent violation for the same property may lead to redundancy [74]. This implies that for instance regular expression $a^*.a$ is satisfied by sequence *'a'*. In other words, a^* is satisfied immediately, with 'a' satisfying expression a (*i.e.,* it is not a continuation of a^*). Finally, opting for the identification of the shortest prefix is crucial for the design of a monitoring algorithm which adheres to the principle of *anticipation* (section 2.3).

- **How do we deal with overlapping sequences?** Given expression $a.\,b.\,a$, and sequence *'a,b,a'*, the second occurrence of 'a' is taken to satisfy the first expression *i.e.,* it does spawn an additional monitor which starts verifying the same expression.

The following sections present the conversion of the regular expression syntax in Fig. 6.1 to orchestrated, statically choreographed and migrating monitor implementations. Assuming synchronisation channels c_1, c_2, c_3, throughout these sections we shall often refer to two auxiliary components;

(i) $or(c_1, c_2, c_3)$, such that a monitor listening on c_3 will synchronise if output on either c_1 or c_2 is detected *i.e.,* as shorthand for

$$or(c_1,c_2,c_3) \triangleq *(c_1?\langle\rangle.c_3!\langle\rangle.\mathsf{stop}) \parallel *(c_1?\langle\rangle.c_2!\langle\rangle.\mathsf{stop})$$

(ii) $repl(c_1, c_2, c_3)$, which repeatedly replicates synchronisation on c_1 to channels c_2 and c_3

i.e., as shorthand for

$$repl(c_1, c_2, c_3) \triangleq *(c_1?\langle\rangle.(c_2!\langle\rangle.\mathsf{stop} \parallel c_3!\langle\rangle.\mathsf{stop}))$$

Their use will become apparent during the definition of the expression conversions. Finally, note that definitions for ψ throughout this chapter are based on the work presented in [22].

6.2 Regular Expressions To Orchestrated Monitors

The following section presents the conversion of regular expressions to an orchestrated monitoring approach *i.e.*, we present a definition for ψ_G. To this effect, we shall adopt a *central monitor* (placed at *global location* $\mathcal{G}$), performing remote trace analysis accordingly. We first consider the matching of basic proposition $(e, \bar{v})$ @ k; verifying this statement entails analysing a trace entity recording output of $\bar{v}$ on channel e, *after* synchronisation on start channel s.

$$\psi_G((e, \bar{v}) \mathbin{@} k, s, f) \triangleq \mathcal{G}[\![s?\langle\rangle.\mathsf{setC}(k).\ *(\mathbf{m}(e, \bar{x}, k).\mathsf{if}\ \bar{x} = \bar{v}\ \mathsf{then}\ (f!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$$

Note the use of operator SetC, forcing the monitor to ignore traces generated prior to receiving the go ahead on start channel s. This addition effectively enables the extraction of temporal orderings on remote events. The remainder of the monitor implementation is straightforward, repeatedly analysing trace entities for one which matches the expression's needs (*i.e.*, matching both on e and $\bar{v}$). When one such trace entity is found, a signal on f is transmitted.

In order for expression $E_1.\,E_2$ to be satisfied E_1 must be matched first, and once matched we attempt to match E_2. This internal signal is implemented through internal (scoped) channel c (see Fig. 6.2).

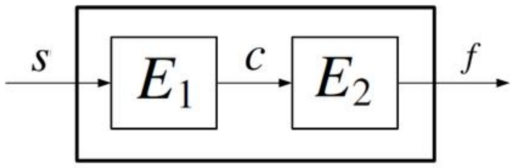


Figure 6.2: Conversion of $E_1.\,E_2$

$$\psi_G(E_1.\,E_2, s, f) \triangleq \mathsf{new}\, c.(\psi_G(E_1, s, c) \parallel (\psi_G(E_1, c, f))$$

Matching E^* is analogous to the matching of E, however with the addition of restarting verification on *each* matching of E. Moreover, given that E^* can be satisfied by zero iterations of E, this implies that the finish channel is triggered upon synchronisation on channel s (see Fig. 6.3). Note that internal components *or* and *repl* are also placed at $\mathcal{G}$.

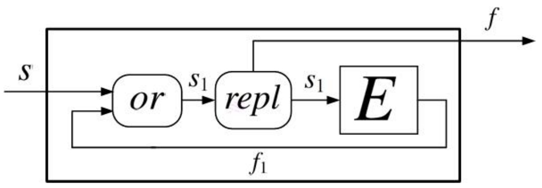


Figure 6.3: Conversion of E^*

$$\psi_G(E^*, s, f) \overset{\wedge}{=} \mathsf{new}\, s_1, f_1.(\mathcal{G}\{\![or(s, f_1, s_1) \parallel repl(s_1, f, s_1)]\!\}^1 \parallel \psi_G(E, s_1, f_1))$$

Expression $E_1 + E_2$ is matched if either E_1 or E_2 are satisfied. The start channel is first replicated in order to start concurrent verification for both expressions. If either expression subsequently triggers on intermediate channels f_1 or f_2, this implies that the compound expression is also satisfied, thus signalling on f (Fig. 6.4).

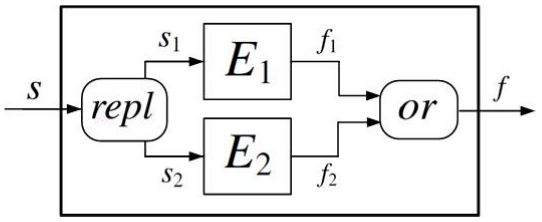


Figure 6.4: Conversion of $E_1 + E_2$

$$\begin{aligned}\psi_G(E_1 + E_2, s, f) \quad \overset{\wedge}{=} \quad & \mathsf{new}\, s_1, s_2, f_1, f_2.(\ \mathcal{G}\{\![repl(s, s_1, s_2) \parallel or(f_1, f_2, f)]\!\}^1 \parallel \\ & \qquad \psi_G(E_1, s_1, f_1) \parallel \psi_G(E_2, s_2, f_2) \quad)\end{aligned}$$

We next present an example conversion, applied to the first property introduced in section 6.1.

An Example

Consider the first example property, specifying that no patient request is to be given another patient's record as a response. We formalised this requirement as

$$\Sigma\, p : \textsc{Pat} \bullet\ (req, \langle\rangle) \mathbin{@} p.\,(sendRec, \langle p, \{p', r\}\rangle) \mathbin{@} be$$

by quantifying over an unspecified set of patients $\textsc{Pat}$. For simplicity, we shall consider the hospital management system to admit two patients *i.e.*, $\textsc{Pat} \overset{\wedge}{=} \{p_1, p_2\}$. The property hence becomes

$$((req, \langle\rangle) \mathbin{@} p_1.\,(sendRec, \langle p_1, \{p', r\}\rangle) \mathbin{@} be) + ((req, \langle\rangle) \mathbin{@} p_2.\,(sendRec, \langle p_2, \{p', r\}\rangle) \mathbin{@} be)$$

We shall convert this property in stepwise fashion. Assuming start and finish channels s, f, the central monitor takes the following form

$$\mathsf{new}\, s_1, s_2, f_1, f_2.(\mathcal{G}[\![repl(s, s_1, s_2) \parallel or(f_1, f_2, f)]\!]^1 \parallel A \parallel B)$$

where A, B represent the monitor implementations for expressions

$$(req, \langle\rangle) \mathbin{@} p_1 . (sendRec, \langle p_1, \{p', r\}\rangle) \mathbin{@} be \ , \quad (req, \langle\rangle) \mathbin{@} p_2 . (sendRec, \langle p_2, \{p', r\}\rangle) \mathbin{@} be$$

respectively. Let us consider the definition for A (B is analogous). The top level operator involves concatenation, implying that $A = \mathsf{new}\, c.(A' \parallel A'')$. A' and A'' involve the verification of basic propositions $(req, \langle\rangle) \mathbin{@} p_1$ and $(sendRec, \langle p_1, \{p', r\}\rangle) \mathbin{@} be$. Hence, A' refers to monitor

$$\mathcal{G}[\![s_1?\langle\rangle.\mathsf{setC}(p_1). *(\mathbf{m}(req, \bar{x}, p_1).\mathsf{if}\ (\bar{x} = \langle\rangle)\ \mathsf{then}\ (c!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$$

whereas A'' refers to

$$\mathcal{G}[\![c?\langle\rangle.\mathsf{setC}(be). *(\mathbf{m}(sendRec, \bar{x}, be).\mathsf{if}\ (\bar{x} = \langle p_1, \{p', r\}\rangle)\ \mathsf{then}\ (f!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$$

where p' refers to some patient location other than p_1. The resulting monitor implementation hence becomes

```
$\mathsf{new}\, s_1, s_2, f_1, f_2.($
    $\mathcal{G}[\![repl(s, s_1, s_2) \parallel or(f_1, f_2, f)]\!]^1 \parallel$
    $\mathsf{new}\, c.($
      $\mathcal{G}[\![s_1?\langle\rangle.\mathsf{setC}(p_1). *(\mathbf{m}(req, \bar{x}, p_1).\mathsf{if}\ (\bar{x} = \langle\rangle)\ \mathsf{then}\ (c!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1 \parallel$
      $\mathcal{G}[\![c?\langle\rangle.\mathsf{setC}(be). *(\mathbf{m}(sendRec, \bar{x}, be).\mathsf{if}\ (\bar{x} = \langle p_1, \{p', r\}\rangle)\ \mathsf{then}\ (f_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$
    $) \parallel$
    $\mathsf{new}\, c.($
      $\mathcal{G}[\![s_1?\langle\rangle.\mathsf{setC}(p_2). *(\mathbf{m}(req, \bar{x}, p_2).\mathsf{if}\ (\bar{x} = \langle\rangle)\ \mathsf{then}\ (c!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1 \parallel$
      $\mathcal{G}[\![c?\langle\rangle.\mathsf{setC}(be). *(\mathbf{m}(sendRec, \bar{x}, be).\mathsf{if}\ (\bar{x} = \langle p_2, \{p', r\}\rangle)\ \mathsf{then}\ (f_2!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$
    $)$
$)$
```

All the resulting components are placed at $\mathcal{G}$, with the central monitoring configuration effecting remote trace analysis accordingly. Hence, although correct, using the above monitors results in information exposure, particularly during the use of operator **m**(...). Admittedly, the resulting configuration is not optimal — one could hand-code simpler monitors which achieve the same result. For instance, an optimised version of the current implementation for component A can be achieved by removing the use of intermediate channel c.

```
$\mathcal{G}[\![ s_1?\langle\rangle.\mathsf{setC}(p_1).*(\mathbf{m}(req,\bar{x},p_1).$
    $\mathsf{if}\,(\bar{x}=\langle\rangle)\,\mathsf{then}$
        $(\mathsf{setC}(be).*(\mathbf{m}(sendRec,\bar{x},be).\mathsf{if}\ (\bar{x}=\langle p_1,\{p',r\}\rangle)\ \mathsf{then}\ (f_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop})))$
    $\mathsf{else\,stop}$
$]\!]^1$
```

Analogously, one can imagine scenarios where the use of setC is made redundant in case where the monitor does not require re-alignment of its counter to remote locations (*i.e.*, the monitor is interested in sequentially analysing events at *one* location). However, these conversions should be rather considered as proofs-of-concept; exposing that mDPi monitors are sufficiently expressive to monitor regular expressions through different approaches. For better performance, one could for instance apply monitor re-writing heuristics to eliminate said redundancy, and is left as future work. Nevertheless, it should be noted that although not optimal, the above conversion still generate monitors which are *linear* in the size of the expression.

6.3 Regular Expressions To Statically Choreographed Monitors

We next consider a definition for ψ_C, implementing regular expressions through statically choreographed monitors. As we shall see, the definition of ψ_C is analogous to ψ_G, excluding monitoring location. More specifically, although we shall adopt the same compilation strategies depicted in Fig. 6.2, Fig. 6.3 and Fig. 6.4, we strive to *localise* monitoring components in order to avoid information exposure. For instance, monitoring $(e,\bar{v})\ @\ k$ locally involves applying the same monitor used in case of orchestration, however placed locally to the event's occurrence *i.e.*, at k.

$$\psi_C((e,\bar{v})\ @\ k,s,f) \triangleq k[\![s?\langle\rangle.\mathsf{setC}(k).*(\mathbf{m}(e,\bar{x},k).\mathsf{if}\ \bar{x}=\bar{v}\ \mathsf{then}\ (f!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$$

The result defines the same monitoring effort as its orchestrated equivalent, while avoiding exposure of trace information. Operator setC is once more used to force extraction of a temporal ordering on traces, by directing the monitor to ignore the trace subsequence generated prior to synchronisation on s. An output on f is triggered once the required trace entity is analysed.

Converting concatenation $E_1.E_2$ is identical to its orchestrated counterpart (see Fig. 6.2). However, note that the monitoring of E_1 and E_2 results in a set of localised monitors, unlike their orchestrated counterparts placed at $\mathcal{G}$. This implies that monitor synchronisation over c may possibly involve *remote* interactions. Nevertheless, this interaction does not constitute information exposure, since no trace information is sent over c.

$$\psi_C(E_1.E_2,s,f) \triangleq \mathsf{new}\,c.(\psi_C(E_1,s,c)\ \|\ (\psi_C(E_1,c,f))$$

Compiling E^* takes an analogous approach to Fig. 6.3. However, compiling E through ψ_C results in a set of localised monitors. We are also faced with a choice *wrt.* the placement of auxiliary components *or* and *repl*; given that we are distributing monitoring functionality it is not clear where they should be placed. For now, we (rather arbitrarily) choose to place additional components at location $\mathcal{G}$. Since these components are used solely for synchronisation purposes, this implies that placing the components remotely does not imply exposure. Nevertheless, we can eventually optimise this definition to choose a more favourable location *wrt.* the monitors verifying E. Given that E may in itself admit a compound expression (involving the use of multiple monitors), it is in general not immediately obvious where this optimal location lies. However, the study of such optimisations is potentially advantageous in order to *minimise* the implied *communication overhead* imposed by monitors.

$$\psi_C(E^*, s, f) \triangleq \mathsf{new}\, s_1, f_1.(\mathcal{G}\{\![or(s, f_1, s_1) \parallel repl(s_1, f, s_1)]\!\}^1 \parallel \psi_C(E, s_1, f_1))$$

The compilation of $E_1 + E_2$ adopts an identical strategy to Fig. 6.4. Note that intermediate components are once more placed at monitoring location $\mathcal{G}$. Analogously, we may apply similar optimisations for choosing optimal locations, and is left as future work.

$$\begin{aligned}\psi_C(E_1 + E_2, s, f) \;\triangleq\;\; & \mathsf{new}\, s_1, s_2, f_1, f_2.(\;\mathcal{G}\{\![repl(s, s_1, s_2) \parallel or(f_1, f_2, f)]\!\}^1 \parallel \\ & \qquad\qquad\qquad \psi_C(E_1, s_1, f_1) \parallel \psi_C(E_2, s_2, f_2)\quad)\end{aligned}$$

An Example

We next consider the second property specifying that multiple patient requests should be responded by at most one medical record release, written

$$\Sigma\, p : \textsc{Pat} \bullet\; ((req, \langle\rangle)\ @\ p)^*.\,(sendRec, p, \{p, r\})\ @\ be.\,(sendRec, p, \{p, r\})\ @\ be$$

We now consider our setting to admit one patient p_1, such that the property is simplified to

$$((req, \langle\rangle)\ @\ p_1)^*.\,(sendRec, p_1, \{p_1, r\})\ @\ be.\,(sendRec, p_1, \{p_1, r\})\ @\ be$$

The statement at hand entails the concatenation of three expressions, resulting in a monitor of the form

$$\mathsf{new}\, c_1.(A' \parallel \mathsf{new}\, c_2.(A'' \parallel A'''))$$

where A', A'' and A''' represent monitors responsible for verifying

$$((req, \langle\rangle)\ @\ p_1)^*,\ (sendRec, p_1, \{p_1, r\})\ @\ be \text{ and } (sendRec, p_1, \{p_1, r\})\ @\ be$$

These three monitors take the following implementations, according to ψ_C

$$\begin{aligned}A' &= \mathsf{new}\, s_1, f_1.(\;\mathcal{G}\{\![or(s, f_1, s_1) \parallel repl(s_1, c_1, s_1)]\!\}^1 \parallel \\ &\qquad p_1\{\![s_1?\langle\rangle.\mathsf{setC}(p_1).\ *(\mathbf{m}(req, \bar{x}, p_1).\mathsf{if}\ \bar{x} = \langle\rangle\ \mathsf{then}\ (f_1!\langle\rangle.\mathsf{stop})\ \mathsf{else\ stop})]\!\}^1\;)\\ A'' &= be\{\![c_1?\langle\rangle.\mathsf{setC}(be).\ *(\mathbf{m}(sendRec, \bar{x}, be).\mathsf{if}\ \bar{x} = \langle p_1, \{p_1, r\}\rangle\ \mathsf{then}\ (c_2!\langle\rangle.\mathsf{stop})\ \mathsf{else\ stop})]\!\}^1\\ A''' &= be\{\![c_2?\langle\rangle.\mathsf{setC}(be).\ *(\mathbf{m}(sendRec, \bar{x}, be).\mathsf{if}\ \bar{x} = \langle p_1, \{p_1, r\}\rangle\ \mathsf{then}\ (f!\langle\rangle.\mathsf{stop})\ \mathsf{else\ stop})]\!\}^1\end{aligned}$$

The monitor resulting from the compilation of the original expression hence takes the form

```
new $c_1$.(
  new $s_1, f_1$.(
    $\mathcal{G}[\![or(s, f_1, s_1) \parallel repl(s_1, c_1, s_1)]\!]^1$ ||
    $p_1[\![s_1?\langle\rangle$.setC($p_1$). $*$(m($req, \bar{x}, p_1$).if $\bar{x} = \langle\rangle$ then ($f_1!\langle\rangle$.stop) else stop)$]\!]^1$ )
  ) ||
  new $c_2$.(
      $be[\![c_1?\langle\rangle$.setC($be$). $*$(m($sendRec, \bar{x}, be$).
                      if $\bar{x} = \langle p_1, \{p_1, r\}\rangle$ then ($c_2!\langle\rangle$.stop) else stop)$]\!]^1$ ||
      $be[\![c_2?\langle\rangle$.setC($be$). $*$(m($sendRec, \bar{x}, be$).
                      if $\bar{x} = \langle p_1, \{p_1, r\}\rangle$ then ($f!\langle\rangle$.stop) else stop)$]\!]^1$
  )
)
```

The resulting monitor successfully encodes a statically choreographed approach, by distributing the monitoring effort. As one can see, certain monitors are placed at location p_1, whereas others are placed at the backend location *be*. Moreover, these monitors are always instructed to analyse traces locally; operator **m**(...) is always directed to read trace entities generated at its current location. However, one underlying assumption with the above conversion is that all locations are available from startup, and remain unchanged during system execution. The next section considers an example where locations are added at runtime, and the implied monitoring approach necessary to handle such scenarios.

6.4 Regular Expressions To Migrating Monitors

The following section presents a definition for ψ_M, compiling regular expressions to mDPi monitors adhering to a migrating monitor approach. The following definition of ψ_M is an adaptation of ψ_G; resulting monitors start their verification process at $\mathcal{G}$, before migrating to pertinent locations as necessary during system computation. In other words, ψ_G and ψ_M diverge on their *monitor re-alignment strategy* — whereas the former makes use of setC to extract temporal orderings on remote events, the latter uses operator go, exploiting migration's natural sequential semantics. However, by physically migrating to the event's location we avoid the exposure of information during trace analysis.

Given expression $(e, \bar{v}) @ k$, this implies that we require a monitor which verifies event e taking place at k. Hence, upon receiving the go ahead on s we instruct the monitor to *migrate* to k before effecting its verification. Note that the use of go also implies monitor re-alignment, by readjusting the monitor's counter value to the next value at k (see Fig. 5.15; rule Go_m). Through this counter update, we force the monitor to only consider trace entities generated *after* migration. Once local to the event, the monitor can analyse traces as necessary, however avoiding the exposure of traces. Upon analysing the necessary trace entity (with the required parameters $\bar{v}$), a signal on finish channel f is triggered.

$$\psi_M((e, \bar{v}) \mathbin{@} k, s, f) \triangleq \mathcal{G}[\![s?\langle\rangle.\mathsf{go}\ k. *(\mathbf{m}(e, \bar{x}, k).\mathsf{if}\ \bar{x} = \bar{v}\ \mathsf{then}\ (f!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1$$

The conversion strategy for concatenation, kleene star and union is identical to Fig. 6.2, Fig. 6.3 and Fig. 6.4 respectively, defined below.

$$\begin{aligned}
\psi_M(E_1 . E_2, s, f) &\triangleq \mathsf{new}\, c.(\psi_M(E_1, s, c) \parallel (\psi_M(E_1, c, f)) \\
\psi_M(E^*, s, f) &\triangleq \mathsf{new}\, s_1, f_1.(\mathcal{G}[\![or(s, f_1, s_1) \parallel repl(s_1, f, s_1)]\!]^1 \parallel \psi_M(E, s_1, f_1)) \\
\psi_M(E_1 + E_2, s, f) &\triangleq \mathsf{new}\, s_1, s_2, f_1, f_2.(\ \mathcal{G}[\![repl(s, s_1, s_2) \parallel or(f_1, f_2, f)]\!]^1 \parallel \\
&\qquad\qquad \psi_M(E_1, s_1, f_1) \parallel \psi_M(E_2, s_2, f_2)\quad)
\end{aligned}$$

Once more, auxiliary components *or* and *repl* are placed at $\mathcal{G}$. In general, the above conversion describes monitors which start computation at monitoring location $\mathcal{G}$, subsequently migrate as necessary to various locations (during system computation), and interact with the centrally located auxiliary components to synchronise the global monitoring effort.

An Example

Consider the third property presented in section 6.1, stating that the release of a patient's record must be approved by supervising doctors. This property was defined as

$$\Sigma\, p : \textsc{Pat} \bullet\ (req, \langle\rangle) \mathbin{@} p. (\Sigma\, d : \mathrm{Doc} \bullet\ (resp, \langle p, false\rangle) \mathbin{@} d. (sendRec, \langle p, \{p, r\}\rangle) \mathbin{@} be)$$

For simplicity, we shall now consider $\textsc{Pat} \triangleq \{p_1\}$, $\mathrm{Doc} \triangleq \{d_1\}$ *i.e.,* a hospital management system admitting one patient and one doctor. However, the system now admits a dynamic configuration, by admitting the eventual login of a second doctor d_2 during the system's execution. Encoding the above property on this setting using ψ_C is impossible, since we cannot place monitors local to d_2 at startup (given that the doctor has not logged in yet). However, using ψ_M we can generate the necessary monitor to start at $\mathcal{G}$, such that it eventually migrates as necessary to d_2 during system execution, once d_2 has logged in. The property is hence expanded to

$$\begin{aligned}
(req, \langle\rangle) \mathbin{@} p_1 . (\ &(resp, \langle p_1, false\rangle) \mathbin{@} d_1 . (sendRec, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} be) \\
&+ (resp, \langle p_1, false\rangle) \mathbin{@} d_2 . (sendRec, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} be)\ )
\end{aligned}$$

Assuming start and finish channels s and f, the resulting monitor hence takes the form $\mathsf{new}\, c.(\mathcal{G}[\![s?\langle\rangle.\mathsf{go}\ p_1 . *(\mathbf{m}(req, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle\rangle\ \mathsf{then}\ (c!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!]^1 \parallel B)$, where B represents the monitor responsible for the verification of expression

$$\begin{aligned}(\mathit{resp}, \langle p_1,\mathit{false}\rangle) \mathbin{@} d_1 . (\mathit{sendRec}, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} \mathit{be}) \\ + (\mathit{resp}, \langle p_1,\mathit{false}\rangle) \mathbin{@} d_2 . (\mathit{sendRec}, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} \mathit{be})\end{aligned}$$

The monitor for B hence takes the form $\mathsf{new}\, s_1, s_2, f_1, f_2.(\mathcal{G}\{\![\mathit{repl}(c, s_1, s_2) \parallel \mathit{or}(f_1, f_2, f)]\!\}^1 \parallel B' \parallel B'')$, where B' represents the monitor tasked with verifying

$$(\mathit{resp}, \langle p_1,\mathit{false}\rangle) \mathbin{@} d_1 . (\mathit{sendRec}, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} \mathit{be})$$

Analogously, B'' verifies $(\mathit{resp}, \langle p_1,\mathit{false}\rangle) \mathbin{@} d_2 . (\mathit{sendRec}, \langle p_1, \{p_1, r\}\rangle) \mathbin{@} \mathit{be})$. The implementation for B' according to ψ_M is

$$\begin{aligned}&\mathsf{new}\, c_1.(\mathcal{G}\{\![s_1?\langle\rangle.\mathsf{go}\ d_1 . *(\mathbf{m}(\mathit{resp}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1,\mathit{false}\rangle\ \mathsf{then}\ (c_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1 \parallel \\ &\quad + \mathcal{G}\{\![c_1?\langle\rangle.\mathsf{go}\ \mathit{be}. *(\mathbf{m}(\mathit{sendRec}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1, \{p_1, r\}\rangle\ \mathsf{then}\ (f_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1\ )\end{aligned}$$

The resulting monitor obtained through the compilation of the original expression hence becomes

$$\begin{aligned}&\mathsf{new}\, \mathsf{c}.(\\ &\quad \mathcal{G}\{\![s?\langle\rangle.\mathsf{go}\ p_1 . *(\mathbf{m}(\mathit{req}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle\rangle\ \mathsf{then}\ (c!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1 \parallel \\ &\qquad \mathsf{new}\, s_1, s_2, f_1, f_2.(\\ &\qquad\quad \mathcal{G}\{\![\mathit{repl}(c, s_1, s_2) \parallel \mathit{or}(f_1, f_2, f)]\!\}^1 \parallel \\ &\qquad\quad \mathsf{new}\, c_1.(\mathcal{G}\{\![s_1?\langle\rangle.\mathsf{go}\ d_1 . *(\mathbf{m}(\mathit{resp}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1,\mathit{false}\rangle\ \mathsf{then}\ (c_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1 \parallel \\ &\qquad\qquad + \mathcal{G}\{\![c_1?\langle\rangle.\mathsf{go}\ \mathit{be}. *(\mathbf{m}(\mathit{sendRec}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1, \{p_1, r\}\rangle\ \mathsf{then}\ (f_1!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1\ ) \\ &\qquad\quad \parallel \\ &\qquad\quad \mathsf{new}\, c_2.(\mathcal{G}\{\![s_2?\langle\rangle.\mathsf{go}\ d_2 . *(\mathbf{m}(\mathit{resp}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1,\mathit{false}\rangle\ \mathsf{then}\ (c_2!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1 \parallel \\ &\qquad\qquad + \mathcal{G}\{\![c_2?\langle\rangle.\mathsf{go}\ \mathit{be}. *(\mathbf{m}(\mathit{sendRec}, \bar{x}, k).\mathsf{if}\ \bar{x} = \langle p_1, \{p_1, r\}\rangle\ \mathsf{then}\ (f_2!\langle\rangle.\mathsf{stop})\ \mathsf{else}\ (\mathsf{stop}))]\!\}^1\ ) \\ &\qquad) \\ &)\end{aligned}$$

The result is rather involving. Moreover, the conversion's inefficiency once more comes to the fore, with the often unnecessary use of synchronisation channels. One could encode simpler, more compact monitors which sequentially migrate to alternate locations. Nevertheless, ψ_M gives us an implementable algorithm for the automatic conversion of regular expressions to migrating monitors. We conjecture this conversion to preserve locality, in that the resulting monitor should never resort to remote trace analysis. However proof of this statement is left as future work.

6.5 Conclusions

This chapter focused on the illustration of different monitoring techniques in mDPi, expressed through regular expression conversions. Through these conversions we are now capable of specifying properties through a (simple) high-level specification language, from which we *automatically* obtain (i) orchestrated, (ii) statically choreographed, and (iii) migrating monitor mDPI implementations. We saw how, at least for the monitoring of regular expressions, applying these three different monitoring approaches turned out to be very similar. In fact, all three adopted the same conversion strategy (figures 6.2, 6.3 and 6.4), with differences in (i) monitor location, and/or (ii) re-alignment policy (*i.e.,* the use of either operator setC or go). However, we do not wish to imply generality of this observation, which is perhaps a result of the simplicity of regular expressions. Instead, we identify the necessity for future investigation on the comparison of distributed monitoring approaches through more expressive specification languages.

One issue which we identify regarding the presented conversions is the *loss of completeness* of the resulting monitors, both *remotely* as well as *locally*. Although the loss of completeness between remote locations is expected (section 3.5), it is undesirable locally. Consider for instance Fig. 6.2 depicting the translation of $E_1.E_2$. The conversion triggers channel c upon the satisfaction of E_1, in turn starting the monitor responsible for E_2. Upon synchronisation (on c), this latter monitor is directed to start analysing events *from that point onwards* through monitor re-alignment (*i.e.,* through go or setC). The problem lies with the fact that there is nothing to stop pertinent events from occurring in that duration. In other words, events which occurred between the satisfaction of E_1 and the triggering of E_2 are ignored, potentially missing certain violations as a result. Analogous scenarios can also be found for the conversion of E^* and $E_1 + E_2$. Two solutions are identified; we either (i) define separate conversions for local and remote conversions, exploiting the sequential nature of the calculus operator $\mathsf{m}(c, \bar{x}, k).M$ (in case of the local conversion), or (ii) we enrich the semantics of setC, giving us control over the monitor counter assignment. In this latter case, the value of the counter state from where the latter monitor should proceed is passed during synchronisation.

Part III

Evaluation

7. Case Study

The following chapter identifies an implementable runtime verification framework for distributed settings, based on mDPɪ. Through this case study, our aim is also to identify *practical aspects* possibly encountered during implementation the calculus. Section 7.1 identifies and motivates these issues, as well as presenting the motivated framework. This is followed by section 7.2, which considers the *extent* to which the framework is capable of solving the identified issues. Section 7.3 explores observations extracted during our proof-of-concept implementation of the proposed framework, achieved through Erlang. Finally, section 7.4 concludes the chapter.

7.1 Overview

We next present an architecture for the runtime monitoring of distributed systems *i.e.,* autonomous, concurrently executing systems communicating through message passing (chapter 3). Each system is assumed to run within different environments, and also potentially admits confidential local information. The framework's design has been *driven* by the mDPɪ calculus, implying *generality wrt.* the choice of instrumentation strategy and specification language. In other words, we present a *meta-framework*, implementing the necessary tools for monitoring in a distributed setting, which can be later *specialised* for particular use.

Our immediate task becomes that of (i) defining a framework which achieves the *monitoring* and *tracing* semantics presented in chapter 5, while also (ii) identifying *practical issues* during the implementation of mDPɪ. As we shall see throughout this chapter, these issues include

- *The implementation of monitor and trace operators* — Mostly including difficulties with monitor migration. More specifically, [38] highlights the difficulty with achieving code mobility remotely, especially *wrt.* highly dynamic variant required by our framework, which automatically migrates and executes code across locations on the fly. Other implementation issues also include the generation and analysis of traces, as well as issues with monitor re-alignment. In other words, who is going to be responsible for extracting local traces per location? By extension, who is responsible for administering the counter state? Finally, we also require a *uniform mechanism* which hides implementation details as to differences in local and remote trace analysis.

- *Dynamic architectures* — Although we have theoretically motivated the need for monitoring distributed systems admitting dynamic topologies, implementing runtime verification frameworks tolerant to such settings admits its own set of practical issues. For instance, how are monitors able to migrate to newly discovered locations? Surely, migrating code requires support at the receiver's end in order for the monitor to execute at the new location. How can we enforce required support? Also, as required by mDPɪ, we need to implement a property agnostic approach which is tolerant to different needs (*wrt.* events of interest) of the various monitors migrating to and from locations.

- *Security Issues* — Relating to information exposure across remote locations, as well as issues with monitor tampering. An immediate worry is the overhearing of remote monitor communication, as well as exposure of monitor logic (from which internal system activity can be inferred) across unsecured interactions. Even worse, monitors can be altered by malicious entities whose intent is to harm (or at least derail the monitoring of) contributing systems, in effect exploiting monitors as a back door to circumvent security mechanisms. We can go further, by identifying that internal system information is also exposed *across locations*, by exposing monitor functionality between remote systems. Clearly, possibilities for exposure, as well as unintended use of monitors are issues we must strive to prohibit.

- *Heterogenous Systems* — In other words, considering implementation issues in face of systems adopting various technologies and protocols. What if the distributed system admits a Java program on a Linux server, and a C# program running on Windows? Although such details are inessential during theoretical development, they become a practical reality during implementation, and is hence something we have to consider.

To this effect, we propose the architecture depicted in Fig. 7.1. The remainder of this section is devoted to a description of how we achieve pertinent monitoring functionality. This is followed by section 7.2, which considers the extent to which this architecture solves (or is amenable to solving) the above issues.

Systems are considered to run within named *environments*; each environment can be taken to represent a distinct location, and/or computational environment (*i.e.,* admitting differences in operating system, hardware *etc.*). Moreover, systems interact across environments by exchanging messages. We take a technology agnostic approach towards the implementation of both the system and its interaction protocol. However, internal system execution, as well as across-environment interactions represent events potentially of interest to the monitoring effort. Informally, both forms of actions represent process output operations in mDPɪ.

The depicted framework is *asynchronous*, in that system and monitor execution are disjoint. Each system generates a log of pertinent events (*i.e.,* a trace per location), which is later analysed

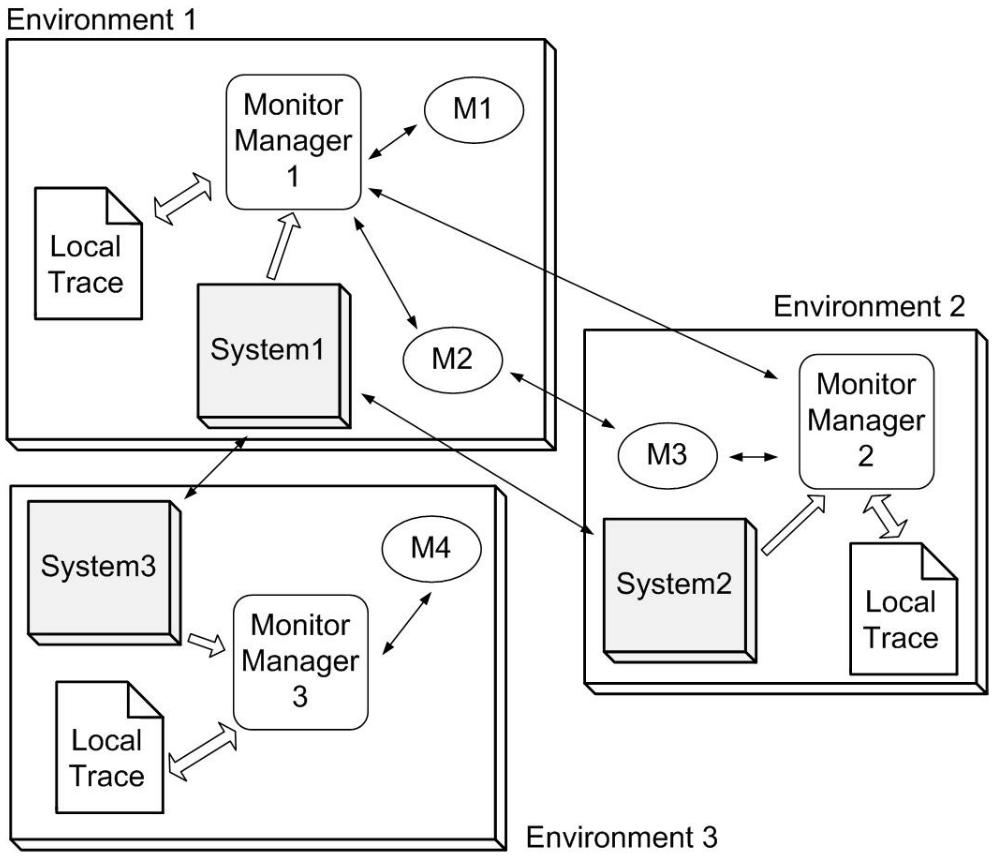


Figure 7.1: An example scenario of the implemented framework.

by monitors. Based on their analysis, these monitors return a verdict on system behaviour. The implementation of this distributed monitoring framework is realized through two components; (i) a *monitor manager* at each environment/location, as well as (ii) an undetermined number of *monitors* per location. These monitors implement mDPɪ operators (section 5.2.1), and are hence additionally capable of (i) interacting and (ii) analysing trace records, both locally and remotely, as well as (iii) migrating across locations. The monitor manager is a novel addition; its responsibility lies as

- *An instrumenter* — intercepting pertinent events, and recording local traces. The monitor manager is responsible for implementing the tracing semantics presented in mDPɪ, thus gradually generating a totally ordered trace per location. The extraction of events is left as a design choice, and can be done either manually, or automatically through the exploitation of some particular technology (such as Aspect Oriented Programming [54]). Nevertheless, each trace record is expected to entail (i) the event name, (ii) associated parameters, and (iii) a (logical) timestamp *wrt.* its temporal ordering within the (local) trace. Note that by implementing the tracing effort, this implies that the monitor managers are also collectively responsible for administering the counter state (section 5.2.1).

Each monitor manager hence admits a monotonically increasing counter, which is used to assign timestamps to (local) trace records.

- *A monitor administrator* — thus responsible for migrating local monitors to remote locations, as well as receiving monitors and spawning their execution locally. In other words, monitor managers interact across locations to exchange monitors, which involves pausing execution at the sender's location and restarting the monitor at the receiver's end (updating the monitor counter accordingly). We shall expand on the implementation of this operation below. The monitor manager also acts as a *trace query engine*, with monitors querying the monitor manager situated at location k for the n^{th} trace entity. Finally, given that the monitor manager administers the trace counter locally, monitors are also able to query the monitor manager (at k) for the current counter value.

The monitor manager hence plays a crucial role in the implementation of the property agnostic approach motivated by mDPI. Each monitor manager exposes a *local alphabet* of events, from which a trace is generated accordingly. Monitors subsequently analyse traces, by *querying* the monitor manager at the required location. In effect, this approach encapsulates a *uniform interface* for trace analysis both locally and remotely, hiding technology details for the implementation of remote communication involved during (remote) trace analysis. Moreover, monitor managers offer an elegant solution for the implementation of monitor re-alignment. Operator $\mathsf{setC}(k)$ is achieved through a request for the counter value to the monitor manager at k. Moreover, as we shall see below, monitor managers (in conjunction with the adopted monitor implementation) also enable migration.

The framework admits an *interpreted monitor expression* strategy, entailing; (i) an mDPI *monitor expression*, and (ii) an *expression interpreter*. The former specifies monitoring code expressed as an mDPI monitor. Its description is best achieved through some language-agnostic markup language, such as through an *XML-based specification*[1]. The expression interpreter is an executable program capable of executing within the local environment. More specifically, its task is to accept a monitor expression, and execute it accordingly. Most of the monitor operations in mDPI are easily encodable in any turing-complete language, including branching and recursion. We employ a lazy interpretation of expression $*M$, in that a copy of M is only created when necessary. Clearly, an eager interpretation of the expression is impossible given finite memory. Inter-monitor interactions (both local and remote), as well as the creation of new channels depends on the native language's support for point-to-point based communication. This functionality is best implemented through *sockets* [84] for maximum generality (across languages) — the socket's port, ip pair are taken to encode the channel name. Parallel composition requires support for concurrent programming, possibly through the use of threads. Finally, monitors are at most allowed to signal the violation (or satisfaction) of properties. This implies that no compensatory actions are allowed. By extension, monitors do not directly interact with

[1]For the full specification visit http://www.w3.org/TR/REC-xml/

system executions.

Given the above setting, monitor migration is achieved in straightforward fashion through a sequence of operations described below.

(i) As soon as an expression interpreter encounters an expression of the form $\mathsf{go}(k).M$, it requests its local monitor manager for the transfer of expression M to location k.

(ii) The interpreter terminates locally. Nevertheless, given that the monitor state is encoded in the expression, this implies that residual monitor M is in a suspended condition.

(iii) The local monitor manager subsequently interacts with the monitor manager at k, transferring the monitor expression in the process.

(iv) Once the latter monitor manager receives the expression, it subsequently spawns a new interpreter in order to execute it.

Expression M hence resumes execution at k, thus achieving required migration. This concludes our discussion of how the framework in Fig. 7.1 achieves necessary monitoring functionality.

7.2 Practical Considerations

The following section reviews the suitability of the proposed architecture *wrt.* the practical issues identified in section 7.1.

The Implementation of Monitor and Trace Operators

We identified four potentially problematic operations at an implementation stage, including trace generation and analysis, as well as monitor re-alignment through counter update and migration. The former three were adequately solved by the introduction of the monitor manager. On the other hand, monitor migration requires a notion of *code mobility* [38] in order to transfer a monitor's execution across locations. This was successfully encoded in our framework through the adoption of monitors as interpreted expressions, in conjunction with monitor managers. More specifically, we have in effect simulated *weak mobility* through *data transfer*, at an additional cost of adopting an interpreted language. Weak mobility refers to the capability of transferring program code across locations (*i.e.*, no state transfer).

However, the application of *strong mobility* may also be worth further consideration. Strong mobility involves the transfer of an executing program by transferring (i) its code, as well as (ii) its state — both explicit as well as implicit (such as the internal program counter, stack *etc.*). The notion of strong mobility is a more powerful technique than its weak counterpart, and may be required in the future if we require migration of more expressive monitors (especially monitors requiring more complex state). Moreover, implementing migration through strong mobility denotes elevated encapsulation of monitor functionality, since we do not depend on the monitor manager for migration to occur. The resulting monitors may also be more efficient, by eliminating our dependance on an interpreted language. On the other hand, strong mobility also introduces new issues. Firstly, from a technology point of view, support for strong mobility is rather sporadic [38], and very technology centric. In other words, a technology which has been implemented for Java might not exist for C#. Moreover, to the best of our knowledge no work has been carried out on mobility *across* languages, which is a big problem when facing heterogenous environments (more below).

Dynamic Architectures

Monitor managers can also help in enforcing necessary support for runtime monitoring across newly added locations. All we need is to ensure each system added at runtime to be instrumented with its local monitor manager. This monitor manager subsequently exposes an alphabet of local events, and records the (local) trace as necessary. Moreover, monitors can now query the monitor manager present at this new location for remote trace analysis (in case of orchestrated instrumentation). Finally, existing monitors obtain the capability to migrate to the new location, by transferring their expression to the new manager.

By extension, the possibility for *dynamic properties* should also be noted (section 2.4.1). In order to monitor newly learnt properties at runtime, we can simply convert their specification into an mDPi monitor expression, and start their verification on the fly *i.e.*, without the need for system recompilation and restart. Analogously, properties can also be altered at runtime, by altering their expression.

Security Issues

Security within the presented architecture can be an important consideration for reasons outlined in section 7.1. In truth, the *urgency* for secure monitoring depends on the scenario at hand. For instance, applying our framework within a private network is less of a risk than monitoring internet-based systems. We next describe necessary security measures in face of (i) unsecure communication mediums, and/or (ii) untrustworthy systems *i.e.*, systems which are not privileged to gain knowledge of each other's internal activity.

Securing monitor interactions can be achieved through the use of *public key encryption* [70], by assigning each monitor manager a public and private key. The former is readily available to all other monitor managers, whereas the latter is confidential and is kept securely at each location. In order to securely exchange monitor expressions, managers at the sender and receiver's location take the following steps

(i) The monitor manager at the sender's location encrypts the expression with its private key.

(ii) This manager also obtains the public key at the receiver's end, and proceeds to encrypt the message with this latter key.

(iii) The encrypted expression is sent to the receiver's monitor manager, which first proceeds to decrypt the message with its private key.

(iv) The latter monitor manager again decrypts the message, this time with the sender's public key, thus obtaining the decrypted message.

(v) An analysis of this message is commenced; if it successfully describes a monitor expression then the message is accepted, else the message is discarded.

The above strategy ensures *confidentiality* of the transferred message, as well as *authenticity* of both the sender and receiver. Moreover, the final step ensures that any message tampering during transfer is detected, which prompts the monitor manager at the receiving end to discard the received message (possibly requesting re-transfer). Note that an analogous approach can be adopted for monitor interactions, perhaps by delegating necessary encryption and decryption to their local monitor managers.

The next issue lies with exposure of information across locations. Consider monitor expression

$$k\{\![\mathbf{m}(e_1, \langle\rangle, k).\mathsf{go}\ l.\mathbf{m}(e_2, \langle\rangle, l).\mathsf{fail}]\!\}^1$$

which specifies that the occurrence of event e_1 at location k followed by e_2 at l denotes property failure. Clearly, this monitor is first started at k, and eventually proceeds to l. However, by placing the monitor at location k we have unnecessarily exposed the occurrence of event e_2 at l. In general, exposing a monitor expression at one location indirectly exposes knowledge of subsequent locations' internal activity. Although this occurrence may not be problematic in certain scenarios, others may prohibit such exposure. This is especially true when systems admit confidential information (*i.e.,* online banks, e-government services *etc.*), and/or an element of competitiveness is present amongst contributing systems (such as the monitoring of web services which offer competing products). To resolve this issue we propose the *staggered encryption* of monitor expressions, in order to ensure that each component is only aware of what it should check for, and to whom control should be passed. In effect, we are motivating the *signing of monitors*. Using this approach, the above example takes the form

$$k\{\![\underbrace{\mathbf{m}(e_1, \langle\rangle, k).\mathsf{go}\ l.\underbrace{\mathbf{m}(e_2, \langle\rangle, l).\mathsf{fail}}_{pub_l}}_{pub_k}]\!\}^1$$

At its outermost level, the expression is encrypted through the public key belonging to the monitor manager at location k. It is hence only this manager (at k) which can decrypt the next operation in the monitor's expression. However, crucially note that upon decrypting the expression, the manager at k only knows (i) the immediate operation to carry out, and (ii) the next location where the expression is to be sent. The remainder of the expression is encrypted by the public key belonging to the manager at *l*, and can only be decrypted upon transferring the expression to *l* through the execution of operator go. In truth, although this overlap of security, runtime verification, and monitor signing was presented as more of a practical consideration, it is an interesting field worth further study at a technical level, and is left as future work.

Heterogenous Environments

Applying our framework in the face of heterogenous environments is also achievable, as long as we implement (i) a monitor manager, and (ii) an expression interpreter for each environment configuration. In other words, each monitor manager implementation will be responsible for extracting necessary information from specific systems (possibly using technologies specific to the language used to write the system). Moreover, the use of (language-agnostic) sockets, as well as XML-based specifications for describing monitor expressions allows remote monitors *and* monitor managers (possibly written in different languages) to interact as necessary, abstracting over inessential technology details across environments.

Similarly to programming languages such as Java and Erlang, an interesting approach we may consider in the future is the view of the monitor manager as a *virtual machine*. In other words, the monitor manager acts as an execution environment (at the process level, rather than at the level of the operating system) within which monitors perform necessary verification. Apart from providing a sense of *standardization* to the monitor's execution environment — necessary in the case of heterogenous environments — concurrent monitors may also provide better performance through *process-level concurrency*, a fact which has been verified with Erlang processes [7]. Finally, the use of virtual machines provides *isolation* by executing monitors in a sandbox environment. This provides standard (desirable) repercussions *wrt.* fault tolerance and security. For instance, malicious monitors would not have access to memory beyond that afforded to its monitor manager. Also, if the monitoring effort crashes at one location, it can easily be restarted without affecting the underlying system.

7.3 Observations

Throughout the following section we shall discuss observations gathered during our proof-of-concept implementation of the above framework. This implementation has been achieved through *Erlang* [7]; a multi-paradigm programming language with strong emphasis on *concurrency* and *distribution*. The use of Erlang for a prototype implementation was a straightforward choice, since its notion of distributed systems easily maps to our setting. The language emphasises the encapsulation of program logic into concurrent *lightweight processes*. By lightweight we imply that processes are built as efficient language primitives, rather than expensive operations (typically involving the operating system). Processes communicate through message passing techniques (*i.e.*, no shared memory). Moreover, the language offers native support for distribution in a similar way to our understanding of named locations. More specifically, each node on the network is taken to represent a computational environment for processes to execute in, and is qualified through a unique identifier. Processes execute concurrently and interact both locally and remotely through language constructs which abstract over differences in location. Crucially, node identifiers explicitly provide processes a notion of located computation, mapping directly to the scenario in Fig. 7.1. Finally, it is interesting to note that Erlang has strong support for fault tolerance techniques, implying that it could be an interesting tool for the study of fault tolerant monitoring in the future.

Our first observation relates to the need for *trust* in the framework implementation. More specifically, we require system administrators to trust a monitoring framework which (i) exposes local information, both locally as well as to remote entities, (ii) may pose security risks, (iii) accepts (and transfers) monitor scripts, which can expose internal activity, and by extension (iv) allows for numerous script interpreters to be executed at runtime, consuming resources otherwise available to the system . Clearly, this is quite a lot to ask, implying that the framework may not be suitable in all scenarios. Although we have outlined solutions for certain issues (including security, safety of running monitor scripts etc), these come at an additional resource cost.

Secondly, the viability of migration through data transfer is driven by our encoding of the monitor's state through its expression. More precisely, the mDPI semantics dictate that in order to determine a monitor's next computational step, we only need to know (i) its expression, and (ii) the counter state. Given that the counter state is simulated through monitor managers, transferring expressions should suffice; there is no need for additional references to memory-based state. In fact, the only stateful component in a monitor's description is the counter. However, its current value is explicitly stated as part of the expression too. It is for this reason that weak mobility suffices in our case. Nevertheless, we may require strong mobility in the future, especially if more expressive monitors are adopted, or monitors with more complex state. Interestingly, Erlang allows for *hot swapping* of code [7], which permits for dynamic update of programs without the need for recompilation. However, although this particular technique might help during extension of our prototype framework, to the best of our knowledge similar functionality does not exist in more traditional languages. Finally, the notion of monitor managers as virtual machines might also help for simulating strong mobility, since it gives us complete control over the monitor state (both implicit and explicit).

Our third observation relates to our use of monitor expressions. In effect, we have presented what can be considered a *monitoring programming language* i.e., we provide a language providing the necessary primitives in order to achieve the monitoring of distributed systems. Through this language, numerous specification languages and/or instrumentation strategies can be expressed. Monitor code is written in a script which describes its operation, which is later interpreted across distinct entities. Although expression interpretation helps us in achieving weak mobility, it is not a necessity. As discussed above, this is especially true if we execute monitors within virtual machines. Irrespective of the execution strategy, by adopting a monitor language we obtain complete control over the monitor's instruction set, which admits desirable repercussions *wrt.* security. More specifically, by disallowing for harmful monitor operators, it is impossible for damaging monitor scripts to be devised. In other words, malicious users can at most only derail the monitoring effort, not cause harm to the underlying system. Clearly, this may change as soon as reparatory actions are introduced.

7.4 Conclusions

This chapter presented an implementable architecture for the realization of mDPɪ. Moreover, we anticipated practical issues which might arise during implementation of the motivated framework, and also offered possible solutions. These issues include problems with dynamic and/or heterogenous systems, as well as potential security liabilities. Another source of discussion was the implementation of monitor migration through two forms of code mobility. On the other hand, we identify a potential source of weakness *wrt.* the associated level of monitoring overhead. Although we do not measure said overhead throughout this dissertation, we recognise that having a monitoring framework which is (i) is based on an interpreted language, and (ii) uses encryption algorithms may require considerable resources, which goes against the principle of adopting lightweight monitors (section 2.2).

Finally, our current prototype implementation written in Erlang is intended as a proof of concept, and does not implement all the functionality described above. More specifically, this implementation does not yet support dynamic or heterogenous environments (all systems are Erlang nodes), and assumes a reliable network infrastructure (*i.e.*, no implementation of security algorithms). Nevertheless, we believe this chapter, along with the developed implementation, should provide enough insight into the development of the full architecture if necessary.

8. Related Work

The following chapter is dedicated to (i) an in depth review of current runtime verification approaches in a distributed setting, and (ii) the comparison of related work to the mDPɪ framework, thus also serving as an evaluation of our work. Section 8.1 introduces the chapter, presenting a broad picture of the identified approaches. Section 8.2 goes into some detail on the operation of each approach. This is followed by section 8.3, which summarises the identified tools, and also conducts an a posteriori comparison to our work. Finally, section 8.4 concludes the chapter.

8.1 Introduction

The following section entails a detailed discussion of encountered candidate solutions for the monitoring of distributed systems. In general, these solutions vary widely in their scope. Some may be more focused on the monitoring of particular technologies (such as web service compositions, or Enterprise Service Bus architectures). Most vary in their assumptions about the underlying system's operation. For instance, some approaches assume the availability of a global clock, thus achieving synchrony across remote locations. In general, we shall look at the mechanisms adopted by each framework, extracting their proposed solutions for pertinent issues we identified in chapter 3. We shall especially keep a look out for (i) their classification *wrt.* the identified taxonomy (*i.e.,* identified as either statically choreographed, orchestrated approach etc), (ii) the issue with asynchrony amongst remote locations, (iii) support for dynamic architectures, and (iv) the recognition of risks with monitoring in the face of confidential information.

We shall also be comparing approaches through an example scenario (where possible). However, given that (as we shall be seeing) certain approaches do not support dynamic architectures, comparing approaches through the hospital management system presented in section 3.3 shall not be possible. Instead, we adopt another scenario, involving a static architecture whose subsystems are known a priori, and remains unchanged throughout execution. More specifically, we shall consider a travel agent responsible for booking holidays on the client's behalf. The agent interacts with the bank, hotel and airline booking agencies, booking the best deal which can match the client's request. One recurring property which we shall be looking to verify involves the check that at no stage can the travel agent book flight and accommodation which exceeds the client's bank balance. In case that the framework supports reparation, a suitable compensatory action in this case involves rolling bank triggered transactions in such a scenario. Also note that this scenario admits confidential information (*i.e.*, bank transactions, flight and hotel bookings), which gives us a medium to check whether frameworks take precautions against the leaking of such data.

8.2 Current Approaches

The following section describes *six* monitoring frameworks which attempt the verification of distributed systems.

DMaC

DMaC [88] is a distributed extension to the Monitoring and Checking (MaC) framework [56]. MaC is a runtime verification framework designed for the runtime monitoring of monolithic systems, and is capable of monitoring various safety properties (including temporal properties). The MaC framework is based on a two-layered specification language called *MEDL* and *PEDL*. The Meta Event Definition Language (MEDL) is a programming language agnostic formalism used to express system properties, with its formal semantics and expressivity similar to past-time linear temporal logic (ptLTL) [23]. The Primitive Event Definition Language (PEDL) complements MEDL by specifying the high-level events referred to in MEDL scripts in terms of low-level system function calls and system data structures. Using this two-leveled approach, a generalised runtime verification framework is hence obtained, potentially achieving both language and platform independence.

On the other hand, DMaC is a choreography-based distributed runtime monitoring tool, focusing on the verification of systems specified using *declarative networking techniques* [62], physically distributing the monitoring effort across a network. Such distribution is achieved by integrating the MaC framework within distributed systems specified using the language *NDLog* [62]. NDLog is a domain specific declarative language used for the abstract specification of (i) network protocols and (ii) distributed architectures, and is based on specifying declarative rules. These rules take the form $p\ :-\ p_1,\ p_2,\ ...\ p_n$ with each p_i constituting a localised predicate, parameterised using variables and/or containing boolean functions. Assuming predicate $link(@S, D)$ which returns node D if there is a link between nodes S and D, an example NDLog rule returning (for node S) all direct neighbours (that is, nodes with which node S has a direct link to) is specified below

$$neighbour(@S, P) = link(@S, D), add(D, P).$$

The above rule automatically adds all nodes directly linked to S into vector P.

Analogously to MaC, system properties specified within DMaC are specified using MEDL at a high abstraction level. Consequently, MEDL scripts are complemented with an additional PEDL script describing the distributed nature of the properties specified within MEDL. PEDL scripts are composed of (i) an *event recogniser*, responsible for specifying where primitive events are generated within the distributed system, and (ii) a *locationer*, which specifies which nodes are responsible for the generation of composite events specified within the MEDL. A typical MEDL script contains variables, imports primitive events (defined in PEDL), and constructs more complex events (using available primitive events as well as conditions).

Consider the following example MEDL script, which is based on a PEDL specification defining primitive events (i) *airlineBooked(a,c,p)* recognising that a plane ticket was booked with agency *a* for client *c* for the price of *p*, (ii) *hotelBooked(h,c,p)* specifying that a booking was placed at a hotel *h* for client *c* for the price of *p*, and (iii) *bankBalance(c,b)* stating that client *c* has balance *b* (we shall assume that this event is triggered at the start of the client's request). Through these events, the following MEDL script raises an alarm when the cost of booking the airline and hotel for any one client exceeds his bank account.

```
import  event                                 airlineBooked(a,c,p)
import  event                                 hotelBooked(h,c,p)
import  event                                 bankBalance(h,c,p)
var     client, balance
event   triggerBalanceSubtractionAirline(p)   = airlineBooked(a,c,p) when (client = c)
event   triggerBalanceSubtractionHotel(p)     = hotelBooked(h,c,p) when (client = c)
event   raiseAlarm                            = (airlineBooked(a,c,p) ∨ hotelBooked(h,c,p))
                                                when((client = c) ∧ (b < 0))
        bankBalance(c,b)                      -> {client = c; balance = b;}
        triggerBalanceSubtractionAirline(p)   -> {balance = balance - p;}
        triggerBalanceSubtractionHotel(p)     -> {balance = balance - p;}
```

The first three lines import the necessary events (defined through PEDL). The fourth line declares required variables acting recording necessary state. The next three lines denote the specification of composite events, such that (i) *triggerBalanceSubtractionAirline(p)* and *triggerBalanceSubtractionHotel(p)* update the monitor's recorded bank balance upon the booking of an airline or hotel accomodation, and (ii) *raiseAlarm* flags the necessary violation when booked airline and hotel accommodation exceed the client's balance. The latter three lines of the above script are specified to update auxiliary variables *client* and *balance*. Note that MEDL extensions for the specification of sliding window event correlation have also been specified, with the role of such extensions being the evaluation of simple statistics (such as the count and average) over attributes defined within events and conditions.

The above MEDL script is first converted to location-agnostic NDLog rules through a process of normalisation followed by conversion through a set of templates. The final step in the compilation entails assigning locations to the resulting rules, and is executed by the *planner* component. Choosing node locations for the resulting rules is non-trivial, and is done partly based on the information specified in the PEDL script in conjunction with query optimisation techniques. Although choosing the optimal set of locations is in general NP-hard, the authors claim that approximate solutions using dynamic programming in conjunction with certain heuristics provide reasonable location configurations. Moreover, given that network performance changes over time, adaptive query optimisation techniques have also been considered as future work, allowing for re-optimisation of monitoring configurations in an online fashion [88].

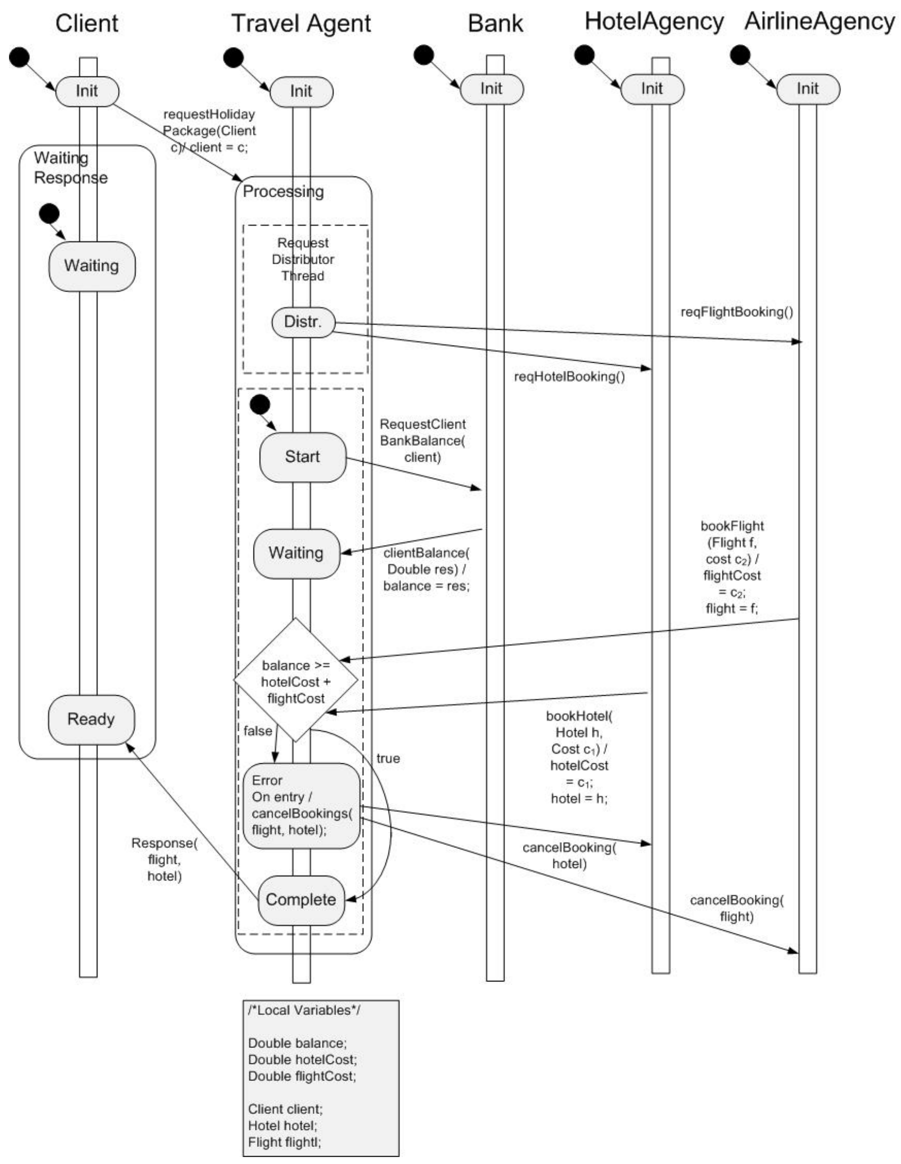
Given that the resulting rules (converted from the high-level MEDL script) are (i) compiled prior to system execution to NDLog rules, (ii) executed at physically different locations, and (iii) whose location remain unchanged throughout, this implies that DMaC is a *static choreography-based approach*. DMaC does not consider issues such as dynamic architectures or information confidentiality, and is instead focused on the minimisation of bandwidth overhead over static architectures. More specifically, even though the approach is choreographed, given that rules located at certain nodes can listen to remote events, this results in possible exposure. Note that the architecture has been experimentally shown to produce acceptable bandwidth overhead, with overheads reported to be around 11% [88]. Finally, from a practical point of view, one flaw with the solution presented by DMaC is its tight binding with NDLog. Although the NDLog language greatly facilitates distributed monitoring through its powerful declarative expressivity, most full-blown industry scale distributed systems are not implemented using declarative networking techniques. Monitoring such systems using DMaC would hence involve (i) exposing local system information stored within distributed tables, (ii) upon which NDLog rules can subsequently operate, hence involving additional overheads. In truth, DMaC is perhaps best suited as a formal tool for the verification of network protocols, whose description using declarative techniques are gathering popularity.

Runtime Monitoring Based on MSC Assertions

The approach presented by Drusinksy *et al.* [77] presents a distributed synchronous runtime verification approach of Service Oriented Architecture (SOA) [32] based *system-of-systems* (SoSs). The term system-of-systems loosely refers to a set of goal-oriented systems, interacting and sharing resources in order to achieve a common goal, and is hence not too unlike our understanding of distributed architectures in section 3.2. Such task-oriented systems are commonly designed in accordance with SOA principles and developed through web services, so much so that SOA based SoSs can be viewed as a set of coordinated web services running over the internet [77].

The authors argue that questions arise regarding the reliability of such systems due to their (i) inherent concurrency, and (ii) unreliable communication mediums which they are forced to make use of. Hence, the global system is monitored at runtime by comparing the system's runtime behavior against a set of formally specified system properties, thus enabling the verification of such systems in less-than-ideal situations. The presented approach is interested in system of systems with no central agency coordinating workflow, which is why the resulting monitoring functionality is executed in distributed fashion throughout the global system.

Message Sequence Chart (MSC) assertions have been proposed as an appropriate formalism for property specification, and are based on UML MSCs and UML statecharts. The diagram below specifies an example MSC assertion, specifying the property that the automated travel agent stating that the cost of booking the flight and hotel do not exceed the client's balance.


Client
Travel Agent
Bank
HotelAgency
AirlineAgency
Init
requestHoliday Package(Client c) / client = c;
Waiting Response
Waiting
Ready
Processing
Request Distributor Thread
Distr.
reqFlightBooking()
reqHotelBooking()
Start
RequestClient BankBalance(client)
Waiting
clientBalance(Double res) / balance = res;
bookFlight (Flight f, cost c2) / flightCost = c2; flight = f;
balance >= hotelCost + flightCost
false
true
bookHotel(Hotel h, Cost c1) / hotelCost = c1; hotel = h;
Error On entry / cancelBookings(flight, hotel);
cancelBooking(hotel)
cancelBooking(flight)
Response(flight, hotel)
Complete
/*Local Variables*/
Double balance;
Double hotelCost;
Double flightCost;
Client client;
Hotel hotel;
Flight flightl;

The property follows the sequence of interactions *across* locations *i.e.,* the interactions between the client and the travel agent, as well as the travel agent with the bank and booking agencies. If the property is satisfied, then the monitor simply checks that the travel agent response (containing the booked flight and hotel) are sent to the client. However, if the property is broken, the booked flights and hotel are canceled, in order to prevent the agent from spending beyond the client's means. The approach hence admits a notion of reparation. As depicted in the diagram, the specification language also has access to local variables (whose state cannot be shared amongst locations). In true UML-MSC fashion, the above assertion defines one task per each entity of interest (client, bank, booking agencies *etc.*). However, unlike UML-MSCs each task is represented by a statechart (instead of the standard timeline), with each task representing a progression of states. Each statechart represents an automaton-like structure by admitting states and transitions, with the latter triggered by events occurring within the system. Each state can contain associated code, executed each time that state is reached. Also, each transition takes the form *event[condition]/Action*, such that each time the specified *event* occurs and the associated *condition* is satisfied, the corresponding *Action* (in the form of Java/C++ code) is executed. The condition is based on parameterised events. Also, note that the statecharts embedded within MSC assertions also have a timer construct. However, each constructed timer can only be used within the task it has been defined, thus avoiding the problem of asynchronous clocks amongst remote locations. Finally, note that interaction between statecharts is hinged on interactions across systems within the underlying SoS.

MSC assertions are monitored through the *Runtime Execution Monitoring (REM)* framework, entailing a set of methods implemented for (i) observing the underlying application and (ii) updating each statechart (within the assertion) as required. Hence, monitoring an MSC assertion involves embedding the code (generated from MSC assertions) at each of the web services' location. Given that the resulting monitoring functionality is (i) compiled prior to execution, (ii) installed across the SoSs with no central monitor, and (iii) remains unchanged throughout execution, then the approach presented by Drusinksy et al. is a static choreography based approach. No explicit reference is made to issues of information confidentiality. Moreover, dynamic architectures are also not handled, since new web services (such as in dynamically discovered services) cannot be monitored on the fly. Although the above approach as presented in [77] admit that the monitoring framework may operate using an unreliable communication medium, it fails to characterise this unreliability. Finally, although the presented approach may yet be feasible for other distributed architectures, its current implementation binds the framework's application to SOA based SoSs, and is hence not applicable to heterogenous environments.

GEM

Mansouri *et al.* present GEM [64], a generalised event monitoring framework based on a *declarative* and *interpreted* rule-based specification language. The presented approach is also event-based, basing its verification on the observation of (i) *primitive events* (basic observable events within each systems) or (ii) *composite events*, relating primitive event sequences gen-

$$E_c \quad ::= \quad E_p \ \| \ E_c \& E_c \ \| \ E_c + t \ \| \ \{E_c; E_c\}!E_c \ \| \ E_c | E_c \ \| \ E_c; E_c$$

Figure 8.1: The GEM specification language.

erated throughout the distributed system. Moreover, the framework is based on the notion of *rules*; triggering a sequence of reactions, upon the observation of an event. Similarly to mDPI the GEM framework considers typical distributed systems to consist of numerous loosely coupled subsystems, executing at physically separate locations and communicating via message passing. Crucially, GEM considers subsystems to have no shared memory, while also acknowledging unbounded communication delays with no guarantees on the preservation of message order. It is also worth noting that the framework presupposes a *global synchronised clock* accessible to all subsystems, simplifying the problem of obtaining order in a distributed system.

Each subsystem is instrumented with an *event monitoring service*, consisting of three subcomponents, these being the (i) *event generator*, (ii) *event monitor* and (iii) *event disseminator*. The event generator is responsible for observing the system's behaviour and extracting primitive events from the local system. The event monitor executes the specified rules by processing, correlating and filtering events generated both locally as well as those received from remote systems. Finally, the event disseminator transmits the occurrence of events (both primitive and composite) to other remote monitors within the architecture, since these events may be pertinent to the system's global monitoring effort. The resulting framework is hence choreography based, since events are observed locally and transmitted remotely. In fact, Mansouri *et al.* advocate that centralised correlation and monitoring of event reports is not practical in very large distributed systems due to the elevated bandwidth requirements. Interestingly, the approach taken by GEM which *interprets* rules at runtime allows for additional rules to be loaded at runtime at each local monitor, which hints to the monitoring of dynamic properties. However, the framework does not explicitly mention the issue of dynamic architectures. Certainly, dynamically adding new systems at runtime would require these new systems to have their local event monitoring service installed prior to execution.

GEM specifications involve a two tiered approach, with the user tasked with specifying (i) events and (ii) rules. Events can either be primitive or composite events. Primitive events can either be generated by the underlying system, or adhere to either format *every[<time_expression>]* or *at[<time_expression>].* An example *time_expression* includes *(hour*$*$24 + *day)*, and is used when events are to trigger either periodically (through operator *every*) or at fixed points in time (using *at*). A special '*' wildcard operator can also be specified to match all possible events. A composite event entails a set of primitive events combined through a set of operators, and whose syntax is defined in Fig. 8.1.

where E_p denotes a primitive event, and t represents a time expression. Operators & and

| act as a conjunction and disjunction of events respectively, whereas operator ; checks for event sequentiality. Operator + triggers the composite event t time units after the triggering of the original event, hence effectively delaying the triggering of the event. Finally, operator $\{e_1; e_2\}!e_3$ checks that events e_1 and e_2 occur in sequence without interleaving of event e_3. Note that both primitive and composite events can specify an additional guard (a boolean condition over the event attributes belonging to the event), acting as a filter on events. Also note that the timestamp of each event can be extracted using the implicit @ operator. Given that all locations are synchronised, the timestamp at one location is relevant to all other locations. However, although the occurrence of primitive events is considered instantaneous, composite events may span over time intervals. Hence, the start and end of an interval event's time interval is extracted using the @ and |@ operators respectively. This information can then be used within the associated events' guards. Assuming that the travel agent emits primitive events of the type *bookFlight(cost,client)*, the following are a few example composite events.

- bookFlight.cost when $cost > 500$
- $at[14:00]$; changeTemperature ; $at[16:00]$
- (x:bookFlight ; y:bookFlight) when $((x.client = y.client) \&\& ((@x - @y) < 30M))$

The syntax x:e simply assigns a variable name x to event e within the expression. The first example specifies a primitive event which triggers on any flight booking which exceeds $e500$. The second example specifies that events of type *bookFlight* should only be considered between 14:00 and 16:00, whereas the final example specifies the composite event which triggers when the same client makes two bookings within a month or less.

Rules within the GEM framework take the form

rule <rule_name> [detection_window] { *<event_expression>* ==> *<action_sequence>* }

thus in a certain sense allowing for an update of the monitoring effort on the fly.

specifying that the *action_sequence* is executed upon the triggering of the *event_expression*. The *detection_window* is an additional mechanism which specifies that the monitor verifying the associated rule should store the event history for all pertinent events (referred to by the rule) for the duration specified. The result is a temporally ordered list of events per location, such that the rule is re-evaluated on this list for each identified event (observed or received from a remote location). Also note that possible actions include notifying, ordering and triggering events, as well as the execution of *control commands*. These commands execute administrative tasks are used for controlling the operation of monitors, including the addition, removal, pausing and resuming of rules. Assuming the previous *bookFlight(cost,client)* event (at the travel agent) and another event *withdraw(amount,client)* emitted by the bank which withdraws a specified amount for that client, consider the following GEM script. More specifically, this script specifies a rule that the booking of a flight followed by a withdrawal followed by another flight within 10 seconds or less (for the same client) indicates an incorrect double booking. If this is the case, an additional *alarm* event is triggered.

The following is an example GEM script specifying a rule that when two successive rises in temperature occur in a time period of less than 10 seconds such that their gradual change is larger than 5 degrees, an *alarm* event is triggered.

```
event bookFlight(cost,client)
event withdraw(amount,client)
event alarm
rule setAlarm { x:bookFlight ; y:withdraw ; z:bookFlight
  when ((x.client = y.client) && (y.client = z.client)
                                    && ((@z - @x) < 10))
  ==> trigger alarm; }
enable setAlarm
```

The following example specifies another example for the automated travel agent scenario, specifying the recurring property that costs for booking a flight and hotel should not exceed the client's bank balance.

```
event bankBalance(client,balance)
event bookFlight(cost,client)
event bookHotel(cost,client)
event alarm
rule setAlarm { x:bankBalance & y:bookFlight & z:bookHotel
  when ((x.client = y.client) && (x.client = z.client)
                        && ((y.cost + z.cost) > x.balance))
  ==> trigger alarm; }
enable setAlarm
```

Note that the event expression specifies that primitive events *bankBalance*, *flightBooked* and *hotelBooked* occur in any order. Like the previous example, this expression also specifies that the client for each event is the same, and that the *cost* for booking the flight and hotel combined exceeds the client's balance. If the event expression is satisfied, then the alarm event is triggered.

One point worth noting is the framework's solution to the issue of dealing with communication delays and loss of event ordering. Although the detection window mechanism (discussed above) is helpful for the solution of the event ordering issue, it is insufficient as shown in the following example. Say three events e_1, e_2 and e_3 occur remotely within a time-span of 10 seconds, and their occurrence is detected remotely (by a remote monitor) in the order e_1, e_3, e_2 within a timespan of 20 seconds. Although these events will be eventually ordered correctly through their (globally valid) timestamp, there still remains the problem that certain rules may trigger upon detecting e_3, which might have acted otherwise had knowledge of e_2 arrived before. An example composite event which exhibits such behaviour includes $\{e_1; e_3\}!e_2$. Given the detection of events (at the remote monitor) e_1 and e_3, the composite event incorrectly triggers, since it has no knowledge that e_2 will eventually be detected, and has in fact occurred between events e_1 and e_3 (and is hence unsatisfactory to the specified composite event). The solution proposed by the GEM framework for such circumstances is by using an *event-specific delaying technique*, such that *each* event (both primitive and composite) is explicitly assigned a delay duration before making any rule judgements. This delay allows for the arrival of additional events which might alter the rules' outcome. Events which arrive beyond this timeframe are ignored. Hence, the previous composite event would be specified as $(\{e_1; e_3\}!e_2) + 20s$, giving the composite event enough leeaway for any additional events to be detected before taking any decision whether to fire. Finally, note that the GEM framework uses a tree-based representation of events allowing for the efficient evaluation of GEM rules. Each node represents an event, and contains an associated guard and event history (as far back as dictated by rule detection windows).

In conclusion, the GEM framework offers a valid solution to the monitoring distributed systems. Given that an event monitoring service is installed at each location, this implies that monitoring functionality is distributed, and is hence a choreography based approach. Moreover, although dynamic architectures are not explicitly mentioned, this framework hints to the monitoring of dynamic properties. On the other hand, GEM is based on a heavy assumption *wrt.* the availability of a global clock. However, as we discussed in section 3.2, this is usually unattainable in typical distributed system implementations. Another issue involves GEM's handling of error-prone communication through event-specific delaying techniques, which entails setting a fixed delay a priori for the firing of rule judgements. Although the framework requires the fixed specification of delays, determining the optimal duration is often dynamic and unpredictable. The framework partly attempts to get around the problem by interpreting the specified delay as the *tolerated delay*, as opposed to the *expected delay*. On the other hand, one issue which is completely ignored is that of information confidentiality; events are continually transmitted to remote locations irrespective of whether they admit sensitive information to the local system.

Runtime Monitoring of Web Services Compositions through RTML

The approach presented in [10] is exclusively concerned with the runtime monitoring of specified web services implemented in BPEL. BPEL [69] is an XML-based executable scripting language which coordinates interactions between web services, and is executed by a central BPEL execution service in charge of executing such coordination. System properties are specified through the *Runtime Monitoring Specification Language* (RTML), an event-based specification focused on the specification of boolean, invariant, numeric and temporal properties. Note that the presented framework supports automatic compilation and instrumentation, but not violation reactions. The main motivation behind the approach is that web services are often developed independently, and moreover often change without prior notification. Hence, this framework serves as an additional check that web service compositions are operating as required. On the other hand, note that by focusing on the interaction between web services, this approach ignores the internal operation of each system.

The RTML specification language is based on a two-tiered approach, allowing for the specification of *instance* and *class monitor* formulas. Whereas instance monitors are concerned with monitoring the behaviour of individual BPEL process instances, class formulas specify properties over all instances of a particular process. Events are the basis upon which formulas are built, whereby four event types are recognised: *creation* and *termination* of a BPEL process, as well as *input* and *output* of messages. Events can also be parameterised in order to extract information regarding the event itself, such as for example *msg(client.output = sendRequest)*, which specifies an event triggered when the BPEL process representing the client outputs a request. Instance monitor formulas are specified using the following structure

$$\begin{array}{lll} \mathrm{b} & ::= & \mathrm{e} \mid \mathbf{Y}\,\mathrm{b} \mid \mathbf{O}\,\mathrm{b} \mid \mathbf{H}\,\mathrm{b} \mid \mathrm{b}\,\mathbf{S}\,\mathrm{b} \mid \mathrm{n} = \mathrm{n} \mid \mathrm{n} > \mathrm{n} \mid \neg\,\mathrm{b} \mid \mathrm{b} \wedge \mathrm{b} \mid \mathrm{b} \vee \mathrm{b} \mid \mathbf{true} \\ \mathrm{n} & ::= & \mathbf{count}(\mathrm{b}) \mid \mathbf{time}(\mathrm{b}) \mid \mathrm{b?n{:}n} \mid \mathrm{n} + \mathrm{n} \mid \mathrm{n} - \mathrm{n} \mid \mathrm{n} * \mathrm{n} \mid \mathrm{n} / \mathrm{n} \mid 0 \mid 1 \ldots \end{array}$$

A boolean formula can either consist of (i) a past time LTL [23] operator (**Y** means b was true in the previous step, **O** refers to b being true at least once in the past, **H** refers to b being always true in the past, and b_1 **S** b_2 refers to b_1 being true since the first time b_2 was true), (ii) a numeric comparison, (iii) a logic operator, or (iv) an event e evaluating to true when the event occurred. A numeric formula can consist of **count**(b), returning the amount of times boolean formula b was true, **time**(b), returning the total time duration b has been true, a conditional expression ($b?n_1 : n_2$) returning n_1 if b is true, or n_2 otherwise, or an arithmetic operation on other numeric formulas. Class monitor formulas are specified similarly to instance monitored formulas, as evidenced by the class monitor syntax below.

$$\begin{array}{lll} \mathrm{B} & ::= & \mathbf{And}(\mathrm{b}) \mid \mathbf{Y}\,\mathrm{B} \mid \mathbf{O}\,\mathrm{B} \mid \mathbf{H}\,\mathrm{B} \mid \mathrm{B}\,\mathbf{S}\,\mathrm{B} \mid \mathrm{N} = \mathrm{N} \mid \mathrm{N} > \mathrm{N} \mid \neg\,\mathrm{B} \mid \mathrm{B} \wedge \mathrm{B} \mid \\ & & \mathrm{B} \vee \mathrm{B} \mid \mathbf{true} \\ \mathrm{N} & ::= & \mathbf{Count}(\mathrm{b}) \mid \mathbf{Sum}(\mathrm{b}) \mid \mathrm{N} + \mathrm{N} \mid \mathrm{N} - \mathrm{N} \mid \mathrm{N} * \mathrm{N} \mid \mathrm{N} / \mathrm{N} \mid 0 \mid 1 \ldots \end{array}$$

Note that b and n refer to instance monitor formulas. Class monitor formulas also distinguish between boolean and numeric formulas. Additional operators include boolean operator **And**(b), which returns true if instance monitor formula b is true for all instances monitoring each individual BPEL process (*i.e.*, quantifying over instance monitor formulas), and numeric operators **Count**(b) and **Sum**(n). Operator **Count**(b) returns the number of BPEL processes which satisfy b, whereas **Sum**(n) returns the total sum over the results returned by computing n over each BPEL instance. These three operators hence act as the link between class and instance monitor formulas. Also, note that an instance monitor is created for each reference to an instance monitor formula specified within an class monitor formula. Finally, although instance monitors terminate with its associated BPEL process, class monitors are persistent, and hence remain active for the duration of the execution of the monitoring functionality. Note that the above approach does not support the specification of conditions on event parameters. This limitation hence stops us from specifying the running automated travel agent example. Consider instead the following example properties specified using the above syntax.

- **H** ((**O** msg(client.input = successfulLogin)) $\wedge$ msg(client.output = logout))
- **count**(sender.output = retrySend)
- **Sum**($\neg$(**H** msg(client.output = buyItem)) **S** msg(client.input = successfulLogin)) / **Count**(**O** start)

The first two examples are instance monitor formulas, whereas the last formula is an example class monitor formula. The first example specifies that it should always be the case that if a logout event occurs, it must have been preceded by a login previously. The second example counts the amount of sender retry events within some transmission protocol. Finally, the third example is a typical statistic frequently used for marketing purposes, which returns the average amount of clients logged in which have not yet purchased an item (from a virtual shop).

Web service composition specified through BPEL inherently implies a central authority (in the form of an execution engine within an application server) coordinating web service calls. The monitoring of such scenarios is hence achieved through an orchestrated approach, by installing the monitor at the central execution engine. More specifically, the presented solution extends the original BPEL execution engine by a set of components, involving (i) the *monitor inventory*, (ii) the *monitor instances directory*, (iii) the *runtime monitor* and (iv) the *mediator*. The monitor inventory contains the set of monitors deployed within the extended engine. The monitor instances directory specifies the set of currently running monitors. The runtime engine caters for the monitor life cycle (for both instance and class monitors), by creating, terminating and updating monitor states according to events observed within the engine. Finally, the mediator is the component responsible for intercepting events within BPEL processes, and alerts the runtime monitor of these event occurrences. However, note that no support is mentioned for dynamic architectures, in that all contributing web services are known a priori through the BPEL specification. Also note that formulas are efficiently executed within monitors according to a specified operational semantics, since evaluating the previously defined operators at worst entails evaluating the operator on the current state and the stored computation on the previous state.

Being orchestrated, the presented approach is conceptually simpler, avoiding complexities introduced when distributing monitor functionality. Moreover, given that this approach focuses on the verification of public interactions amongst web services, moreover by listening to information passing through the (already central) BPEL engine, this implies that the framework does not impose additional bandwidth overheads. Information confidentiality is also not an issue in the above setting, since all monitored interactions are public. However, once more this solution fails to specifically mention dynamic architectures. Moreover, the framework does not support dynamic properties. In general, the work presented in this approach can be considered as an adaptation of current work for the runtime monitoring of monolithic architectures.

Runtime Monitoring of System-of-Systems Integration

Similarly to the work presented in [77], Krüger *et al.* [57] present an approach for the runtime monitoring of distributed system-of-systems. However, this approach is immediately distinguished from previous work SoSs on two fronts. Firstly, this approach is focused exclusively on the runtime monitoring of the *integration* of system-of-systems (*i.e.*, it is not interested in systems' internal operation). Moreover, whereas previous work is bound to monitoring SOA-based system-of-system implementations, the presented approach focuses on SoSs whose system integration is handled by the Enterprise Service Bus (ESB) architecture [18]. The ESB is a modern approach toward the integration of loosely coupled, interacting systems with no central authority to control information and execution flow. The framework provides a heterogenous message-oriented middleware with expressive mechanisms for message interception, manipulation, distribution, and also behaviour injection.

Property specification is based on a two layered approach, involving *Message Sequence Charts* (MSCs) and *High-Level Message Sequence Charts* (HMSCs). MSCs are used to specify each individual system behaviour traits of interest, entailing the specification of interaction patterns and causal communication relationships across systems. On the other hand, HMSCs are used to specify interaction flows *between* MSCs. These flows can take the form of (i) alternatives between behaviour patterns, (ii) iteration, as well as (iii) concurrent join, specifying that two individual aspects of system behaviour (specified using MSCs) are to be monitored concurrently. However, the framework does not support either parameterised events or conditions, implying that the travel agent example cannot be specified through this language. Instead, we present an example MSC specifying the interaction sequence between a client, server and adjacent databases. The client first submits a request to the server. Upon receiving this request, the server triggers two queries to its local databases, and waits for a response. When both (databases) comply with a response, a response is formulated by the server and sent to the client.

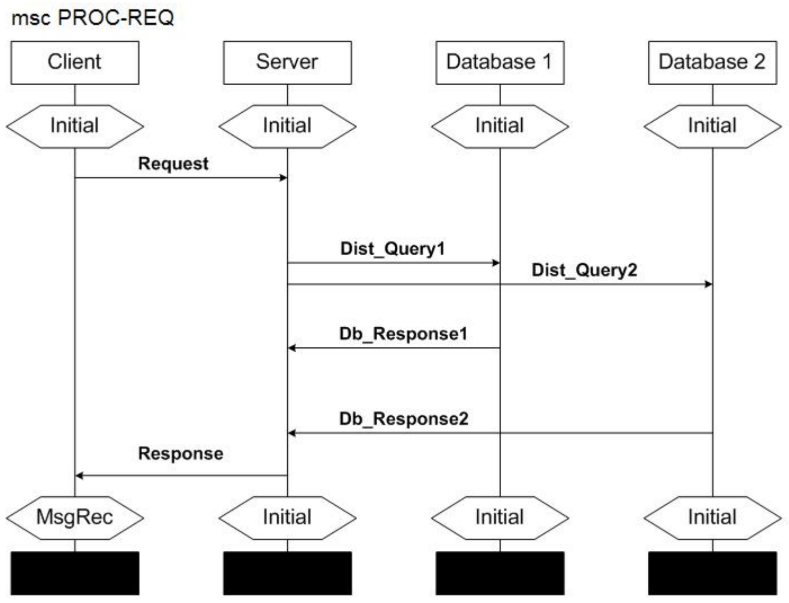


Figure 8.2: An example MSC.

Each vertical axis specified within the MSC represents an entity which has an active role within the monitored activity. This set of MSCs are related using HMSCs, as exemplified below. Suppose during the processing of a client request (as specified through MSC entitled PROC-REQ), the client also interfaces with a web service in order to request an optimal pricing strategy (whose behaviour is specified through another MSC named REQ-STRAT), with this behaviour looping indefinitely, the resulting HMSC combining the above two MSCs is defined as follows.

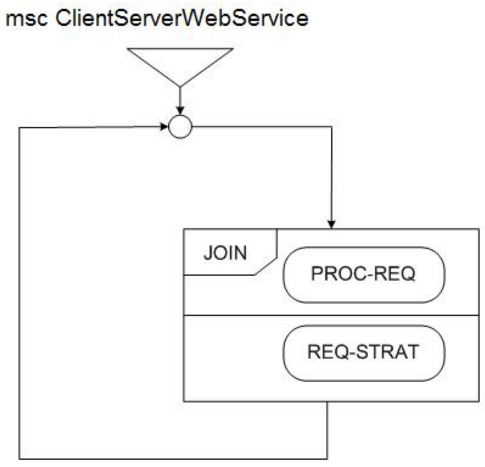


Figure 8.3: An example HMSC.

MSCs PROC-REQ and REQ-STRAT are combined using a join operator to specify that both behaviour traits are executed concurrently. Note that MSCs can synchronise on specific message calls, such as the message call **Response** specified in MSC PROC-REQ.

The high level specifications specified through MSCs and HMSCs are translated into executable state machines through a series of processes, generated through a sequence of *projection*, *normalisation*, *automaton synthesis* and *optimisation*. Projection involves mapping the set of MSCs to the appropriate locations within the system. Normalisation involves adding and altering certain structures in order to convert each MSC to an equivalent normal form. Automaton synthesis converts normalised MSCs to state machines, which finally go through a process of optimisation. Each state machine is defined through a set of states and transitions. Each transition entails a message name and direction (*i.e.,* whether sending or receiving). Note that the resulting state machines are always defined to be deterministic. Automatic property translation and instrumentation are supported, and is achieved by generating aspects [53, 54] which instrument the resulting state machines (translated from the MSC specifications) with the underlying systems. However reactions to violations are not, with the monitoring algorithm only reporting failure if properties are violated.

The resulting monitors are distributed *across systems*, with each message (transferred within the system) encapsulating (i) the message content, (ii) sender information, and (iii) the monitor state. Hence, this approach is choreography based, since monitor execution is spread throughout the system. Also, the approach is implemented for ESB 2.0 architectures, which does not yet support dynamic distributed architectures. Information confidentiality is not an issue when monitoring system integration, since monitored information is already available on the bus. Issues such as communication duration and delay, as well as difficulties with remote unsynchronised systems are not handled. Perhaps the most helpful insight gathered from this approach is the fact that choreography-based runtime monitoring of distributed system lends itself well to system-of-system implementations whose integration is handled by an ESB architecture. This is achieved by the architecture's inherent message oriented communication methodology between distributed systems, which not only enforces loose coupling between systems, but offers an intuitive event-based monitoring mechanism based on the interception and manipulation of messages.

DiAna

DiAna[80] is a runtime verification framework focused on monitoring safety properties of distributed systems. The authors argue that it is generally impractical to monitor system properties in a distributed setting specified through classical temporal logics, motivating the need for a monitoring framework based on a language which respects physical distribution. More specifically, temporal logics used during the runtime verification of monolithic architectures do not lend themselves well to monitor instrumentation across distributed locations, since they are usually based on the assumption of a readily available global state. To this effect, the presented approach proposes the use of *Past Time-Distributed Temporal Logic* (PT-DTL), a distributed

$$\begin{array}{lcll}
F & ::= & @_i F_i & \\
F_i & ::= & \text{true} \mid \text{false} \mid P(\vec{\xi_i}) \mid \neg F_i \mid F_i \text{ op } F_i & \text{propositional} \\
 & & \mid \odot F_i \mid \Diamond F_i \mid \boxdot F_i \mid F_i \,\mathcal{S}\, F_i & \text{temporal} \\
 & & \mid @_{\forall J} F_j \mid @_{\exists J} F_j & \text{epistemic} \\
\xi_i & ::= & c \mid v_i \mid f(\vec{\xi_i}) & \text{functional} \\
 & & \mid @_J \xi_j & \text{epistemic} \\
\vec{\xi_i} & ::= & (\xi_i, ..., \xi_i) &
\end{array}$$

Figure 8.4: PT-DTL syntax.

variant of the *Past Time-Linear Temporal Logic* (PT-LTL) [23] which however explicitly introduces the notion of location within its LTL expressions. This is mainly achieved through the introduction of *epistemic operators* i.e. operators which reason about knowledge (more below).

Similarly to mDPɪ, this framework considers distributed systems consisting of a set of subsystems, each admitting a local local state. Inter-system communication occurs through asynchronous message passing interactions (*i.e.,* no shared memory) of indeterminate duration. Moreover, the framework makes no assumptions on the preservation of message ordering. Note that PT-DTL is an event based logic, thus verifying system properties through the interception of system events. To this effect, the framework categorises events into three distinct categories; (i) *internal* events which update local system states, (ii) *send* events, occurring (on the sender's side) when a system transmits a message, and (iii) *receive* events, triggered on the receiver's side when a message is received.

The intuition behind PT-DTL is that each expression is *localised*, by binding each expression to a subsystem. However, each localised expression may also refer to expressions whose location is bound to remote remote subsystems, hence allowing expressions to refer to an illusion of the global state. The underlying monitoring algorithm is subsequently responsible for efficiently evaluating remote expressions in appropriate fashion, as shall be discussed below. Fig. 8.4 presents the PT-DTL syntax; its novel addition over PT-LTL involves the addition of $@_i$, an epistemic operator which allows expressions to refer to remote expression evaluations on the *last known state* of (remote) subsystem i.

A PT-DTL formula F is an expression over the global system constituting of (i) *localised expressions* and (ii) *localised formulas*. The former serve as atomic propositions upon which the latter are built on. A localised expression is represented by ξ_i, and consists of either constant c, variable v_i local to system i, or a tuple of expressions $\vec{\xi_i}$. Operator $f(\vec{\xi_i})$ represents a function (such as + or -) — returning a non-boolean result — over $\vec{\xi_i}$, whereas $P(\vec{\xi_i})$ represents a predicate (such as $\geq$). Localised formula F_i consists of either (i) a propositional formula (*op* represents logic operators), (ii) a predicate over expressions, (iii) past-time temporal operators, or (iv) epistemic operators. Temporal operators are analogous their PT-LTL equivalent, with $\odot F_i$ representing the value of F_i on the previous state, $\Diamond F_i$ stating that F_i was previously true, $\boxdot F_i$ stating that F_i was always true in the past, and finally $F_1 \mathcal{S} F_2$ being true if F_1 has been true since F_2 was true. Epistemic operators $@_{\forall J} F_j$ and $@_{\exists J} F_j$ return true if all (or at least one) subsystems $j \in J$ satisfy expression F_j. Operator $@_{\forall J} \xi_j$ admits an analogous interpretation, applied over expressions. Refer to [80], section 5.2 for a formal definition of the logic semantics. However crucially note that the specified semantics admit an efficient execution strategy, with the evaluation of each operator referring to (i) the current state and (ii) its evaluation on the previous state. The following are two example PT-DTL formulas based on those presented in [80].

- $@_1 \boxdot (a == @_2 b)$
- $@_{\{j:\text{Subsystem} \,\bullet\, j\}} \boxdot ((@_j \text{isInCriticalSection} \rightarrow$
  $@_{\{i:\text{Subsystem} \,\mid\, j \neq i \,\bullet i\}} (\neg \text{isInCriticalSection})) \,\mathcal{S}\, @_j \mathit{initialised})$

The first example specifies that at no time can variables a (located at subsystem 1) and b (located at subsystem 2) be unequal. On the other hand, the second example specifies that from the moment each subsystem is initialised, if any such subsystem is in its critical section, no other subsystem can be in its critical section. Also, note how both formulas are localised. The following example specifies the automated travel agent property, dictating that the airline and hotel booking subsystems do not place bookings which the client cannot afford

$$@_{bank} \boxdot (balance > (@_{airlineBookingAgency} flightCost + @_{hotelBookingAgency} hotelCost)$$

Note that we arbitrarily assign the localised expression to the bank subsystem, and use remote expressions which return the cost of the flight and hotel at the airline and hotel booking agencies respectively.

A PT-DTL formula is represented by its set of localised expressions, which are compiled prior to execution into a set of local monitors, retaining this configuration throughout execution. This implies that the framework is a static choreography-based approach. Resulting monitors subsequently evaluate operations locally, and also refer to the evaluation of remote expressions. The execution of these monitors has been developed with two principles in mind; (i) Local monitors should be computationally fast, and consume little memory overhead, and (ii) The number of additional messages with regards to monitoring communication should be minimal. Moreover, the monitoring algorithm adopts the *knowledge vector* construct, a conceptual tool which (i) extracts a causal ordering on remote events from underlying system interactions, and (ii) disseminates local expression valuations to remote monitors. More specifically, each subsystem keeps a local copy of a knowledge vector, which contains evaluations of all remote expressions evaluated within the system.

Each knowledge vector KV entails a vector of entries, with $KV[j]$ representing the entry for subsystem j. Each $KV[j]$ contains (i) the sequence number, representing the last event seen at subsystem j (referred to by $KV[j].seq$), and (ii) a set of values $KV[j].values$, containing the set of formula and expression evaluations localised to subsystem j. The mechanism admits a simple update strategy which transfers and updates knowledge vectors throughout the system. Upon the observation of an internal event, all local formulas and expressions (local to that system where the event has occurred) are re-evaluated, according to the new local state and the local knowledge vector. If a send event occurs, the local knowledge vector sequence number is incremented, the local knowledge vector is tagged to the outgoing message and sent with the message payload. Upon detecting a receive event, the monitor at the receiving end extracts the knowledge vector and updates knowledge of its remote expression evaluations. Said evaluation is carried out for each $KV[j]$, such that if the $KV[j].seq$ entry of the received knowledge vector is larger than the local $KV[j].seq$, then the whole $KV[j]$ is updated locally to the received entry. Once the local knowledge vector is updated, all local formulas and expressions are re-evaluated in order to determine whether the knowledge vector update invalidates any local formulas.

The above algorithm is called the *knowledge vector algorithm*, and ensures that local formula execution is based on the latest *locally known evaluation* of remote formulas and expressions. Stale information (identified through smaller $KV[j].seq$ values) are immediately discarded, ensuring that only the latest information is considered. The knowledge vector is based on the notion of vector clocks, and hence extracts a happened-before relationship on remote events (section 3.5). It is for this reason that the described m onitoring algorithm is adept at verifying causal properties. Note that no additional messages for the purposes of monitoring are required, with an additional bandwidth overhead instead incurred over each message sent by the underlying system. The authors claim that the monitoring algorithm's space and time complexity for processing each recognised event e is $\Theta(mn)$, where m is the size of the local formula and n is the number of subsystems constituting the distributed system [80]. Moreover, the authors also claim that the knowledge vector size is in general linear to n (number of subsystems constituting the distributed system), and can be independent of n in particular cases such as when all sub-formulas containing epistemic operators are of the form $@_j F_j$ or $@_j \xi_j$ (*i.e.*, all sub-formulas do not refer to remote expressions).

DiAna is a Java implementation of the choreography-based monitoring framework defined above. The tool adheres to the Actor framework [2], with subsystems represented through communicating actors. Each actor is represented by (i) a unique name, (ii) an internal state, (iii) an execution thread, and (iii) a set of procedures which can alter the internal state. Moreover, actors communicate by asynchronous message passing with indefinite communication duration, which mirrors distributed systems considered by mDPi. Hence, a distributed system is represented within through a set of actors (one for each subsystem). Finally, note that DiAna supports property violation reactions, since it allows for the association of Java code with each formula, eventually executed upon formula violation. Moreover, the framework also supports automated instrumentation, since all formulas are converted to executable Java code automatically instrumented (at the bytecode level) with the underlying system. The resulting code monitors for an update of a select number of variables (representing an internal event), as well as the sending or receiving of a message.

The above approach is a valid attempt to distributed monitoring; by localising the monitoring effort it achieves an efficient approach to the verification of causal properties in a distributed setting. As previously stated, DiAnatakes a statically choreographed approach. Although dynamic architectures are not explicitly mentioned, it is clear that the framework is not capable of supporting such configurations, reason being that vector clocks (and by extension, knowledge vectors) are not tolerant to dynamic environments. Conversely, in order for the knowledge vector algorithm to be effective, complete knowledge of the system's contributing nodes must be available a priori. Moreover, by employing knowledge vectors which export local event knowledge to remote locations implies the possibility for data exposure.

8.3 Summary

The following section summarises the presented frameworks, and also presents an a posteriori comparison to mDPi. This summary is given in tabulated form (following the above order), thus allowing for direct comparison on specific issues. Note that to our knowledge, some solutions fail to discuss certain issues we previously identified as pertinent to the design of a distributed monitoring frameworks, which is why certain tools are incomparable on some criteria. Moreover, these tools vary on numerous issues, including (i) the *scope* of system implementations the framework has been designed for (for example, some are tailored for web service compositions only, others for ESB architectures), as well as (ii) framework implementation details (including the choice of language, abstraction level *etc.*). We shall hence compare approaches at a conceptual level, using criteria identified below.

Synchronous vs Asynchronous — Specifies whether the monitoring framework verifies properties on the fly during system execution, or on a pre-recorded trace.

Reaction to Violations — Identifies whether the framework allows for the execution of some violation reaction upon detection of a property violation.

Conditions — Refers to the capability of filtering events based on event parameters extracted during their detection.

Numerical Queries — Refers to the explicit support for the specification of numerical queries for the evaluation of statistical properties.

Instrumentation Approach — Classifies the presented framework under one of the instrumentation strategies presented in section 3.4 *i.e.*, static/dynamic orchestration, static/dynamic choreography, or supporting multiple strategies.

Communication Model — Specifies whether the framework is defined for a system employing either message passing or shared memory communication.

Locality — Specifies whether the framework acknowledges the issue of information confidentiality, while also presenting solutions for the monitoring of system behaviour while avoiding exposure.

System Topology — States whether the framework only supports distributed systems admitting static configurations, or whether it also supports dynamic configurations which evolve during execution.

Figure 8.5: A comparison of distributed monitoring frameworks.

Framework	DMaC	REM	GEM	RTML	SoS Integr.	DiAna	mDPi
Synch. vs. Asynch.	Synch.	Synch.	Synch.	Synch.	Synch.	Synch.	Asynch.
Violation Reaction	×	✓	✓	×	×	✓	×
Conditions	✓	✓	✓	×	×	✓	✓
Num. Queries	✓	×	×	✓	×	×	×
Instr. Approach	S. Chor.	S. Chor	S. Chor[a]	S. Orch.	S.Chor	S. Chor	Gen.
Comm. Model	M.Pass	M.Pass[b]	M. Pass.[c]	N/A	M. Pass.[d]	M. Pass.[e]	M.Pass
Locality	×	×	×	N/A	N/A	×	✓
Syst. Topology	Static	Static	N/A	N/A	Static	Static	Dynamic

[a]Although a static approach, partially supports dynamic properties
[b]Recognises possibility of unreliable communication medium.
[c]Admits issues with communication channels involving order changing communication with unbounded delays.
[d]Handled by the ESB middleware.
[e]Assumes asynchronous, order-changing communication of indeterminate duration.

Some interesting trends emerge from the table above. Firstly, although all approaches readily fit into our broad taxonomy (section 3.4), no current approach has yet identified the enhanced complexities with monitoring dynamic architectures. In other words, current approaches all assume that the underlying system remains unchanged throughout execution. mDPi is distinguished in this sense, in that it offers necessary tools for the monitoring of evolving configurations. Tt is also worth notion that one approach partially supports dynamic properties. This is achieved in similar fashion to mDPi in that it adopts an interpreted language whose rules can be loaded at runtime.

By extension, to the best of our knowledge, the migrating monitor approach seems to be novel. In other words, current tools do not employ monitors with additional capabilities of moving across the system on the fly. However, while promising, it remains to be seen how feasible this approach is in industrial settings, in that some may reject the flexibility of this proposed instrumentation strategy (for fears of security). On the other hand, no current approach identifies potential security risks with employing privileged monitors in a distributed setting either.

The issue of information confidentiality has also been neglected so far. In other words, current approaches do not recognise the hazards with employing monitors which import local information available at remote subsystems. As a result, current approaches potentially result in the exposure of information. Note that although mDPi recognises this issue, it is still possible to define mDPi monitors which expose information (such as in the case of implementing an orchestrated approach). However, the framework also offers necessary support so as to avoid unnecessary exposure. Clearly, the possibility avoiding exposure also depends on the property at hand; for instance an invariance property across locations unavoidably leads to some form of exposure. On the other hand, properties of sequentiality are amenable to monitoring while avoiding transfer of local information.

Some commonalities; all current approaches define monitoring algorithms for distributed systems communicating through message passing techniques. Moreover, most identify issues of an unreliable communication medium to various degrees. More specifically, all accept that interactions take some indeterminate duration. Whereas some approaches (including [64]) get around this problem by assuming the availability of a global clock (which is unrealistic), others attempt more sophisticated techniques by extracting a causal ordering on events through mechanisms analogous to Vector Clocks (see [80]). In this respect, our approach is perhaps closest to the latter attempt, in that we adopt logical clocks (*i.e.,* the monitor counter) for extracting a (partial) temporal ordering on events. However, we recognise the possibility of adopting more powerful mechanisms in the future.

Other distributed monitoring approaches allow for violation reactions, implying the need to study its applicability in mDPi. Current approaches allow an unrestricted form of reactions, in that the execution of some reaction at one subsystem can potentially affect other subsystems too. However, we recognise the risk with this approach due to security issues mentioned in chapter 7; what if the system admits malicious (or competing) subsystems, who purposefully execute code in order to damage adjacent systems? Perhaps a more controlled form of violation reactions is necessary, and is left as future work. Finally, the addition of numerical queries can also be investigated in the future, allowing for the verification of non-functional requirements in a distributed setting. This may be particularly interesting for instance in order to verify network throughput, and/or failure rates within a network topology.

8.4 Conclusions

Numerous alternate approaches to the monitoring of distributed has been studied in the literature, with various degrees of success. In truth, current approaches are more focused on the choice of appropriate specification languages, and the efficiency of their associated monitoring algorithms. Although necessary, as a result other pertinent issues have been ignored throughout, including the enhanced complexities with monitoring dynamic architectures, as well as monitoring in the face of confidential information. To this effect, mDPi might offer an appropriate solution in such circumstances, especially due to its introduction of migrating monitors and associated proof of well-behaved semantics. This is not to say that mDPi is unsuitable for more traditional scenarios. In fact, the language describes an alternate approach, by instead offering a *general monitoring framework* through which various instrumentation strategies can be defined, as opposed to forcing a particular approach from the offset. On the other hand, a more fair comparison would require a concrete implementation of the framework defined in chapter 7, in order to (i) compare the framework's applicability *wrt.* its applicability to industrial applications, and (ii) investigate overheads associated with executing the framework. This second point is especially pertinent, since it is an issue we have not tackled throughout this dissertation, and would allow for a direct comparison with other tools *wrt.* associated algorithm overheads.

Part IV

Conclusions

9. Conclusions

This chapter first presents an a posteriori summary of the work presented throughout this thesis, and is followed by future avenues worth exploring as part of the future direction of this research. The final section concludes this work by sharing some final thoughts.

9.1 Summary

Whether by choice or necessity, more and more distributed architectures are being adopted as solutions to both business and computational problems. This shift is encouraged further by the push for service oriented and component-based implementations, as well as the ever increasingly ubiquitous presence of the internet. However, system distribution also implies (i) an additional degree of complexity, as well as (ii) an adverse effect on dependability, due to potential issues with performance when operating across unreliable communication mediums. This implies a corresponding need for software verification techniques tailored for distributed systems.

To this effect, this thesis focused on the study of runtime verification techniques applied to distributed settings. Runtime verification is a lightweight software verification technique concerned with checking whether system behaviour exhibited at runtime adheres to a set of formalised requirements. Necessary verification is performed by a monitor, which analyses the system's trace representing exhibited runtime behaviour. However, although this approach has achieved appreciable success on standalone architectures, system distribution poses a major challenge to runtime monitoring. Firstly, it is not immediately clear whether it is best to centralise the monitoring effort, or whether to distribute monitors accordingly. Although the first approach is computationally simpler, and may be adept in certain cases, it also runs the risk of consuming excessive bandwidth, as well as issues with exposure of sensitive information. On the other hand, although more flexible, distributing the monitoring effort is a more complex task, especially when considering systems whose topology may evolve at runtime. We hence presented a broad taxonomy of approaches to distributed monitoring, highlighting scenarios where each approach fits best.

This thesis' main contribution lies with the development of mDPɪ, a location-aware calculus with explicit notions of monitoring. mDPɪ has been designed with practical considerations in mind, including the difficulty of total event orderings in distributed settings, as well as recognising differences between local and remote interactions. This calculus enabled the formalisation of the distributed monitoring scenario, which in turn allowed us to model different monitor instrumentation strategies. Through the definition of an extensible LTS semantics we were also able to consider system behaviour at various levels of abstraction. This allowed for the selective reasoning about a number of pertinent issues, including the locality of communication and trace analysis — which potentially represents data exposure — as well as distinguishing between monitor and system computation (which, in turn, allows us to focus on either as necessary). Moreover, through the use of the bisimilarity relation applied over different (filtered) LTSs we were able to formalise three statements which we believe distill and identify the core aspects of our approach. We provided a proof for the first statement, thus showing that mDPɪ monitors do not affect process computation (at a conceptual level), and formalised the remaining two questions, whose proof is left as future work. In truth, although we believe mDPɪ to be a novel approach to formalising distributed monitoring through the use of a process calculus, it is best considered as an initial study, with numerous avenues left for future analysis. One limitation we immediately recognise with our current approach is the loss of completeness *wrt.* the monitoring of properties, due to the lack of synchrony amongst remote locations. However, this loss is, to a certain extent, beyond our control, since it is attributed to an underlying system characteristic. It can be considered an achievement of our work that the calculus precisely formalises the problem at hand, which in turn allows us to study possible approaches to the problem. This immediately highlights a possible area of future work, involving the extension of monitor counters to more powerful mechanisms for extracting temporal orderings across remote events. Other issues which may benefit from future study include the application of mDPɪ when faced with mobile systems (*i.e.*, systems whose components migrate as well; formalising mobile computational devices), as well as a study of distributed monitoring in face of failure (more below). Other potential issues worth investigating include the extension of our approach from a non-intrusive to an intrusive form of monitoring, allowing for (a controlled form of) reparations in order to steer the system back to an acceptable state.

We also introduced the migrating monitor approach, employing monitors which verify locally, and physically migrate to remote locations when their behaviour becomes pertinent to the system's overall correctness *wrt.* property being verified. The calculus also succinctly formulated this approach, by adopting migration as a language primitive. This formalisation highlighted the potential of migrating monitors; we can adopt monitors whose next choice of monitoring location depends on information learnt at runtime. This in effect describes a monitoring algorithm which is tolerant to dynamic topologies, thus being able to keep up with the system's unpredictable changes. We also saw how migrating monitors offer potential solutions to monitoring systems admitting sensitive information, by localising the monitoring effort. Finally, migrating monitors also introduced the potential for dynamic properties *i.e.,* properties which can only be fully characterised at runtime, or even new properties learnt during system execution. However, adopting migrating monitors can result in a heavy handed mechanism, and is best used when necessary. In short, migrating monitors is an approach which is best applied in face of system dynamicity as well as the presence of confidential information. To the best of our knowledge, this technique is novel, in that no known related approach to distributed monitoring motivates the use of mobile monitors. Moreover, it is worth noting that current approaches to distributed monitoring fail to recognise the difficulty with monitoring dynamic architectures, or the issue of distributed verification in the face of confidential information. Although we believe migrating monitors to be a promising attempt to distributed monitoring, much more research is required in order to ascertain its validity. One major worry thus far is that questions remain about whether systems are willing to trust the flexibility associated to these monitors, especially due to perceived security risks.

We have also shown how the calculus can be used to encode different monitoring approaches for specifications written through regular expressions, enabling their comparison. The motivation behind the choice of regular expressions as an initial language is its simplicity; however the focus of this task firmly lies with the comparison of different monitoring strategies. In turn, this points to the need for studying the conversion of more expressive languages to mDPı monitors, in order to study the expressiveness, as well as the limitations of our framework. Finally, we presented a proof-of-concept case study implemented in Erlang, and whose purpose was the identification of practical issues which arise during implementation of the calculus. Numerous issues were identified, ranging from potential security risks associated with transferring monitors across unsafe mediums (not to mention exposing internal system operation across locations), as well as difficulties with achieving an implementable framework across dynamic and/or heterogenous environments. To this effect, we proposed the use of the monitor manager, an additional component installed at each location which serves as an instrumenter (extracting a local trace per location), as well as interpreted mDPı monitor expressions. By detaching the monitoring and tracing effort, we were also able to achieve the property agnostic approach motivated by migrating monitors. Nevertheless, considerable work is envisaged *wrt.* the application of the motivated framework to real-life scenarios.

In conclusion, although the ideas presented in this thesis propose promising new developments *wrt.* the application of runtime monitoring on distributed settings, further study is required on proposed innovations in order to ascertain their viability both at a theoretical and practical level.

9.2 Future Work

The following section discusses avenues for future work, and emerge both from current framework limitations, as well as being a result of certain potentially fruitful distributed monitoring aspects which we have not been tackled throughout this thesis.

9.2.1 Continuation of Theoretical Development

Our first thought goes to the proof of the remaining two statements formalised in section 5.4, thus proving that, in a sense, global monitoring can monitor the same properties verified in localised fashion, and that a properly defined migrating monitor approach preserves locality by avoiding data exposure. Although we intuitively believe both statements to be possibly true, we aim to be convinced of their veracity. Moreover, obtaining such proofs goes a long way into justifying migrating monitors, and is hence something we strive for. We also recognise the need to study theoretical properties of the mDPi calculus further. More specifically, we particularly recognise the need for justifying the bisimilarity relation within the calculus by proving its contextuality *wrt.* the formally defined language semantics.

Another extension to the calculus worth studying involves a possible enhancement of the current rudimentary logical clocks adopted by mDPi monitors. Currently, mDPi monitors employ a single counter in order to extract a temporal ordering amongst remote locations, operating analogously to Lamport Timestamps [58], however adapted to extract a monitored-before relationship amongst remote events (section 3.5). However, given the success of Vector Clocks as a more powerful mechanism as opposed to Lamport timestamps [35], it may be worth studying the application of these constructs within our language. Moreover, we believe mDPi to be easily extendible *wrt.* the adoption of vector clocks, replacing the current counter with a list. Hence, augmented mDPi monitors can be formalised as follows

$$\{\![M]\!\}^{\Delta}$$

where $\phi \in \Delta :: \text{Locs} \to \mathbb{N}$ represents a *monitor vector i.e.,* a mapping from locations to natural numbers, and represents the last counter value known by that monitor *per location*. Updated rules for the execution of operators go and setC, as well as trace input hence take the following form

$$\textsc{Go'}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{go}\; l.M]\!\}^{\phi} \xrightarrow{\tau_{\langle m:k,l\rangle}} \delta \triangleright l\{\![M]\!\}^{\phi \oplus \{l,\, \delta(l)\}}}$$

$$\textsc{SetC'}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{setC}(l).M]\!\}^{n} \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright k\{\![M]\!\}^{\phi \oplus \{l\, \delta(l)\}}}$$

$$\textsc{MonTr'}_i \; \frac{}{\delta \triangleright k\{\![\mathbf{m}(c, \bar{x}, l).M]\!\}^{\phi} \xrightarrow{\mathrm{m}(c,\bar{d},l,\phi(l))_{\langle m:k\rangle}} \delta \triangleright k\{\![M\{{}^{\bar{d}}/_{\bar{x}}\}]\!\}^{inc(\phi,l)}}$$

Figure 9.1: Updated rules for handling the monitor vector.

Such that the monitor vector ϕ is updated on migration/re-alignment, updating the assigned value for location l to the current value mapped by the counter state. Rule MonTr'$_i$ additionally describes the handling of the monitor vector during trace input; when a monitor (at location k) reads the next trace element from location l whose position (in the local trace) is dictated by its monitor vector to be $\phi(l)$, the corresponding value recorded within the vector for location l is incremented accordingly through the use of *inc*. Note that the action is also labelled accordingly. One would also possibly imagine the addition of new rules, specifying necessary machinery for migrating and/or interacting monitors to exchange local instances of vector clocks in order to update each other's extracted temporal orderings, thus achieving a collective effort by executing monitors *wrt.* the extraction of temporal orderings, similarly to the algorithm presented in [35]. Note that since Vector Clocks simply entail a list of counter values used for monitoring purposes, exchanging vector clocks amongst locations does not denote data exposure. Nevertheless, it remains to be seen what Vector Clocks can give us *wrt.* an enhanced mechanism for the extraction of temporal orderings, and is left as future work.

Finally, other additions we are looking to study further include (i) the addition of reparation for violation reactions, as well as (ii) the introduction of (a limited form of) real time operators. One could for instance look at the monitoring of a class of properties characterised by a locally synchronous, globally asynchronous specification. In such cases, the adoption of timers on a per location basis becomes a possibility.

9.2.2 Conversion of Specification Languages

Although the conversion of regular expressions served as a valid first approach to the conversion of specification languages to mDPi monitors, the study of other, more expressive languages is required. In turn, the use of these languages serves as a tool which allows us to better understand the capabilities, as well as limitations of the presented framework. Two languages which immediately spring to our attention include PT-DTL [80] and (a subset of) Larva [25]. The former is attractive to our setting since it represents a distributed extension of Linear Temporal Logic [23], which is one of the most studied logics applied in a runtime verification setting. Its introduction of epistemic operators which localise expressions fits perfectly with our notion of located monitors. Moreover, although the logic has so far been used in a static setting, its use of quantifiers over locations can be easily interpreted over dynamic configurations.

Being automaton based, Larva admits a straightforward conversion to a process calculus such as mDPi[67]. Moreover, its introduction of communicating monitors over channels readily fits with our definition of mDPimonitors, and can also be used as a tool for implementing a form of synchronisation across localised Larva scripts. Moreover, the current understanding of contextual properties can also be interpreted over dynamic configurations, in that an instance of the monitor is spawned and migrated to each node which is added to the system at runtime. On the other hand, one feature which has to be severely restricted involves the use of timers. The problem with implementing this feature is the lack of synchrony amongst remote clocks. In this case, we are faced with a design choice; we can either disallow timers completely, or at most permit their use on a per location basis.

9.2.3 Framework Extensions

We also plan to extend our study of monitoring distributed architectures beyond the the setting defined in section 3.2. Our first interest goes to the study of monitoring under more sever failure scenarios. More specifically, although we currently recognise that the communication medium can be unreliable, and that system architectures are possibly dynamic (*i.e.*, their nodes come and go at runtime), we do not tackle the issue *wrt.* failure of interactions and/or nodes. For instance, what happens if a monitor migrates to some location, which suddenly fails, taking down this monitor in the process? Does this imply that the global monitoring effort is invalidated as a result? Analogously, we currently expect interactions to take an indeterminate by finite duration. What happens if some node the monitor wishes to interact with/migrate to is currently unavailable? Does this mean that the whole monitoring effort is halted as a result? Perhaps in such a scenario it may be possible to suspend monitoring of this particular location and continue with the remainder of the monitoring task, in order to ensure progress. In general such scenarios imply the need for studying *fault tolerant monitoring i.e.*, monitoring in the face of failure. As discussed in [7], fault tolerance, encapsulation and redundancy go hand in hand. It is for this reason that the migrating monitor approach may be amenable to a fault tolerant monitoring approach, since we can easily replicate migrating monitors on the fly without the need for system restart. Another, analogous application of the migrating monitor approach involves the study of load balancing from a monitoring perspective; migrating monitors located at heavily stressed nodes to other locations which can afford some overhead for monitoring purposes.

Another area worth further investigation involves the monitoring of mobile systems. In fact, although mDPI allows for the study of architectures whose communication explicitly evolve, the addition of new locations is at most an implicit concept thus far. However, extending the underlying processes' mobility, including the ability to explicitly create new locations at runtime, as well as the migration of processes (as seen in [47]) presents interesting new challenges from a monitoring perspective. With the arrival of mobile computing and agent technology, such systems represent a whole class of innovative distributed architectures, and best of all can highlight further the need for migrating monitors. In such scenarios it may for instance be interesting to define monitors which follow around processes in order to verify their mobile behaviour. Moreover, this extension highlights further the difficulties with verifying dynamic architectures; how can a static approach monitor a system which creates a new location at runtime? In short, mDPI opens up a whole variety of alternate scenarios worth exploring, and is something we intend to do as future work.

9.2.4 Application to Industrial Settings

Work presented throughout this thesis is carried out at a conceptual level, considering potential problems and offering a calculus which formalises theoretical solutions to monitoring distributed settings. The next logical step involves applying the framework to more practical settings, preferably on real-life industrial systems in order for the framework to reach an elevated level of maturity. By extension, it is crucial to study the applicability of the novel migrating monitor approach in real life scenarios. In truth, although this proposed approach is a promising development to the monitoring of distributed systems, further study of its applicability is required.Clearly, with the myriad of distributed system implementations available today, even verifying properties of sequentiality in a safe and confidentiality aware manner represents a step forward in proving applicability of the calculus. On the other hand, applying the framework on real-life case studies requires a more concrete implementation of the framework, preferably implementing the mechanisms defined in chapter 7.

9.3 Concluding Thoughts

Applying runtime verification techniques to distributed settings enhances the complexity of an already non-trivial field. More specifically, the effectiveness of a chosen monitoring approach is more than ever subject to underlying system characteristics, including but not limited to issues regarding asynchrony amongst its subsystems, the possibility of dynamic architectures as well as the presence of confidential information. To this effect, we proposed mDPi, an initial formal study of the scenario whose purpose is to distill and identify the core aspects of distributed monitoring. We also presented the promising migrating monitor approach, which to the best of our knowledge is the first approach to explicitly support dynamic topologies. Nevertheless, the field is vast and requires substantial more research. We are hopeful that this study serves as a springboard for further investigations, including a study of the framework's applicability in industrial settings, an exploration of current semantics' limits and possible extensions, as well as the conversion of more specification languages into mDPi monitors.

A. mDPi Syntax and Semantics

Syntax

$$
\begin{array}{rcl}
S, V & ::= & k[\![P]\!] \mid S \parallel V \mid \mathsf{new}\, c.S \\
P, Q & ::= & u!\bar{v}.P \mid u?\bar{x}.P \mid \mathsf{new}\, c.P \mid \mathsf{if}\ u{=}v\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P \| Q \mid *P \mid \mathsf{stop} \mid \\
 & & \{M\}^n \mid T \\
T & ::= & \mathbf{t}(c, \bar{d}, n) \\
M, N & ::= & \mathsf{go}\ u.M \mid u!\bar{v}.M \mid u?\bar{x}.M \mid \mathsf{new}\, c.M \mid \mathsf{if}\ u{=}v\ \mathsf{then}\ M\ \mathsf{else}\ N \mid M \| N \mid *M \mid \\
 & & \mathsf{setC}(u).M \mid \mathsf{ok} \mid \mathsf{fail} \mid \mathbf{m}(c, \bar{x}, k).M
\end{array}
$$

Semantics

$$
\textsc{Out}_p\ \frac{}{\delta \triangleright k[\![c!\bar{d}.P]\!] \xrightarrow{c!\bar{d}_{\langle p:k\rangle}} \mathsf{inc}(\delta, k) \triangleright k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{d}, \delta(k))]\!]}
$$

$$
\textsc{In}_p\ \frac{}{\delta \triangleright k[\![c?\bar{x}.P]\!] \xrightarrow{c?\bar{d}_{\langle p:k\rangle}} \delta \triangleright k[\![P\{^{\bar{d}}/_{\bar{x}}\}]\!]}
\qquad
\textsc{Out}_m\ \frac{}{\delta \triangleright k\{\![c!\bar{d}.M]\!\}^n \xrightarrow{c!\bar{d}_{\langle m:k\rangle}} \delta \triangleright k\{\![M]\!\}^n}
$$

$$
\textsc{In}_m\ \frac{}{\delta \triangleright k\{\![c?\bar{x}.M]\!\}^n \xrightarrow{c?\bar{d}_{\langle m:k\rangle}} \delta \triangleright k\{\![M\{^{\bar{d}}/_{\bar{x}}\}]\!\}^n}
$$

$$
\textsc{Com}_1\ \frac{\delta \triangleright S \xrightarrow{(\bar{b})c!\bar{d}_{\langle \mu:k\rangle}} \delta' \triangleright S' \qquad \delta \triangleright V \xrightarrow{c?\bar{d}_{\langle \mu:l\rangle}} \delta \triangleright V'}{\delta \triangleright S \parallel V \xrightarrow{\tau_{\langle \mu:k,l\rangle}} \delta' \triangleright \mathsf{new}\, \bar{b}.(S' \parallel V')}\ [\bar{b} \cap \mathsf{FN}(V) = \emptyset]
$$

$$
\textsc{Com}_2\ \frac{\delta \triangleright S \xrightarrow{(\bar{b})c!\bar{d}_{\langle \mu:k\rangle}} \delta' \triangleright S' \qquad \delta \triangleright V \xrightarrow{c?\bar{d}_{\langle \mu:l\rangle}} \delta \triangleright V'}{\delta \triangleright V \parallel S \xrightarrow{\tau_{\langle \mu:k,l\rangle}} \delta' \triangleright \mathsf{new}\, \bar{b}.(V' \parallel S')}\ [\bar{b} \cap \mathsf{FN}(V) = \emptyset]
$$

$$\textsc{Trace}_e \; \frac{}{\delta \triangleright k[\![\mathbf{t}(c,\bar{d},n)]\!] \xrightarrow{\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta \triangleright k[\![\mathbf{t}(c,\bar{d},n)]\!]}$$

$$\textsc{MonTr}_i \; \frac{}{\delta \triangleright k\{\![\mathbf{m}(c,\bar{x},l).M]\!\}^n \xrightarrow{\mathsf{m}(c,\bar{d},l,n)_{\langle m:k\rangle}} \delta \triangleright k\{\![M\{^{\bar{d}}/_{\bar{x}}\}]\!\}^{n+1}}$$

$$\textsc{Mon}_1 \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta \triangleright S \qquad \delta \triangleright V \xrightarrow{\mathsf{m}(c,\bar{d},k,n)_{\langle m:l\rangle}} \delta \triangleright V'}{\delta \triangleright S \parallel V \xrightarrow{\tau_{\langle t:k,l\rangle}} \delta \triangleright \mathsf{new}\,\bar{b}.(S \parallel V')} \; [\bar{b} \cap \mathsf{FN}(V) = \emptyset]$$

$$\textsc{Mon}_2 \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta \triangleright S \qquad \delta \triangleright V \xrightarrow{\mathsf{m}(c,\bar{d},k,n)_{\langle m:l\rangle}} \delta \triangleright V'}{\delta \triangleright V \parallel S \xrightarrow{\tau_{\langle t:k,l\rangle}} \delta \triangleright \mathsf{new}\,\bar{b}.(V' \parallel S)} \; [\bar{b} \cap \mathsf{FN}(V) = \emptyset]$$

$$\textsc{Go}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{go}\ l.M]\!\}^n \xrightarrow{\tau_{\langle m:k,l\rangle}} \delta \triangleright l\{\![M]\!\}^{\delta(l)}} \qquad \textsc{SetC}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{setC}(l).M]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright k\{\![M]\!\}^{\delta(l)}}$$

$$\textsc{Inc}_m \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})\mathsf{t}(c_1,\bar{d},n)_{\langle t:l\rangle}} \delta \triangleright S \qquad \delta \triangleright k\{\![M]\!\}^n \xrightarrow{\mathsf{m}(c_2,\bar{e},l,n)_{\langle m:k\rangle}} \delta \triangleright k\{\![M']\!\}^{n+1}}{\delta \triangleright S \parallel k\{\![M]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright S \parallel k\{\![M]\!\}^{n+1}} \; [c_1 \neq c_2]$$

$$\textsc{Rec}_p \; \frac{}{\delta \triangleright k[\![*P]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright k[\![P \parallel *P]\!]} \qquad \textsc{Rec}_m \; \frac{}{\delta \triangleright k\{\![*M]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright k\{\![M \parallel \mathsf{setC}(k).\,*M]\!\}^n}$$

$$\mathrm{EQ}_p \; \frac{}{\delta \triangleright k[\![\mathsf{if}\ u=v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright k[\![P]\!]} \; [u=v]$$

$$\mathrm{NEQ}_p \; \frac{}{\delta \triangleright k[\![\mathsf{if}\ u=v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright k[\![Q]\!]} \; [u \neq v]$$

$$\mathrm{EQ}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u=v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright k\{\![M]\!\}^n} \; [u=v]$$

$$\mathrm{NEQ}_m \; \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u=v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \triangleright k\{\![N]\!\}^n} \; [u \neq v]$$

$$\textsc{Open}_s \; \frac{\delta \triangleright S \xrightarrow{(\bar{b})c!\bar{d}_{\langle \mu:k\rangle}} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\,b.S \xrightarrow{(b,\bar{b})c!\bar{d}_{\langle \mu:k\rangle}} \delta' \triangleright S'} \; [b \in \bar{d}]$$

$$\textsc{Open}_t\ \frac{\delta \rhd S \xrightarrow{(\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta' \rhd S'}{\delta \rhd \mathsf{new}\, b.S \xrightarrow{(b,\bar{b})\mathsf{t}(c,\bar{d},n)_{\langle t:k\rangle}} \delta' \rhd S'}\ [b \in \bar{d}]$$

$$\textsc{Split}_p\ \frac{}{\delta \rhd k[\![P \parallel Q]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \rhd k[\![P]\!] \parallel k[\![Q]\!]}$$

$$\textsc{Split}_m\ \frac{}{\delta \rhd k\{\![M \parallel N]\!\}^n \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \rhd k\{\![M]\!\}^n \parallel k\{\![N]\!\}^n}$$

$$\textsc{Exp}_p\ \frac{}{\delta \rhd k[\![\mathsf{new}\, c.P]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \rhd \mathsf{new}\, c.k[\![P]\!]}$$

$$\textsc{Exp}_m\ \frac{}{\delta \rhd k\{\![\mathsf{new}\, c.M]\!\}^n \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \rhd \mathsf{new}\, c.k\{\![M]\!\}^n}$$

$$\textsc{Cntx}_1\ \frac{\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'}{\delta \rhd \mathsf{new}\, b.S \xrightarrow{\alpha} \delta' \rhd \mathsf{new}\, b.S'}\ [b \notin \mathsf{FN}(\alpha)]$$

$$\textsc{Cntx}_2\ \frac{\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'}{\delta \rhd S \parallel V \xrightarrow{\alpha} \delta' \rhd S' \parallel V}\ [\mathsf{BN}(\alpha) \cap \mathsf{FN}(V) = \emptyset]$$

$$\textsc{Cntx}_3\ \frac{\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'}{\delta \rhd V \parallel S \xrightarrow{\alpha} \delta' \rhd V \parallel S'}\ [\mathsf{BN}(\alpha) \cap \mathsf{FN}(V) = \emptyset]$$

B. Substitution in mDPi

Substitution is defined by using overloaded notation $v\sigma$ on identifiers, which returns v when $v \notin dom(\sigma)$, and $\sigma(v)$ otherwise. In other words, we elevate σ to a total function of the form $id \oplus \sigma$ (id is the identity relation over variables). This notation is lifted over lists of the form $\bar{v}\sigma$, hence applying $v\sigma$ for all $v_i \in \bar{v}$. $S\sigma$ is defined over two stratified levels *i.e.*, at the system and process levels. We first define substitution over systems (*Def$^{\underline{n}}$* B.0.1), which subsequently pushes substitution to its process constituents (*Def$^{\underline{n}}$* B.0.2). Operator $S \trianglelefteq R$ denotes *domain co-restriction* [82] of relation R to elements not in set S *i.e.*, if $R :: X \leftrightarrow Y$ then $S \trianglelefteq R \triangleq \{x : X, y : Y \mid x \notin S \ \wedge \ (x, y) \in R \bullet (x, y)\}$.

Definition B.0.1. *(Substitution $S\sigma$) We define substitution of σ on $S \in$ Sys, written $S\sigma$, as follows*

$$\begin{array}{rcl}
k[\![P]\!]\sigma & \triangleq & k[\![(P\sigma)]\!] \\
(S_1 \parallel S_2)\sigma & \triangleq & (S_1\sigma) \parallel (S_2\sigma) \\
(\mathsf{new}\, c.S)\sigma & \triangleq & \mathsf{new}\, c.(S\sigma') \ s.t.\ \sigma' = \{c\} \trianglelefteq \sigma
\end{array}$$

Definition B.0.2. *(Substitution $P\sigma$) We define substitution of σ on $P \in$ Proc, written $P\sigma$, as follows*

$$\begin{array}{rcl}
c!\bar{v}.P\sigma & \triangleq & (c\,\sigma)!(\bar{v}\,\sigma).(P\,\sigma) \\
u?\bar{x}.P\sigma & \triangleq & (u\,\sigma)?\bar{x}.(P\,\sigma') \ s.t.\ \sigma' = \bar{x} \trianglelefteq \sigma \\
\mathsf{new}\, c.(P'\sigma')\sigma & \triangleq & \mathsf{new}\, c.P \ s.t.\ \sigma' = \{c\} \trianglelefteq \sigma \\
\mathsf{if}\, u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P''\sigma & \triangleq & \mathsf{if}\, (u\sigma) = (v\sigma)\ \mathsf{then}\ (P'\sigma)\ \mathsf{else}\ (P''\sigma) \\
P' \parallel P''\sigma & \triangleq & (P'\sigma) \parallel (P''\sigma) \\
*P\sigma & \triangleq & *(P\,\sigma) \\
\mathsf{stop}\sigma & \triangleq & \mathsf{stop}
\end{array}$$

$$\begin{array}{rcl}
\{\textit{go}\ k.M\}^n\sigma & \triangleq & \{\textit{go}\ (k\sigma).(M\sigma)\}^n \\
\{u!\bar{v}.M\}^n\sigma & \triangleq & \{(u\,\sigma)!(\bar{v}\,\sigma).(M\sigma)\}^n \\
\{u?\bar{x}.M\}^n\sigma & \triangleq & \{(u\sigma)?\bar{x}.(M\sigma')\}^n \ \ s.t.\ \sigma' = \bar{x} \trianglelefteq \sigma \\
\{\textit{new}\,c.M\}^n\sigma & \triangleq & \{\textit{new}\,c.(M\sigma')\}^n \ \ s.t.\ \sigma' = \{c\} \trianglelefteq \sigma \\
\{\textit{if}\ u{=}v\ \textit{then}\ M\ \textit{else}\ M'\}^n\sigma & \triangleq & \{\textit{if}\ (u\sigma){=}(v\sigma)\ \textit{then}\ (M\sigma)\ \textit{else}\ (M'\sigma)\}^n \\
\{(M\sigma)\,\|\,(N\sigma)\}^n\sigma & \triangleq & \{M\,\|\,N\}^n \\
\{\textit{setC}(k).M\}^n\sigma & \triangleq & \{\textit{setC}(k\sigma).(M\sigma)\}^n \\
\{\textit{ok}\}^n\sigma & \triangleq & \{\textit{ok}\}^n \\
\{\textit{fail}\}^n\sigma & \triangleq & \{\textit{fail}\}^n \\
\boldsymbol{t}(c,\bar{d},n)\sigma & \triangleq & \boldsymbol{t}(c,\bar{d},n) \\
\{\boldsymbol{m}(c,\bar{x},k).M\}^n\sigma & \triangleq & \{\boldsymbol{m}((c\sigma),\bar{x},(k\sigma)).(M\sigma')\}^n \ \ s.t.\ \sigma' = \bar{x} \trianglelefteq \sigma
\end{array}$$

C. System Projection

Systems are internally made up of process, monitor and trace components. Given this setting, we are often interested in considering *one* aspect of a system's configuration. For instance, we may be interested in considering a system's monitoring effort to reason about the properties being verified. Conversely, other scenarios may require us to focus on the system's processing aspect, disregarding monitors. To this effect we introduce notation S_μ, representing a *projection* of S at a syntactic level. The resulting system extracts exclusively components of modality μ from the original system. Consider system $S = \mathsf{new}\, c_1.k[\![c_1!\bar{v}.\mathsf{stop} \parallel \mathbf{t}(c_1, \bar{y}, 1) \parallel \{\mathbf{m}(c_1, \bar{x}, k).d!\langle\rangle.\mathsf{stop}\}^2]\!]$ The projection of S in case of each modality works out to

$$\begin{array}{rcl} S_p & = & \mathsf{new}\, c_1.k[\![c_1!\bar{v}.\mathsf{stop}]\!] \\ S_m & = & \mathsf{new}\, c_1.k[\![\{\mathbf{m}(c_1, \bar{x}, k).d!\langle\rangle.\mathsf{stop}\}^2]\!] \\ S_t & = & \mathsf{new}\, c_1.k[\![\mathbf{t}(c_1, \bar{y}, 1)]\!] \end{array}$$

We next present the definition of process projection, due to its use when proving the result that processing behaviour is unaffected by the monitoring semantics (Sec. 5.4). The definition is presented over two stratified levels; at the system and at the process level.

Definition C.0.3. *(Process projection at the system level S_P) Process projection for system S, written S_P is defined as*

$$\begin{array}{rcl} (k[\![P]\!])_p & \triangleq & k[\![P_p]\!] \\ (\mathsf{new}\, c.S)_p & \triangleq & \mathsf{new}\, c.(S_p) \\ (S' \parallel S'')_p & \triangleq & S'_p \parallel S''_p \end{array}$$

Projection at the *system* level pushes projection to its internal constituents, with the case of $S = k[\![P]\!]$ pushing projection to the process level.

Definition C.0.4. *(Process Projection at the process level P_P) Process projection for process P, written P_P is defined as follows*

$$
\begin{array}{lcl}
(u!\bar{v}.P)_p & \triangleq & u!\bar{v}.(P_p) \\
(u?\bar{x}.P)_p & \triangleq & u?\bar{x}.(P_p) \\
(\mathsf{new}\, c.P)_p & \triangleq & \mathsf{new}\, c.(P_p) \\
(\mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P'')_p & \triangleq & \mathsf{if}\ u = v\ \mathsf{then}\ (P'_p)\ \mathsf{else}\ (P''_p) \\
(P' \parallel P'')_p & \triangleq & (P'_p) \parallel (P''_p) \\
(*P)_p & \triangleq & *(P_p) \\
(\mathsf{stop})_p & \triangleq & \mathsf{stop} \\
(\{M\}^n)_p & \triangleq & \mathsf{stop} \\
(\mathbf{t}(c, \bar{d}, n))_p & \triangleq & \mathsf{stop}
\end{array}
$$

The definition of projection at the *process* level converts monitor and trace components to stop, which does nothing. Unnecessary components are hence rendered inert. On the other hand, process components are unaltered.

D. Proof $(P\sigma)_P = P_p\sigma$

Proof. By induction on the structure of P.

Base Cases:

– $P =$ **stop:**
By defn. of P_P, $\mathsf{stop}_P = \mathsf{stop}$. We are hence required to prove $(\mathsf{stop}\,\sigma)_P = \mathsf{stop}\sigma$.

$LHS = (\mathsf{stop}\sigma)_P = \mathsf{stop}_P$ $\quad(Def^{\underline{n}}\ P\sigma)$
$\mathsf{stop}_P = \mathsf{stop}$ $\quad(Def^{\underline{n}}\ P_P)$

$RHS = \mathsf{stop}\sigma = \mathsf{stop}$ $\quad(Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

– $P = \{\mathsf{ok}\}^n$**:**
By defn. of P_P, $(\{\mathsf{ok}\}^n)_P = \mathsf{stop}$. We are hence required to prove $(\{\mathsf{ok}\}^n\sigma)_P = \mathsf{stop}\sigma$.

$LHS = (\{\mathsf{ok}\}^n\sigma)_P = (\{\mathsf{ok}\sigma\}^n)_P$ $\quad(Def^{\underline{n}}\ P\sigma)$
$(\{\mathsf{ok}\sigma\}^n)_P = \mathsf{stop}$ $\quad(Def^{\underline{n}}\ P_P)$

$RHS = \mathsf{stop}\sigma = \mathsf{stop}$ $\quad(Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

– $P = \{\mathsf{fail}\}^n$**:**
Same as in the case of $P = \{\mathsf{ok}\}^n$.

– $P = \{\mathsf{stop}\}^n$**:**
Same as in the case of $P = \{\mathsf{ok}\}^n$.

Inductive Hypothesis: $(P\sigma)_P = P_p\sigma$

Inductive Cases:

– $P = c!\bar{d}.P'$:

By defn. of P_P, $(c!\bar{d}.P')_P = c!\bar{d}.(P'_P)$. We are hence required to prove

$$((c!\bar{d}.P')\sigma)_P = (c!\bar{d}.(P'_P))\sigma.$$

We shall write $(c!\bar{d}.P')\sigma = c'!\bar{d}'.(P'\sigma)$, where c' and d' are the result of substitution as defined by σ. Clearly, $c = c'$ iff $c \notin dom(\sigma)$ and $\bar{d} = \bar{d}'$ iff $\forall x \in \bar{d}.x \notin dom(\sigma)$.

$LHS = ((c!\bar{d}.P')\sigma)_P = (c'!\bar{d}'.(P'\sigma))_P$ $\quad (Def^{\underline{n}}\ P\sigma)$
$(c'!\bar{d}'.(P'\sigma))_P = c'!\bar{d}'.((P'\sigma)_P)$ $\quad (Def^{\underline{n}}\ P_P)$
$c'!\bar{d}'.((P'\sigma)_P) = c'!\bar{d}'.(P'_P\sigma))$ $\quad$ (IH)

Clearly, substitution of σ on $c!\bar{d}.((P')_P)$ affects $c, \bar{d}$ in the same way as before.
Hence, $(c!\bar{d}.(P'_P))\sigma = c'!\bar{d}'.(P'_P\sigma)$.

$RHS = (c!\bar{d}.((P')_P))\sigma = c'!\bar{d}'.(P'_P\sigma)$ $\quad (Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

– $P = c?\bar{x}.P'$:

By defn. of P_P, $(c?\bar{x}.P')_P = c?\bar{x}.(P'_P)$. We are hence required to prove

$$((c?\bar{x}.P')\sigma)_P = (c?\bar{x}.(P'_P))\sigma.$$

Although c is free in term $c?\bar{x}.P'$, variable tuple $\bar{x}$ is considered bound. Hence, term $(c?\bar{x}.P')\sigma$ evaluates to $c'?\bar{x}.(P'\sigma')$, where $\sigma' = (\bar{x} \trianglelefteq \sigma) \triangleleft \sigma$ *i.e.*, restricting the domain of σ to all variables which are not in $\bar{x}$. The value of c' depends on whether $c \in dom(\sigma)$, as before.

$LHS = ((c?\bar{x}.P')\sigma)_P = (c'?\bar{x}.(P'\sigma'))_P$ $\quad (Def^{\underline{n}}\ P\sigma)$
$(c'?\bar{x}.(P'\sigma'))_P = c'?\bar{x}.((P'\sigma')_P)$ $\quad (Def^{\underline{n}}\ P_P)$
$c'?\bar{x}.((P'\sigma')_P) = c'?\bar{x}.(P'_P\sigma'))$ $\quad$ (IH)

Substitution of σ on $c?\bar{x}.((P')_P)$ affects $c, \bar{d}, \sigma$ in the same way as before.
Hence, $(c?\bar{x}.(P'_P))\sigma = c'?\bar{x}.(P'_P\sigma')$.

$RHS = (c?\bar{x}.((P')_P))\sigma = c'?\bar{x}.(P'_P\sigma')$ $\quad (Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

– $P = \mathsf{new}\, c.P'$:
By defn. of P_P, $(\mathsf{new}\, c.P')_P = \mathsf{new}\, c.(P'_P)$. We are hence required to prove

$$((\mathsf{new}\, c.P')\sigma)_P = (\mathsf{new}\, c.(P'_P))\sigma.$$

Channel name c is bound in term $\mathsf{new}\, c.P'$. Hence, term $(\mathsf{new}\, c.P')\sigma$ evaluates to $\mathsf{new}\, c.(P'\sigma')$, where $\sigma' = (dom(\sigma) \setminus \{c\}) \lhd \sigma$ *i.e.*, restricting the domain of σ through the removal of c.

$LHS = ((\mathsf{new}\, c.P')\sigma)_P = (\mathsf{new}\, c.(P'\sigma'))_P$ ($Def^{\underline{n}}\ P\sigma$)
$(\mathsf{new}\, c.(P'\sigma'))_P = \mathsf{new}\, c.((P'\sigma')_P)$ ($Def^{\underline{n}}\ P_P$)
$\mathsf{new}\, c.((P'\sigma')_P) = \mathsf{new}\, c.(P'_P\sigma')$ (IH)

Substitution of σ on $\mathsf{new}\, c.(P'_P)$ affects σ in the same way as before. Hence, $(\mathsf{new}\, c.P'_P)\sigma = \mathsf{new}\, c.(P'_P\sigma')$.

$RHS = (\mathsf{new}\, c.P'_P)\sigma = \mathsf{new}\, c.(P'_P\sigma')$ ($Def^{\underline{n}}\ P\sigma$)

Hence, $LHS = RHS$.

– $P = \mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P''$:
By defn. of P_P, $(\mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P'')_P = \mathsf{if}\ u = v\ \mathsf{then}\ P'_P\ \mathsf{else}\ P''_P$. We are hence required to prove $((\mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P'')\sigma)_P = (\mathsf{if}\ u = v\ \mathsf{then}\ P'_P\ \mathsf{else}\ P''_P)\sigma$.

We take $(\mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P'')\sigma = (\mathsf{if}\ u' = v'\ \mathsf{then}\ (P'\sigma)\ \mathsf{else}\ (P''\sigma))$, where u', v' are the result of substitution as defined by σ.

$LHS = ((\mathsf{if}\ u = v\ \mathsf{then}\ P'\ \mathsf{else}\ P'')\sigma)_P =$
$(\mathsf{if}\ u' = v'\ \mathsf{then}\ (P'\sigma)\ \mathsf{else}\ (P''\sigma))_P$ ($Def^{\underline{n}}\ P\sigma$)
$(\mathsf{if}\ u' = v'\ \mathsf{then}\ (P'\sigma)\ \mathsf{else}\ (P''\sigma))_P =$
$\mathsf{if}\ u' = v'\ \mathsf{then}\ ((P'\sigma)_P)\ \mathsf{else}\ ((P''\sigma)_P)$ ($Def^{\underline{n}}\ P_P$)
$\mathsf{if}\ u' = v'\ \mathsf{then}\ ((P'\sigma)_P)\ \mathsf{else}\ ((P''\sigma)_P) =$
$\mathsf{if}\ u' = v'\ \mathsf{then}\ ((P'_P\sigma))\ \mathsf{else}\ ((P''_P\sigma))$ (IH)

Substitution of σ on $\mathsf{if}\ u = v\ \mathsf{then}\ P'_P\ \mathsf{else}\ P''_P$ affects u, v in the same way as before. Hence, $(\mathsf{if}\ u = v\ \mathsf{then}\ P'_P\ \mathsf{else}\ P''_P)\sigma = (\mathsf{if}\ u' = v'\ \mathsf{then}\ (P'_P\sigma)\ \mathsf{else}\ (P''_P\sigma))$.

$RHS = (\mathsf{if}\ u = v\ \mathsf{then}\ P'_P\ \mathsf{else}\ P''_P)\sigma =$
$\mathsf{if}\ u' = v'\ \mathsf{then}\ (P'_P\sigma)\ \mathsf{else}\ (P''_P\sigma)$ ($Def^{\underline{n}}\ P\sigma$)

Hence, $LHS = RHS$.

$-P = P' \parallel P''$:

By defn. of P_P, $(P' \parallel P'')_P = (P'_P) \parallel (P''_P)$. We are hence required to prove

$$((P' \parallel P'')\sigma)_P = ((P'_P) \parallel (P''_P))\sigma.$$

$LHS = ((P' \parallel P'')\sigma)_P = ((P'\sigma) \parallel (P''\sigma))_P$ $(Def^{\underline{n}}\ P\sigma)$

$((P'\sigma) \parallel (P''\sigma))_P = (P'\sigma)_P \parallel (P''\sigma)_P$ $(Def^{\underline{n}}\ P_P)$

$(P'\sigma)_P \parallel (P''\sigma)_P = (P'_P\sigma) \parallel (P''_P\sigma)$ (IH)

$RHS = ((P'_P) \parallel (P''_P))\sigma = (P'_P\sigma) \parallel (P''_P\sigma)$ $(Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

$-P = *P'$:

By defn. of P_P, $(*P')_P = *(P'_P)$. We are hence required to prove

$$(*P'\sigma)_P = (*(P'_P))\sigma.$$

$LHS = (*P'\sigma)_P = (*(P'\sigma))_P$ $(Def^{\underline{n}}\ P\sigma)$

$(*(P'\sigma))_P = *((P'\sigma)_P)$ $(Def^{\underline{n}}\ P_P)$

$*((P'\sigma)_P) = *(P'_P\sigma)$ (IH)

$RHS = (*(P'_P))\sigma = *(P'_P\sigma)$ $(Def^{\underline{n}}\ P\sigma)$

Hence, $LHS = RHS$.

Proof of lemma for all monitor operators take the same structure, by virtue of the fact that $(\{M\}^n)_P = \mathsf{stop}$ and $(\mathbf{t}(c, \bar{d}, n))_P = \mathsf{stop}$. Intuitively, this implies that it does not matter if projection is applied before or after substitution, we still end up with stop. We present below a typical case exposing this intuition.

$-\{\mathsf{go}\ k.M\}^n$:

By defn. of P_P, $(\{\mathsf{go}\ k.M\}^n)_P = \mathsf{stop}$. We are hence required to prove

$$(\{\mathsf{go}\ k.M\}^n\sigma)_P = \mathsf{stop}\,\sigma.$$

We shall write $\{\mathsf{go}\ k.M\}^n\sigma = \{\mathsf{go}\ k'.(M\sigma)\}^n$, where c' is the result of substitution as defined by σ.

$LHS = (\{\mathsf{go}\ k.M\}^n\sigma)_P = (\{\mathsf{go}\ k'.(M\sigma)\}^n)_P$ $(Def^{\underline{n}}\ P\sigma)$

$(\{\mathsf{go}\ k'.(M\sigma)\}^n)_P = \mathsf{stop}$ $(Def^{\underline{n}}\ P_P)$

$RHS = \mathsf{stop}\sigma = \mathsf{stop}$ $(Def^{\underline{n}}\ P\sigma)$

Hence, *LHS* = *RHS*.

All other cases follow the same structure.

□

E. Proof $((C \equiv R) \land (C \xrightarrow{\alpha} C')) \Rightarrow ((R \xrightarrow{\alpha} R') \land (C' \equiv R'))$

Proof. Proof by rule induction on the derivation of $(C \equiv R)$.

Base Cases:

• (S-Com) : $\delta \rhd S_1 \parallel S_2 \equiv \delta \rhd S_2 \parallel S_1$

Hence, this case considers the situation where $C = \delta \rhd (S_1 \parallel S_2)$. We are hence required to prove that $((\delta \rhd S_1 \parallel S_2 \equiv \delta \rhd S_2 \parallel S_1) \land ((\delta \rhd S_1 \parallel S_2) \xrightarrow{\alpha} A)) \Rightarrow (((\delta \rhd S_2 \parallel S_1) \xrightarrow{\alpha} B) \land (A \equiv B))$. The structure of configuration $\delta \rhd S_1 \parallel S_2$ indicates the possible use of one of six rules for the derivation of $(\delta \rhd S_1 \parallel S_2) \xrightarrow{\alpha} A$. We shall be considering each case.

–(Cntx$_2$): The rule dictates that in this case $A = \delta' \rhd S_1' \parallel S_2$, such that $(\delta \rhd S_1 \parallel S_2) \xrightarrow{\alpha} (\delta' \rhd S_1' \parallel S_2)$. By rule Cntx$_2$ we can infer transition $\delta \rhd S_1 \xrightarrow{\alpha} \delta' \rhd S_1'$.

Using rule Cntx$_3$ we can hence infer $(\delta \rhd S_2 \parallel S_1) \xrightarrow{\alpha} (\delta' \rhd S_2 \parallel S_1')$ which gives us the required matching move, since the transitions match and $\delta' \rhd S_2 \parallel S_1' \equiv \delta' \rhd S_1' \parallel S_2$.

– (Cntx$_3$): Analogous to the above argument.

– (Com$_1$): Hence $A = \delta \rhd \mathsf{new}\,\bar{b}.(S_1' \parallel S_2')$, $\alpha = \tau_{(\mu:k,l)}$ such that $(\delta \rhd (S_1 \parallel S_2)) \xrightarrow{\tau_{(\mu:k,l)}} (\delta' \rhd \mathsf{new}\,\bar{b}.(S_1' \parallel S_2'))$. By Com$_1$ we can infer transitions *(i)* $\delta \rhd S_1 \xrightarrow{(\bar{b})c!d_{\langle p:k \rangle}} \delta' \rhd S_1'$, *(ii)* $\delta \rhd S_2 \xrightarrow{c?d_{\langle p:k \rangle}} \delta \rhd S_2'$.

By rule Com$_2$ we can hence infer $(\delta \rhd (S_2 \parallel S_1)) \xrightarrow{\tau_{(\mu:k,l)}} (\delta' \rhd \mathsf{new}\,\bar{b}.(S_2' \parallel S_1'))$ which gives us the required matching move, since the transitions match and $\delta' \rhd \mathsf{new}\,\bar{b}.(S_1' \parallel S_2') \equiv \delta' \rhd \mathsf{new}\,\bar{b}.(S_2' \parallel S_1')$.

– *(*Com$_2$*)*: Analogous argument to that given in case of Com$_1$.

The last two cases deal with the case when a monitor within S_1 or S_2 imports trace information. Proof of said case is analogous to that given for system communication, with one difference; counter state remains unaffected during trace import. We have to ensure that the property also holds in the said case.

– *(*Mon$_1$*)*: Hence $A = \delta \triangleright \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}')$, $\alpha = \tau_{(t:k,l)}$ such that $(\delta \triangleright (S_1 \parallel S_2)) \xrightarrow{\tau_{(t:k,l)}} (\delta \triangleright \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}'))$. By Mon$_1$ we can infer transitions *(i)* $\delta \triangleright S_1 \xrightarrow{(\bar{b})\mathsf{t}(c,d,n)_{\langle t:k\rangle}} \delta \triangleright S_1{}'$, *(ii)* $\delta \triangleright S_2 \xrightarrow{\mathsf{m}(c,d,k,n)_{\langle m:l\rangle}} \delta \triangleright S_2{}'$.

By rule Mon$_2$ we can hence infer $(\delta \triangleright (S_2 \parallel S_1)) \xrightarrow{\tau_{(t:k,l)}} (\delta \triangleright \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}'))$ which gives us the required matching move, since the transitions match and $\delta \triangleright \mathsf{new}\,\bar{b}.(S_1{}' \parallel S_2{}') \equiv \delta \triangleright \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_1{}')$.

– *(*Mon$_2$*)*: Analogous argument to that given in case of Mon$_1$.

- (S-Stop$_1$) : $\delta \triangleright S \parallel k[\![\mathsf{stop}]\!] \equiv \delta \triangleright S$

Hence, $C = \delta \triangleright S \parallel k[\![\mathsf{stop}]\!]$. We are subsequently required to prove

$$((\delta \triangleright S \parallel k[\![\mathsf{stop}]\!] \equiv \delta \triangleright S) \wedge ((\delta \triangleright S \parallel k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} A)) \Rightarrow (((\delta \triangleright S) \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

Configuration $\delta \triangleright S \parallel k[\![\mathsf{stop}]\!]$ indicates that sub-system $k[\![\mathsf{stop}]\!]$ is a terminal process, hence being incapable of executing further. This implies the possibility of only one rule for the derivation of $(\delta \triangleright S \parallel k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} A$.

– *(*Cntx$_2$*)*: Hence $A = \delta \triangleright S' \parallel k[\![\mathsf{stop}]\!]$ such that $(\delta \triangleright S \parallel k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} (\delta \triangleright S' \parallel k[\![\mathsf{stop}]\!])$. By Cntx$_2$ we can infer transition $\delta \triangleright S \xrightarrow{\alpha} \delta \triangleright S'$. This gives us the required transition, since the actions match and $\delta \triangleright S' \parallel k[\![\mathsf{stop}]\!] \equiv \delta \triangleright S'$.

Although rules Com$_1$,Com$_2$,Cntx$_3$,Mon$_1$ and Mon$_2$ might at first seem applicable due to their description of possible system behaviour when placed in parallel, they can be discounted based on the structure of C. For instance, Cntx$_3$ can be ignored since it describes the situation when the latter of two systems in parallel computes, which is impossible in this case since the latter system is a terminal located process. On the other hand, rules Mon$_1$ and Mon$_2$ can be discounted since they require both systems to produce either an m or a t action, which is impossible since $k[\![\mathsf{stop}]\!]$ cannot compute further. Com$_1$,Com$_2$ can be discounted for analogous reasons.

- (S-Stop$_2$) : $\delta \triangleright \mathsf{new}\,c.k[\![\mathsf{stop}]\!] \equiv \delta \triangleright k[\![\mathsf{stop}]\!]$

Whereby $C = \delta \rhd \mathsf{new}\, c.k[\![\mathsf{stop}]\!]$. Therefore, we are required to prove

$$((\delta \rhd \mathsf{new}\, c.k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} A) \Rightarrow (((\delta \rhd k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

On first glance, the only applicable rules for the derivation of $(\delta \rhd \mathsf{new}\, c.k[\![\mathsf{stop}]\!]) \xrightarrow{\alpha} A$ in this case are $\textsc{Cntx}_1$, $\textsc{Open}_s$ and $\textsc{Open}_t$, due to their description of behaviour for systems of the form $\mathsf{new}\, c.S$. However, all three rules are based on the premise that S can perform a transition. However, in our case S is a terminal located process, implying that C cannot satisfy any of the three applicable rules. This implies that C cannot compute further, rendering the statement to prove vacuously true.

- (S-$\textsc{Flip}$) : $\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S \equiv \delta \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S$

  Such that $C = \delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S$. In this case, we are required to prove

$$((\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S \equiv \delta \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S) \wedge ((\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S) \xrightarrow{\alpha} A)) \Rightarrow$$
$$(((\delta \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S) \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

  The structure of C dictates the possible use of two rules for the derivation of $(\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S) \xrightarrow{\alpha} A$.

  – ($\textsc{Cntx}_1$): Hence $\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S \xrightarrow{\alpha} \delta' \rhd \mathsf{new}\, c_1.A'$, thus implying transition $\delta \rhd \mathsf{new}\, c_2.S \xrightarrow{\alpha} \delta' \rhd A'$. This latter transition could have been derived using one of three rules:

  - ($\textsc{Cntx}_1$): Hence $\delta \rhd \mathsf{new}\, c_2.S \xrightarrow{\alpha} \delta' \rhd \mathsf{new}\, c_2.S'$, such that $\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'$. This implies that the original transition is of the form $(\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S) \xrightarrow{\alpha} (\delta' \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S')$.

    Using rule $\textsc{Cntx}_1$ twice and derived transition $\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'$ we derive $(\delta \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S) \xrightarrow{\alpha} (\delta' \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S')$, which gives us the desired transition, since the actions match and $\delta' \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S' \equiv \delta' \rhd \mathsf{new}\, c_2.\mathsf{new}\, c_1.S'$. Note that we needn't worry about side conditions during the derivation of the latter transition, since their satisfaction is guaranteed through assumptions during the derivation of $\delta \rhd S \xrightarrow{\alpha} \delta' \rhd S'$.

  - ($\textsc{Open}_s$): Hence $\delta \rhd \mathsf{new}\, c_2.S \xrightarrow{(c_2,\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$, such that $\delta \rhd S \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$. This implies that the original transition is of the form $(\delta \rhd \mathsf{new}\, c_1.\mathsf{new}\, c_2.S) \xrightarrow{(c_2,\bar{b})c!d_{\langle p:k\rangle}} (\delta' \rhd \mathsf{new}\, c_1.S')$.

Using rule $\textsc{Cntx}_1$ and derived transition $\delta \rhd S \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$ we derive $\delta \rhd \mathsf{new}\,c_1.S \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd \mathsf{new}\,c_1.S'$. Subsequently using rule $\textsc{Open}$, we next derive transition $\delta \rhd \mathsf{new}\,c_2.\mathsf{new}\,c_1.S \xrightarrow{(c_2,\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd \mathsf{new}\,c_1.S'$, which gives us the desired transition, since the actions match and $\delta' \rhd \mathsf{new}\,c_1.S' \equiv \delta' \rhd \mathsf{new}\,c_1.S'$. Once more, satisfaction of the side condition during extraction of the latter transition is guaranteed through assumptions made during the derivation of $\delta \rhd S \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$.

- ($\textsc{Open}_t$): Analogous to the case of $\textsc{Open}_s$.

– ($\textsc{Open}$): We hence infer transition $(\delta \rhd \mathsf{new}\,c_1.\mathsf{new}\,c_2.S) \xrightarrow{(c_1,\bar{b})c!d_{\langle p:k\rangle}} (\delta' \rhd \mathsf{new}\,c_2.S')$, implying validity of transition $(\delta \rhd \mathsf{new}\,c_2.S) \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} (\delta' \rhd \mathsf{new}\,c_2.S')$.

Given the structure of the latter transition (especially the restriction on its action and residual configuration's structure), this implies that it must have been derived using rule $\textsc{Cntx}_1$. Hence, through $\textsc{Cntx}_1$ we can derive transition $\delta \rhd S \xrightarrow{(\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$.

Through rule $\textsc{Open}_s$ we derive $\delta \rhd \mathsf{new}\,c_1.S \xrightarrow{(c_1,\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd S'$. We subsequently use rule $\textsc{Cntx}_1$ to derive $\delta \rhd \mathsf{new}\,c_2.\mathsf{new}\,c_1.S \xrightarrow{(c_1,\bar{b})c!d_{\langle p:k\rangle}} \delta' \rhd \mathsf{new}\,c_2.S'$, which gives us the desired transition, since the actions match and $\delta' \rhd \mathsf{new}\,c_2.S' \equiv \delta' \rhd \mathsf{new}\,c_2.S'$.

- (S-Assoc) : $\delta \rhd S_1 \parallel (S_2 \parallel S_3) \equiv \delta \rhd (S_1 \parallel S_2) \parallel S_3$

  Hence, $C = \delta \rhd S_1 \parallel (S_2 \parallel S_3)$. We are therefore tasked with proving

$$((\delta \rhd S_1 \parallel (S_2 \parallel S_3) \equiv \delta \rhd (S_1 \parallel S_2) \parallel S_3) \wedge ((\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\alpha} A)) \Rightarrow$$
$$(((\delta \rhd (S_1 \parallel S_2) \parallel S_3) \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

  Six possible rules could have been used for the derivation of $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\alpha} A$.

  – ($\textsc{Cntx}_2$): We hence infer transition $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\alpha} (\delta' \rhd S_1{}' \parallel (S_2 \parallel S_3))$, from which we infer $(\delta \rhd S_1 \xrightarrow{\alpha} (\delta' \rhd S_1{}')$.

  Using rule $\textsc{Cntx}$ twice and transition $(\delta \rhd S_1 \xrightarrow{\alpha} (\delta' \rhd S_1{}')$, we infer $(\delta \rhd (S_1 \parallel S_2) \parallel S_3) \xrightarrow{\alpha} (\delta' \rhd (S_1{}' \parallel S_2) \parallel S_3)$, which gives us the required transition, since the actions match and $\delta' \rhd S_1{}' \parallel (S_2 \parallel S_3) \equiv \delta' \rhd (S_1{}' \parallel S_2) \parallel S_3$.

– *(*CNTX$_3$*)*: Thus obtaining transition $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\alpha} (\delta' \rhd S_1 \parallel A')$, from which we infer transition $(\delta \rhd S_2 \parallel S_3) \xrightarrow{\alpha} (\delta' \rhd A')$. This latter transition could have been inferred using one of six rules.

- *(*CNTX$_2$*)*: Thus, the latter two transition takes the form $(\delta \rhd S_2 \parallel S_3) \xrightarrow{\alpha} (\delta' \rhd S_2{}' \parallel S_3)$, which through rule CNTX$_2$ we infer $(\delta \rhd S_2) \xrightarrow{\alpha} (\delta' \rhd S_2{}')$. Moreover, the original transition takes the form $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\alpha} (\delta' \rhd S_1 \parallel (S_2{}' \parallel S_3))$.

Using rule CNTX twice and transition $(\delta \rhd S_2) \xrightarrow{\alpha} (\delta' \rhd S_2{}')$, we infer $(\delta \rhd (S_1 \parallel S_2) \parallel S_3) \xrightarrow{\alpha} (\delta' \rhd (S_1 \parallel S_2{}') \parallel S_3)$, which gives us the required transition, since the actions match and $\delta' \rhd S_1 \parallel (S_2' \parallel S_3) \equiv \delta' \rhd (S_1 \parallel S_2') \parallel S_3$.

- *(*CNTX$_3$*)*: Analogous to the above case.

- *(*COM$_1$*)*: Through rule COM$_1$ we infer that the transition takes the form $(\delta \rhd S_2 \parallel S_3) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd \mathsf{new}\,\bar{b}.(S_2{}' \parallel S_3{}'))$, thus implying (i) $(\delta \rhd S_2) \xrightarrow{(\bar{b})c!d\langle\mu:k\rangle} (\delta' \rhd S_2{}')$, and (ii) $(\delta \rhd S_3) \xrightarrow{c?d\langle\mu:l\rangle} (\delta \rhd S_3{}')$. Moreover, the original transition takes the form $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd S_1 \parallel (\mathsf{new}\,\bar{b}.(S_2{}' \parallel S_3{}')))$. Note that the structure of derived transitions from S_2 and S_3 impose a restriction on the structure of α.

Through rule CNTX$_3$ we derive transition $(\delta \rhd S_1 \parallel S_2) \xrightarrow{(\bar{b})c!d\langle\mu:k\rangle} (\delta' \rhd S_1 \parallel S_2{}')$. Subsequently through rule COM$_1$, we derive $(\delta \rhd (S_1 \parallel S_2) \parallel S_3) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd \mathsf{new}\,\bar{b}.((S_1 \parallel S_2{}') \parallel S_3))$, which gives us the required transition, since the actions match and $\delta' \rhd S_1 \parallel (\mathsf{new}\,\bar{b}.(S_2{}' \parallel S_3{}')) \equiv \delta' \rhd \mathsf{new}\,\bar{b}.((S_1 \parallel S_2{}') \parallel S_3)$.

Note that truth of the last statement regarding structural equivalence is based on the validity of the side condition that $bn(\bar{b}) \notin fn(S_1)$. We ascertain truth of this side condition next. Given that rule COM$_1$ was used during the derivation of the latter matching transition, this implies that $bn(\bar{b}) \cap fn(S_1) = \emptyset$. Hence, given that $fn(S_1) \subseteq fn(S_1 \parallel S_2)$ this implies truth of $bn(\bar{b}) \notin fn(S_1)$.

- *(*COM$_2$*)*: Analogous to the argument for COM$_1$.

The last two rules yet to be analysed involve rules MON$_1$ and MON$_2$. However, proof of their validity is once more analogous to the proof for COM$_1$ (minus one discrepancy regarding the behaviour of the counter state, which happens to be ineffectual).

- *(*MON$_1$*)*: Analogous to the argument for COM$_1$.

- *(*MON$_2$*)*: Analogous to the argument for COM$_1$.

– *(*Com$_1$*)*: Thus obtaining $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd \mathsf{new}\,\bar{b}.({S_1}' \parallel A'))$, from which we infer (i) $(\delta \rhd S_1) \xrightarrow{(\bar{b})c!d_{\langle\mu:k\rangle}} (\delta' \rhd {S_1}')$, and (ii) $(\delta \rhd S_2 \parallel S_3) \xrightarrow{c?d_{\langle\mu:l\rangle}} (\delta \rhd A')$. Given the latter transition's structure, more specifically the structure of its action which dictates that it must be an input action $c?d_{\langle\mu:l\rangle}$, this implies that the transition could only have been derived using one of two rules.

- *(*Cntx$_2$*)*: Hence $A' = {S_2}' \parallel S_3$, such that $(\delta \rhd S_2 \parallel S_3) \xrightarrow{c?d_{\langle\mu:l\rangle}} (\delta \rhd {S_2}' \parallel S_3)$ and $(\delta \rhd S_1 \parallel (S_2 \parallel S_3)) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd \mathsf{new}\,\bar{b}.({S_1}' \parallel ({S_2}' \parallel S_3)))$. By Cntx$_2$ and the latter transition, we infer $(\delta \rhd S_2) \xrightarrow{c?d_{\langle\mu:l\rangle}} (\delta \rhd {S_2}')$. Subsequently through rule Com, we get $(\delta \rhd S_1 \parallel S_2) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd \mathsf{new}\,\bar{b}.({S_1}' \parallel {S_2}'))$. Finally by rule Cntx$_2$ we get $(\delta \rhd (S_1 \parallel S_2) \parallel S_3) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta' \rhd (\mathsf{new}\,\bar{b}.({S_1}' \parallel {S_2}')) \parallel S_3)$, which gives us the required transition since the transitions match and $\delta' \rhd \mathsf{new}\,\bar{b}.({S_1}' \parallel ({S_2}' \parallel S_3)) \equiv \delta' \rhd (\mathsf{new}\,\bar{b}.({S_1}' \parallel {S_2}')) \parallel S_3$. This last statement about structural equivalence holds by virtue of the fact that $bn(\bar{b}) \notin fn(S_3)$ (known to be true as a direct implication of the application of the original Cntx$_2$).

- *(*Cntx$_3$*)*: Analogous to the above case for Cntx$_2$.

– *(*Com$_2$*)*: Analogous to the above argument for Com$_1$.

– *(*Mon$_1$*)*: Analogous to the above argument for Com$_1$.

– *(*Mon$_2$*)*: Analogous to the above argument for Com$_1$.

All cases so far have been rather straightforward, resulting in a more relaxed approach so far *wrt.* the handling of side conditions (mainly for brevity, since one could still discuss at length their validity for each case). On the other hand we next present the only non-trivial case of the entire proof, requiring a more subtle understanding of the way side conditions operate. To this effect, we treat the reasoning of side conditions in a more explicit manner throughout the next case.

• (S-Extr) : $\delta \rhd \mathsf{new}\,d.(S_1 \parallel S_2) \equiv \delta \rhd S_1 \parallel \mathsf{new}\,d.S_2 \; s.t. \, \mathrm{bn}(d) \notin \mathrm{fn}(S_1)$

Hence $C = \delta \rhd \mathsf{new}\,d.(S_1 \parallel S_2)$ and are required to prove

$$((\delta \rhd \mathsf{new}\,d.(S_1 \parallel S_2) \equiv \delta \rhd S_1 \parallel \mathsf{new}\,d.S_2) \wedge ((\delta \rhd \mathsf{new}\,d.(S_1 \parallel S_2)) \xrightarrow{\alpha} A)) \Rightarrow$$
$$(((\delta \rhd S_1 \parallel \mathsf{new}\,d.S_2) \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

The structure of C denotes three possible rules for the derivation of $(\delta \rhd \mathsf{new}\,d.(S_1 \parallel S_2)) \xrightarrow{\alpha} A$.

– *(*Open$_s$*)*: We know that $\mathrm{bn}(d) \notin \mathrm{fn}(S_1)$. However, the application of Open$_s$ to C demands that action α is of the form $(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}$ and also that $d \in \bar{d}'$. Hence we infer that action $(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}$ must have been effected by S_2, since d cannot appear free in S_1. This gives rise to a transition of the form $(\delta \triangleright \mathsf{new}\, d.(S_1 \parallel S_2)) \xrightarrow{(d,\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright (S_1 \parallel S_2'))$. Moreover, by Open$_s$ and Cntx$_3$ we infer $\delta \triangleright S_2 \xrightarrow{(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} \delta' \triangleright S_2'$.

By applying rule Open$_s$ to the derived transition above, we obtain $\delta \triangleright \mathsf{new}\, d.S_2 \xrightarrow{(d,\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} \delta' \triangleright S_2'$. Note that the above derivation is valid since we already ascertained that $d \in \bar{d}'$. Applying Cntx$_3$ to the latter transition gives $(\delta \triangleright S_1 \parallel (\mathsf{new}\, d.S_2)) \xrightarrow{(d,\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright S_1 \parallel S_2')$, which is the required transition, since the actions match and $\delta' \triangleright (S_1 \parallel S_2') \equiv (\delta' \triangleright S_1 \parallel S_2')$.

– *(*Open$_t$*)*: Analogous to the above argument for Open$_s$.

– *(*Cntx$_1$*)*: We hence obtain $(\delta \triangleright \mathsf{new}\, d.(S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta \triangleright \mathsf{new}\, d.A')$ $s.t.\, d \notin n(\alpha)$. Truth of this latter transition implies that it must have been derived from another transition of the form $(\delta \triangleright (S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta \triangleright A')$. We are next faced with a situation where either S_1 or S_2 execute independently, or they both participate in some internal action (either in the form of traditional communication, or as trace import). Hence, we next consider six transition rules for the derivation of $(\delta \triangleright (S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta \triangleright A')$.

- *(*Cntx$_2$*)*: Obtaining a transition of the form $(\delta \triangleright (S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta' \triangleright (S_1' \parallel S_2))$ $s.t.\, bn(\alpha) \cap fn(S_2) = \emptyset$, from which we also infer $(\delta \triangleright S_1) \xrightarrow{\alpha} (\delta' \triangleright S_1')$. The original transition hence takes the form $(\delta \triangleright \mathsf{new}\, d.(S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta' \triangleright \mathsf{new}\, d.(S_1' \parallel S_2))$.

By rule Cntx$_2$ and $(\delta \triangleright S_1) \xrightarrow{\alpha} (\delta' \triangleright S_1')$ we infer $(\delta \triangleright S_1 \parallel \mathsf{new}\, d.S_2) \xrightarrow{\alpha} (\delta' \triangleright S_1' \parallel \mathsf{new}\, d.S_2)$, which is valid since we know $bn(\alpha) \cap fn(S_2) = \emptyset$, and d is bound in $\mathsf{new}\, d.S_2$ (hence $bn(\alpha) \cap fn(\mathsf{new}\, d.S_2) = \emptyset$). This latter transition gives us a matching transition, since the actions match and $\delta' \triangleright \mathsf{new}\, d.(S_1' \parallel S_2) \equiv \delta' \triangleright S_1' \parallel \mathsf{new}\, d.S_2$. Since $\mathrm{bn}(d) \notin \mathrm{fn}(S_1)$ and $d \notin n(\alpha)$, then this implies that $\mathrm{bn}(d) \notin \mathrm{fn}(S_1')$ by virtue of transition $(\delta \triangleright S_1) \xrightarrow{\alpha} (\delta' \triangleright S_1')$, which satisfies the side condition necessary for validity of the previous statement on structural equivalence.

- *(*Cntx$_3$*)*: Analogous case to that proven above for Cntx$_2$.

- *(*Com$_1$*)*: Thus, the resulting transition is of the form $(\delta \triangleright (S_1 \parallel S_2)) \xrightarrow{\alpha} (\delta' \triangleright (\mathsf{new}\, \bar{b}.(S_1' \parallel S_2')))$ $s.t.\, \bar{b} \cap fn(S_1) = \emptyset$, from which we further derive transitions

(i) $(\delta \triangleright S_1) \xrightarrow{c?\bar{d}'_{\langle\mu:l\rangle}} (\delta \triangleright S_1{}')$, and (ii) $(\delta \triangleright S_2) \xrightarrow{(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright S_2{}')$. This implies that the original transition takes the form $(\delta \triangleright \mathsf{new}\, d.(S_1 \parallel S_2)) \xrightarrow{\tau_{\langle\mu:k,l\rangle}} (\delta' \triangleright \mathsf{new}\, d.(\mathsf{new}\, \bar{b}.(S_1{}' \parallel S_2{}')))$. Note that although we know that $d \notin n(\tau_{\langle\mu:k,l\rangle})$, we know nothing about whether d is free in $(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}$. We therefore have to consider both cases.

- Case $d \notin fn((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$:

Given that d is not free in $(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}$, then we can ensure by α-conversion that d does not appear in the aforementioned action, implying that $d \notin n((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$. Hence, given transition $(\delta \triangleright S_2) \xrightarrow{(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright S_2{}')$ and the guarantee that $d \notin n((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$, by CNTX₁ we infer $(\delta \triangleright \mathsf{new}\, d.S_2) \xrightarrow{(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright \mathsf{new}\, d.S_2{}')$. Applying rule COM₂ we get

$$(\delta \triangleright (S_1 \parallel \mathsf{new}\, d.S_2)) \xrightarrow{\tau_{\langle\mu:k,l\rangle}} (\delta' \triangleright \mathsf{new}\, \bar{b}.(S_1{}' \parallel \mathsf{new}\, d.S_2{}'))$$

Note that the above transition can be derived since we know that $\bar{b} \cap fn(\mathsf{new}\, d.S_1) = \emptyset$, since we know from the original application of COM₁ that $\bar{b} \cap fn(S_1) = \emptyset$. We hence get a matching transition, since the actions match and $\delta' \triangleright \mathsf{new}\, d.(\mathsf{new}\, \bar{b}.(S_1{}' \parallel S_2{}')) \equiv \delta' \triangleright \mathsf{new}\, \bar{b}.(S_1{}' \parallel \mathsf{new}\, d.S_2{}')$ by S-FLIP and S-EXTR.

Note that the last statement is only valid if $\mathrm{bn}(d) \notin \mathrm{fn}(S_1{}')$. Given that $d \notin n((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$, then this implies that $d \notin n(c?\bar{d}'_{\langle\mu:l\rangle})$ since $n(c?\bar{d}'_{\langle\mu:l\rangle}) \subseteq n((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$. Hence, given that $\mathrm{bn}(d) \notin \mathrm{fn}(S_1)$, $d \notin n(c?\bar{d}'_{\langle\mu:l\rangle})$ and by transition $(\delta \triangleright S_1) \xrightarrow{c?\bar{d}'_{\langle\mu:l\rangle}} (\delta \triangleright S_1{}')$ we know $\mathrm{bn}(d) \notin \mathrm{fn}(S_1{}')$ to be true.

- Case $d \in fn((\bar{b})c!\bar{d}'_{\langle\mu:k\rangle})$:

Given that d is free in action $(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}$ and $\bar{d}'$ is by definition free (in the previous action), this implies that $d \in \bar{d}'$. Hence, given transition $(\delta \triangleright S_2) \xrightarrow{(\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright S_2{}')$ and guarantee that $d \in \bar{d}'$, then by rule OPEN$_s$ we derive $(\delta \triangleright \mathsf{new}\, d.S_2) \xrightarrow{(d,\bar{b})c!\bar{d}'_{\langle\mu:k\rangle}} (\delta' \triangleright S_2{}')$. Subsequently through COM₂, we get

$$(\delta \triangleright S_1 \parallel \mathsf{new}\, d.S_2) \xrightarrow{\tau_{\langle\mu:k,l\rangle}} (\delta' \triangleright \mathsf{new}\, d.(\mathsf{new}\, \bar{b}.(S_1{}' \parallel S_2{}')))$$

which gives us the required matching move, since both the actions and the residual systems match.

- *(*COM₂*)*: Analogous case to that proven above for COM₁.

- *(*Mon$_1$*)*: Analogous case to that proven above for Com$_1$.

- *(*Mon$_2$*)*: Analogous case to that proven above for Com$_1$.

Before moving on to proving our result for the rules of inference, there is still one axiom left requiring proof *i.e.*, an implicit axiom defining $\equiv$ as a reflexive relation (since $\equiv$ is defined as an equivalence relation) *i.e.*, stating that $C \equiv C$.

• (S-Refl) : $\delta \rhd C \equiv \delta \rhd C$

We are hence required to prove that

$$((\delta \rhd C \equiv \delta \rhd C) \wedge (\delta \rhd C \xrightarrow{\alpha} A)) \Rightarrow ((\delta \rhd C \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

Clearly, any transition of the form $\delta \rhd C \xrightarrow{\alpha} A$ can be matched by itself, such that residual configuration A (for both transitions) is structurally equivalent to itself.

Inductive Hypothesis:

$$((C \equiv R) \wedge (C \xrightarrow{\alpha} C')) \Rightarrow (\exists R' : \text{Sys}. (R \xrightarrow{\alpha} R') \wedge (C' \equiv R'))$$

Inductive Cases:

• (S-Contextual$_1$) : $(\delta \rhd S_1 \equiv \delta \rhd S_2) \Rightarrow (\delta \rhd S_1 \parallel V \equiv \delta \rhd S_2 \parallel V)$

We hence know (i) $\delta \rhd S_1 \parallel V \equiv \delta \rhd S_2 \parallel V$, (ii) $(\delta \rhd S_1 \parallel V \xrightarrow{\alpha} A)$ and are required to prove (iii) $(\delta \rhd \delta \rhd S_2 \parallel V \xrightarrow{\alpha} B)$, (iv) $(A \equiv B)$.

From (i) we obtain equality $\delta \rhd S_1 \equiv \delta \rhd S_2$ by rule S-Contextual$_1$. Now, the structure of $(\delta \rhd S_1 \parallel V \xrightarrow{\alpha} A)$ implies that it must have been derived through one of six rules.

- *(*Cntx$_2$*)*: We hence get a transition of the form $(\delta \rhd S_1 \parallel V) \xrightarrow{\alpha} (\delta \rhd S_1' \parallel V)$, from which we infer $\delta \rhd S_1 \xrightarrow{\alpha} \delta \rhd S_1'$. Given this derived transition and knowledge of equality $\delta \rhd S_1 \equiv \delta \rhd S_2$, then by the inductive hypothesis we obtain $\delta \rhd S_2 \xrightarrow{\alpha} \delta \rhd S_2'$ such that $\delta \rhd S_1' \equiv \delta \rhd S_2'$.

  Using Cntx$_2$ we derive $(\delta \rhd S_2 \parallel V) \xrightarrow{\alpha} (\delta \rhd S_2' \parallel V)$ which is the required matching transition, since the actions match and $\delta \rhd S_1' \parallel V \equiv \delta \rhd S_2' \parallel V$. Validity of this latter equality is by virtue of statement $\delta \rhd S_1' \equiv \delta \rhd S_2'$, and the fact that $\equiv$ is a contextual relation.

- *(*CNTX$_3$*)*: Hence we get a transition of the form $(\delta \triangleright S_1 \parallel V) \xrightarrow{\alpha} (\delta \triangleright S_1 \parallel V')$, from which we infer $\delta \triangleright V \xrightarrow{\alpha} \delta \triangleright V'$. Through rule CNTX$_3$ and the latter transition we infer

$$(\delta \triangleright S_2 \parallel V) \xrightarrow{\alpha} (\delta \triangleright S_2 \parallel V')$$

which we argue is the matching transition, since the actions match and $\delta \triangleright S_1 \parallel V' \equiv \delta \triangleright S_2 \parallel V'$. The previous statement is true by virtue of the fact that $\delta \triangleright S_1 \equiv \delta \triangleright S_2$ and $\equiv$ is a contextual relation.

- *(*COM$_1$*)*: Such that $(\delta \triangleright S_1 \parallel V) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta \triangleright \mathsf{new}\,\bar{b}.S_1{}' \parallel V')$, while also implying two additional transitions (v) $(\delta \triangleright S_1) \xrightarrow{(\bar{b})c!\bar{d}'\langle\mu:k\rangle} (\delta \triangleright S_1{}')$ and (vi) $(\delta \triangleright V) \xrightarrow{c?\bar{d}'\langle\mu:l\rangle} (\delta \triangleright V')$.

By transition (v), statement $\delta \triangleright S_1 \equiv \delta \triangleright S_2$ and the inductive hypothesis we obtain $(\delta \triangleright S_2) \xrightarrow{(\bar{b})c!\bar{d}'\langle\mu:k\rangle} (\delta \triangleright S_2{}')$ such that $\delta \triangleright S_1{}' \equiv \delta \triangleright S_2{}'$. Given the latter derived transition and (vi) we derive an additional transition (through COM$_1$) of the form

$$(\delta \triangleright S_2 \parallel V) \xrightarrow{\tau\langle\mu:k,l\rangle} (\delta \triangleright \mathsf{new}\,\bar{b}.S_2{}' \parallel V')$$

which is our required matching transition, since the actions match and $\delta \triangleright \mathsf{new}\,\bar{b}.S_1{}' \parallel V' \equiv \delta \triangleright \mathsf{new}\,\bar{b}.S_2{}' \parallel V'$. Note that the previous transition is true by virtue of the fact that $\delta \triangleright S_1 \equiv \delta \triangleright S_2$, and $\equiv$ is a contextual relation.

- *(*COM$_2$*)*: Analogous argument to that given for COM$_1$.

- *(*MON$_1$*)*: Analogous argument to that given for COM$_1$.

- *(*MON$_2$*)*: Analogous argument to that given for COM$_1$.

• (S-CONTEXTUAL$_2$) : $(\delta \triangleright S_1 \equiv \delta \triangleright S_2) \Rightarrow (\delta \triangleright V \parallel S_1 \equiv \delta \triangleright V \parallel S_2)$

Analogous argument to that given for S-CONTEXTUAL$_1$.

• (S-CONTEXTUAL$_3$) : $(\delta \triangleright S_1 \equiv \delta \triangleright S_2) \Rightarrow (\delta \triangleright \mathsf{new}\,d.S_1 \equiv \delta \triangleright \mathsf{new}\,d.S_2)$

We are thus tasked with proving statement

$$((\delta \triangleright \mathsf{new}\,d.S_1 \equiv \delta \triangleright \mathsf{new}\,d.S_2) \wedge (\delta \triangleright \mathsf{new}\,d.S_1 \xrightarrow{\alpha} A)) \Rightarrow ((\delta \triangleright \mathsf{new}\,d.S_2 \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

Through equivalence $\delta \triangleright \mathsf{new}\,d.S_1 \equiv \delta \triangleright \mathsf{new}\,d.S_2$ and S-CONTEXTUAL$_3$ we derive $\delta \triangleright S_1 \equiv \delta \triangleright S_2$. Moreover, transition $\delta \triangleright \mathsf{new}\,d.S_1 \xrightarrow{\alpha} A$ could have been derived using one of three possible rules.

– *(*Open$_s$*)*: Thus obtaining a statement of the form $\delta \triangleright \mathsf{new}\, d.S_1 \xrightarrow{(d,\bar{b})c!\bar{d}'\langle\mu:k\rangle} \delta' \triangleright S_1{}'$. We also infer transition $\delta \triangleright S_1 \xrightarrow{(\bar{b})c!\bar{d}'\langle\mu:k\rangle} \delta' \triangleright S_1{}'$ by Open$_s$. Given knowledge of $\delta \triangleright S_1 \equiv \delta \triangleright S_2$ and the latter transition, by the inductive hypothesis we obtain $\delta \triangleright S_2 \xrightarrow{(\bar{b})c!\bar{d}'\langle\mu:k\rangle} \delta' \triangleright S_2{}'$ such that $\delta' \triangleright S_1{}' \equiv \delta' \triangleright S_2{}'$.

By rule Open$_s$ we infer $\delta \triangleright \mathsf{new}\, d.S_2 \xrightarrow{(d,\bar{b})c!\bar{d}'\langle\mu:k\rangle} \delta' \triangleright S_2{}'$. Note that we can infer the previous transition since we know $d \in \bar{d}'$ through the original application of Open$_s$. We argue this latter transition is the required matching move, since the actions match and $\delta' \triangleright S_1{}' \equiv \delta' \triangleright S_2{}'$ (derived above).

– *(*Open$_t$*)*: Analogous to the above argument for Open$_s$.

– *(*Cntx$_1$*)*: We thus get $\delta \triangleright \mathsf{new}\, d.S_1 \xrightarrow{\alpha} \delta' \triangleright \mathsf{new}\, d.S_1{}'$. We also infer transition $\delta \triangleright S_1 \xrightarrow{\alpha} \delta' \triangleright S_1{}'$ by Cntx$_1$. Given knowledge of $\delta \triangleright S_1 \equiv \delta \triangleright S_2$ and the latter transition, by the inductive hypothesis we get $\delta \triangleright S_2 \xrightarrow{\alpha} \delta' \triangleright S_2{}'$ such that $\delta' \triangleright S_1{}' \equiv \delta' \triangleright S_2{}'$.

By rule Cntx$_1$ we infer $\delta \triangleright \mathsf{new}\, d.S_2 \xrightarrow{\alpha} \delta' \triangleright \mathsf{new}\, d.S_2{}'$. Once more, the necessary side condition for the latter application of Cntx$_1$ is satisfied by the former use of the same rule. We hence obtain the matching transition, since the actions match and $\delta' \triangleright \mathsf{new}\, d.S_1{}' \equiv \delta' \triangleright \mathsf{new}\, d.S_2{}'$ by virtue of proven equivalence $\delta' \triangleright S_1{}' \equiv \delta' \triangleright S_2{}'$ and the contextuality of $\equiv$.

- (S-Symm) : $(\delta \triangleright S_1 \equiv \delta \triangleright S_2) \Rightarrow (\delta \triangleright S_2 \equiv \delta \triangleright S_1)$

  We are hence required to prove that

$$((\delta \triangleright S_2 \equiv \delta \triangleright S_1) \wedge (\delta \triangleright S_2 \xrightarrow{\alpha} A)) \Rightarrow ((\delta \triangleright S_1 \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

Given that $(\delta \triangleright S_2 \equiv \delta \triangleright S_1)$ and we know of transition $\delta \triangleright S_2 \xrightarrow{\alpha} A$, then by the inductive hypothesis we get matching transition $(\delta \triangleright S_1 \xrightarrow{\alpha} B)$ such that $(A \equiv B)$.

- (S-Tran) : $((\delta \triangleright S_1 \equiv \delta \triangleright S_2) \wedge (\delta \triangleright S_2 \equiv \delta \triangleright S_3)) \Rightarrow (\delta \triangleright S_1 \equiv \delta \triangleright S_3)$

  We are required to prove that

$$((\delta \triangleright S_1 \equiv \delta \triangleright S_3) \wedge (\delta \triangleright S_1 \xrightarrow{\alpha} A)) \Rightarrow ((\delta \triangleright S_3 \xrightarrow{\alpha} B) \wedge (A \equiv B))$$

Given $(\delta \triangleright S_1 \equiv \delta \triangleright S_3)$ by rule S-Tran we infer two further equivalence statements $\delta \triangleright S_1 \equiv \delta \triangleright S_2$ and $\delta \triangleright S_2 \equiv \delta \triangleright S_3$.

Given knowledge of $\delta \rhd S_1 \equiv \delta \rhd S_2$ and transition $\delta \rhd S_1 \xrightarrow{\alpha} A$ by the inductive hypothesis we infer $\delta \rhd S_2 \xrightarrow{\alpha} A'$ such that $A \equiv A'$. Once more, given knowledge of $\delta \rhd S_2 \equiv \delta \rhd S_3$ and transition $\delta \rhd S_2 \xrightarrow{\alpha} A'$ by the inductive hypothesis we derive transition $\delta \rhd S_3 \xrightarrow{\alpha} B$ such that $A' \equiv B$. This latter transition satisfies the required statement to prove, since the actions match and $A \equiv B$. Note that truth of this equivalence is by virtue of statements $A \equiv A'$, $A' \equiv B$ and the transitivity of $\equiv$.

$\square$

F. $\approx_{\Omega_P}$ Is A Bisimilarity Up To Structural Equivalence

Proof. We first define bisimulation relation $(C, R) \in \mathcal{R}$ matching residual systems up to structurally equivalent terms. Outgoing transitions from configurations in $\mathcal{R}$ are filtered through Ω_P due to our interest in $\approx_{\Omega_P}$. More formally, we define $\mathcal{R}$ :: Conf $\leftrightarrow$ Conf adhering to the following two properties:

- $((C\ \mathcal{R}\ R) \wedge (C \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R') \wedge (C' \equiv \circ \mathcal{R} \circ \equiv R'))$
- $((C\ \mathcal{R}\ R) \wedge (R \xrightarrow{\alpha}_{\Omega_P} R')) \Rightarrow ((C \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C') \wedge (C' \equiv \circ \mathcal{R} \circ \equiv R'))$

where $\circ$ denotes relational composition, such that $\equiv \circ \mathcal{R} \circ \equiv$ is the resulting composition of $\equiv$ (twice) and $\mathcal{R}$. Hence, $\mathcal{R}$ specifies that pairing of residual configurations C', R' occurs up to structural equivalence. The desired result hence follows if we can prove that $\mathcal{R} \subseteq \approx_{\Omega_P}$.

To achieve this result, we define relation $\mathcal{S} \triangleq \equiv \circ \mathcal{R} \circ \equiv$, and prove that $\mathcal{S} \subseteq \approx_{\Omega_P}$. Given that $\equiv$ is an equivalence relation, by implication $\equiv$ contains the identity relation. Hence by *Def$^{\underline{n}}$* of $\mathcal{S}$, $\mathcal{R}$ must be contained in $\mathcal{S}$ *i.e.*, $\mathcal{R} \subseteq \mathcal{S}$. If we prove that $\mathcal{S} \subseteq \approx_{\Omega_P}$, this would prove that $\mathcal{R} \subseteq \mathcal{S} \subseteq \approx_{\Omega_P}$, implying that $\mathcal{R} \subseteq \approx_L$, which is our desired result.

To prove that $\mathcal{S} \subseteq \approx_{\Omega_P}$ we would like to prove that for any two configurations $(\Omega_P(C), \Omega_P(R)) \in \mathcal{S}$, $\mathcal{S}$ adheres to the following two properties

(1) – $((C\ \mathcal{S}\ R) \wedge (C \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R') \wedge (C'\ \mathcal{S}\ R'))$
(2) – $((C\ \mathcal{S}\ R) \wedge (R \xrightarrow{\alpha}_{\Omega_P} R')) \Rightarrow ((C \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C') \wedge (C'\ \mathcal{S}\ R'))$

Proof of (1)**:**

Given the structure of (1), we know *(i)* $(C\ \mathcal{S}\ R)$, *(ii)* $(C \xrightarrow{\alpha}_{\Omega_P} C')$ to be true, and are required to prove *(iii)* $(R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R') \wedge (C'\ \mathcal{S}\ R')$.

By (i) and *Def$^{\underline{n}}$* of $\mathcal{S}$, $(C, R) \in \equiv \circ \mathcal{R} \circ \equiv$. Hence

$$\exists C_1, R_1 : \textsc{Conf}.\ C \equiv C_1\ \mathcal{R}\ R_1\ \equiv R\ ...(iv)$$

Given that $C \equiv C_1$ (by (iv)) and $C \xrightarrow{\alpha}_{\Omega_P} C'$, by corollary 5.3.1 we conclude that *(v)* $C_1 \xrightarrow{\alpha}_{\Omega_P} C_1'$ and *(vi)* $C' \equiv C_1'$.

We also know $C_1\ \mathcal{R}\ R_1$ to be true (from (iv)). Hence, given that $(C_1\ \mathcal{R}\ R_1)$ and $C_1 \xrightarrow{\alpha}_{\Omega_P} C_1'$, by *Def$^{\underline{n}}$* of $\mathcal{R}$ we infer

(vii) $R_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R_1'$

(viii) $C_1'\ \equiv \circ \mathcal{R} \circ \equiv\ R_1'$

Hence by *Def$^{\underline{n}}$* of $\mathcal{S}$, $(C_1', R_1') \in \mathcal{S}$. Let us analyse the structure of statement (vii) *i.e.*, $R_1 \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R_1'$. By definition 5.3.2, this implies that the weak transition in (vii) was derived from a transition sequence adhering to one of two forms, depending on α:

- $R_1(\xrightarrow{\tau}_{\Omega_P})^* R_1'$ if $\alpha = \tau$
- $R_1(\xrightarrow{\tau}_{\Omega_P})^* X \xrightarrow{\alpha}_{\Omega_P} Y(\xrightarrow{\tau}_{\Omega_P})^* R_1'$ if α is an external action.

The first transition can be restated as

$$R_1 \xrightarrow{\tau}_{\Omega_P} X_1 \xrightarrow{\tau}_{\Omega_P} X_2 \xrightarrow{\tau}_{\Omega_P} ... \xrightarrow{\tau}_{\Omega_P} X_n \xrightarrow{\tau}_{\Omega_P} R_1' \quad ...(ix)$$

where $X_1 ... X_n$ are intermediate configurations. Given that $R_1 \xrightarrow{\tau}_{\Omega_P} X_1$ (from (ix)) and $R_1 \equiv R$ (from (iv)), by corollary 5.3.1 $((R \xrightarrow{\tau}_{\Omega_P} Y_1) \wedge (X_1 \equiv Y_1))$ *...(x)*.

Once more by corollary 5.3.1, given that $X_1 \xrightarrow{\alpha}_{\Omega_P} X_2$ (from (ix)) and $X_1 \equiv Y_1$ (from (x)) this implies $((Y_1 \xrightarrow{\tau}_{\Omega_P} Y_2) \wedge (X_2 \approx Y_2))$.

This reasoning can be extended throughout (ix), obtaining a new transition sequence of the form

$$R \xrightarrow{\tau}_{\Omega_P} Y_1 \xrightarrow{\tau}_{\Omega_P} Y_2 \xrightarrow{\tau}_{\Omega_P} .. \xrightarrow{\tau}_{\Omega_P} Y_n \xrightarrow{\tau}_{\Omega_P} R' ...(xi)$$

such that $R_1' \equiv \Omega_P(R')$. Transition sequence (xi) represents weak filtered action $R \overset{\widehat{\tau}}{\Rightarrow}_{\Omega_P} R'$. Note that we conveniently name the last system in sequence (xi) to R', to represent the last system in a sequence of transition derivations. Moreover, it is proven that R' is structurally equivalent to R_1' by virtue of corollary 5.3.1, derived information $X_n \equiv Y_n$ and transition $X_n \xrightarrow{\tau}_{\Omega_P} R_1'$ (from (ix)).

Analogously, if α is an external action, then we would be able to conclude that $R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R'$ *...(xii)* and $R_1{}' \equiv R'$ *...(xiii)*. This proves that both (xii) and (xiii) hold irrespective of the value of α.

By (vi), (viii) and (xiii) we have so far proven $C' \equiv C_1{}' \; \mathcal{S} \; R_1{}' \equiv R'$. Let us analyse this statement further.

$$\begin{array}{l}
C' \equiv C_1{}' \; \mathcal{S} \; R_1{}' \equiv R' \\
\Leftrightarrow \{\mathit{Def}^{\underline{n}} \text{ of } \mathcal{S}\} \\
\exists A, B : \textsc{Conf.}\; C' \equiv C_1{}' \equiv A \; \mathcal{R} \; B \equiv R_1{}' \equiv R' \\
\Leftrightarrow \{\text{Transitivity,symmetry of} \equiv \text{twice}\} \\
\exists A, B : \textsc{Conf.}\; C' \equiv C_1{}' \; \mathcal{R} \; R_1{}' \equiv R' \\
\Leftrightarrow \{\mathit{Def}^{\underline{n}} \text{ of } \mathcal{S}\} \\
(C', R') \in \mathcal{S}
\end{array}$$

Through the above reasoning and (xii) we have thus proven that $\exists R' : \textsc{Conf.}(R \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} R') \wedge (C' \; \mathcal{S} \; R')$, proving the statement we set out to prove.

Proof of (2)**:**

In order to prove that $\approx_{\Omega_P}$ is a bisimilarity up to structural equivalence, we would also have to prove statement (2). However, its proof is analogous to the one given above.

□

G. Monitoring Does Not Affect Computation

To Prove:

$$(\delta \triangleright S) \approx_{\Omega_P} (\delta \triangleright S_P)$$

Proof. Proof by coinduction, while inductively exploiting the structure of S.

We are interested in matching the *process computation* of a system and its projection. However, given the problems faced when proving the above statement due to the syntactic generation of persistent traces during process execution, we choose to ignore generated traces (at a syntactic level), as long as they match on both sides (see section 5.4). The required statement to prove therefore becomes

$$(\delta \triangleright (S \parallel T)) \approx_{\Omega_P} (\delta \triangleright (S_P \parallel T))$$

To this effect, we define relation $\mathcal{R}$ *s.t.*

- $(\delta \triangleright S)\ \mathcal{R}\ (\delta \triangleright S_P)$
- $(\delta \triangleright S)\ \mathcal{R}\ (\delta \triangleright S')$ implies $(\delta \triangleright (S \parallel T))\ \mathcal{R}\ (\delta \triangleright (S' \parallel T))$

and go on to prove $\mathcal{R} \subseteq \approx_{\Omega_P}$. The desired original result follows from the first clause. Relation $\mathcal{R}$ is inductively defined; its base elements are obtained by projecting the terminal processes (*i.e.,* obtained from the first clause). Moreover, both the first and the second clause contribute to the generation of the remaining pairs in $\mathcal{R}$. The proof proceeds by induction on the pair $(\delta \triangleright S, \delta \triangleright S'') \in \mathcal{R}$, thus proving that configurations $\delta \triangleright S$, $\delta \triangleright S''$ adhere to the transfer property. We are hence required to prove that if $\delta \triangleright S\ \mathcal{R}\ \delta \triangleright S''$, then the following two statements hold.

$$\textit{(1)}\ (\delta \triangleright S \xrightarrow{\alpha}_{\Omega_P} C') \Rightarrow ((\delta \triangleright S'' \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$$
$$\textit{(2)}\ (\delta \triangleright S'' \xrightarrow{\alpha}_{\Omega_P} C') \Rightarrow ((\delta \triangleright S \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$$

Base Cases:

- $S = k[\![\mathsf{stop}]\!],\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Proof of (1):

  We know *(i)* $(\delta \rhd k[\![\mathsf{stop}]\!])\ \mathcal{R}\ (\delta \rhd k[\![\mathsf{stop}]\!])$, *(ii)* $\delta \rhd k[\![\mathsf{stop}]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \rhd k[\![\mathsf{stop}]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$.

  Given the structure of configuration $\delta \rhd k[\![\mathsf{stop}]\!]$, we know that no transition exists such that $\delta \rhd k[\![\mathsf{stop}]\!] \xrightarrow{\alpha} C'$, since stop is a terminal process. By extension, this implies that $\delta \rhd k[\![\mathsf{stop}]\!] \xrightarrow{\alpha}_{\Omega_P} C'$ is a false statement. This renders the antecedent of (1) in this case, false, rendering the whole statement vacuously true as a result.

  Proof of (2):

  Analogous to above argument.

All other base cases happen to be terminal processes, implying that proof of statements (1) and (2) in these cases happen to follow the above argument. More specifically, given that terminal processes cannot perform another computational step, truth of the required statements becomes vacuously true.

- $S = k\{\![\mathsf{ok}]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![\mathsf{fail}]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![\mathsf{stop}]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

Inductive Hypothesis:

- $((\delta \rhd S\ \mathcal{R}\ \delta \rhd S'') \wedge (\delta \rhd S \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((\delta \rhd S'' \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$
- $((\delta \rhd S\ \mathcal{R}\ \delta \rhd S'') \wedge (\delta \rhd S'' \xrightarrow{\alpha}_{\Omega_P} C')) \Rightarrow ((\delta \rhd S \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$

Inductive Cases:

- $S = k[\![c!\bar{x}.P]\!],\;\; S'' = S_P = k[\![c!\bar{x}.P_P]\!]$

  Proof of (1):

  We know *(i)* $(\delta \triangleright k[\![c!\bar{x}.P]\!])\ \mathcal{R}\ (\delta \triangleright k[\![c!\bar{x}.P_P]\!])$, *(ii)* $\delta \triangleright k[\![c!\bar{x}.P]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \triangleright k[\![c!\bar{x}.P_P]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C'\,\mathcal{R}\,C''))$.

  Given the structure of configuration $\delta \triangleright k[\![c!\bar{x}.P]\!]$, the only valid transition at this stage is

  $$\delta \triangleright k[\![c!\bar{x}.P]\!] \xrightarrow{c!v_{\langle p:k\rangle}} \mathsf{inc}(\delta, k) \triangleright (k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]) \quad \text{by rule (Out}_P\text{)}$$
  $$\delta \triangleright k[\![c!\bar{x}.P]\!] \xrightarrow{c!\bar{v}_{\langle k\rangle}}_{\Omega_P} \mathsf{inc}(\delta, k) \triangleright (k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]) \quad \text{by } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

  We can derive a matching transition for $k[\![c!\bar{x}.P_P]\!]$ *i.e.,*

  $$\delta \triangleright k[\![c!\bar{x}.P_P]\!] \xrightarrow{c!v_{\langle p:k\rangle}} \mathsf{inc}(\delta, k) \triangleright (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]) \quad \text{by rule (Out}_P\text{)}$$
  $$\delta \triangleright k[\![c!\bar{x}.P_P]\!] \xrightarrow{c!\bar{v}_{\langle k\rangle}}_{\Omega_P} \mathsf{inc}(\delta, k) \triangleright (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]) \quad \text{by } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

  which satisfies (iii), since the transitions match, and $(\mathsf{inc}(\delta, k) \triangleright (k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]),\ \mathsf{inc}(\delta, k) \triangleright (k[\![P_P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!])) \in \mathcal{R}$ since $(k[\![P]\!])_P = k[\![P_P]\!]$, with the counter states and generated traces matching.

  Proof of (2):

  Analogous to the first case.

- $S = k[\![c?\bar{x}.P]\!],\;\; S'' = S_P = k[\![c?\bar{x}.P_P]\!]$

  Proof of (1):

  We know *(i)* $(\delta \triangleright k[\![c?\bar{x}.P]\!])\ \mathcal{R}\ (\delta \triangleright k[\![c?\bar{x}.P_P]\!])$, *(ii)* $\delta \triangleright k[\![c?\bar{x}.P]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \triangleright k[\![c?\bar{x}.P_P]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C'\,\mathcal{R}\,C''))$.

  Given the structure of configuration $\delta \triangleright k[\![c?\bar{x}.P]\!]$, the only valid transition at this stage is

  $$\delta \triangleright k[\![c?\bar{x}.P]\!] \xrightarrow{c?v_{\langle p:k\rangle}} \delta \triangleright k[\![P\{^{\bar{v}}/_{\bar{x}}\}]\!] \quad \text{by rule (In}_P\text{)}$$
  $$\delta \triangleright k[\![c?\bar{x}.P]\!] \xrightarrow{c?\bar{v}_{\langle k\rangle}}_{\Omega_P} \delta \triangleright k[\![P\{^{\bar{d}}/_{\bar{x}}\}]\!] \quad \text{by } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

We can derive a matching transition for $k[\![c?\bar{x}.P_P]\!]$ *i.e.,*

$\delta \triangleright k[\![c?\bar{x}.P_P]\!] \xrightarrow{c?v_{\langle p:k\rangle}} \delta \triangleright k[\![P_P\{^{\bar{v}}/_{\bar{x}}\}]\!]$ by rule (In$_P$)
$\delta \triangleright k[\![c?\bar{x}.P_P]\!] \xrightarrow{c?\bar{v}_{\langle k\rangle}}_{\Omega_P} \delta \triangleright k[\![P_P\{^{\bar{v}}/_{\bar{x}}\}]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

which satisfies (iii), since the transitions match, and $(\delta \triangleright k[\![P\{^{\bar{d}}/_{\bar{x}}\}]\!], \delta \triangleright k[\![P_P\{^{\bar{v}}/_{\bar{x}}\}]\!]) \in \mathcal{R}$ since $(k[\![P]\!])_P = k[\![P_P]\!]$, as well as by virtue of lemma $(P\sigma)_P = P_p\sigma$ (see appendix D).

Proof of (2):

Analogous to the first case.

- $S = k[\![\mathsf{new}\, c.P]\!],\ \ S'' = S_P = k[\![\mathsf{new}\, c.P_P]\!]$

Proof of (1):

We know *(i)* $(\delta \triangleright k[\![\mathsf{new}\, c.P]\!])\ \mathcal{R}\ (\delta \triangleright k[\![\mathsf{new}\, c.P_P]\!])$, *(ii)* $\delta \triangleright k[\![\mathsf{new}\, c.P]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \triangleright k[\![\mathsf{new}\, c.P_P]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C'\,\mathcal{R}\,C''))$.

Given the structure of configuration $\delta \triangleright k[\![\mathsf{new}\, c.P]\!]$, the only valid transition at this stage is

$\delta \triangleright k[\![\mathsf{new}\, c.P]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright \mathsf{new}\, c.k[\![P]\!]$ by rule (Exp$_P$)
$\delta \triangleright k[\![\mathsf{new}\, c.P]\!] \xrightarrow{\tau}_{\Omega_P} \delta \triangleright \mathsf{new}\, c.k[\![P]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

We can derive a matching transition for $k[\![\mathsf{new}\, c.P_P]\!]$ *i.e.,*

$\delta \triangleright k[\![\mathsf{new}\, c.P_P]\!] \xrightarrow{\tau_{\langle p:k,k\rangle}} \delta \triangleright \mathsf{new}\, c.k[\![P_P]\!]$ by rule (Exp$_P$)
$\delta \triangleright k[\![\mathsf{new}\, c.P_P]\!] \xrightarrow{\tau}_{\Omega_P} \delta \triangleright \mathsf{new}\, c.k[\![P_P]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

which satisfies (iii), since the transitions match, and $(\delta \triangleright \mathsf{new}\, c.k[\![P]\!], \delta \triangleright \mathsf{new}\, c.k[\![P_P]\!]) \in \mathcal{R}$ since $(\mathsf{new}\, c.k[\![P]\!])_P = \mathsf{new}\, c.k[\![P_P]\!]$.

Proof of (2):

Analogous to the first case.

- $S = k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!],\ \ S'' = S_P = k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P_P\ \mathsf{else}\ Q_P]\!]$

Proof of (1):

Given the structure of configuration $k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]$, there exist two possible transitions which this configuration can take.

– By $\textsc{Eq}_P$:

$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P]\!] \quad \text{by } \textsc{Eq}_P$$
$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P]\!] \quad \text{by } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

A matching transition can be derived for $k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P_P\ \mathsf{else}\ Q_P]\!]$ *i.e.*,

$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P_P\ \mathsf{else}\ Q_P]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P_P]\!] \quad \text{by rule } (\textsc{Eq}_P) \text{ and } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

proving the required statement in this case, since the transitions match, and $(\delta \triangleright k[\![P]\!], \delta \triangleright k[\![P_P]\!]) \in \mathcal{R}$ by $\mathit{Def}^{\underline{n}}$ of system projection.

– By $\textsc{Neq}_P$:

$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![Q]\!] \quad \text{by } \textsc{Neq}_P$$
$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![Q]\!] \quad \text{by } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

A matching transition can be derived for $k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P_P\ \mathsf{else}\ Q_P]\!]$ *i.e.*,

$$\delta \triangleright k[\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ P_P\ \mathsf{else}\ Q_P]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![Q_P]\!] \quad \text{by rule } (\textsc{Neq}_P) \text{ and } \mathit{Def}^{\underline{n}} \text{ of } \rightarrow_{\Omega_P}$$

proving the required statement in this case, since the transitions match, and $(\delta \triangleright k[\![Q]\!], \delta \triangleright k[\![Q_P]\!]) \in \mathcal{R}$ by $\mathit{Def}^{\underline{n}}$ of system projection.

We have hence proven validity of the required statement in both possible cases.

Proof of (2):

Analogous to the first case.

- $S = k[\![P \parallel Q]\!],\ \ S'' = S_P = k[\![P_P \parallel Q_P]\!]$

Proof of (1):

Given the structure of configuration $\delta \triangleright k[\![P \parallel Q]\!]$, the only valid transition at this stage is

$\delta \triangleright k[\![P \parallel Q]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P]\!] \parallel \delta \triangleright k[\![Q]\!]$ by rule (Split$_P$)
$\delta \triangleright k[\![P \parallel Q]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P]\!] \parallel \delta \triangleright k[\![Q]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

We can derive a matching transition for $k[\![P_P \parallel Q_P]\!]$ *i.e.*,

$\delta \triangleright k[\![P_P \parallel Q_P]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P_P]\!] \parallel \delta \triangleright k[\![Q_P]\!]$ by rule (Split$_P$)
$\delta \triangleright k[\![P_P \parallel Q_P]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P_P]\!] \parallel \delta \triangleright k[\![Q_P]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

which satisfies our required statement, since the transitions match, and $(\delta \triangleright k[\![P]\!] \parallel \delta \triangleright k[\![Q]\!], \delta \triangleright k[\![P_P]\!] \parallel \delta \triangleright k[\![Q_P]\!]) \in \mathcal{R}$ by $Def^{\underline{n}}$ of system projection.

Proof of (2):

Analogous to the first case.

- $S = k[\![*P]\!],\ S'' = S_P = k[\![*(P_P)]\!]$

Proof of (1):

Given the structure of configuration $\delta \triangleright k[\![*P]\!]$, the only valid transition at this stage is

$\delta \triangleright k[\![*P]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P \parallel *P]\!]$ by rule (Rec$_P$)
$\delta \triangleright k[\![*P]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P \parallel *P]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

We can derive a matching transition for $k[\![*(P_P)]\!]$ *i.e.*,

$\delta \triangleright k[\![*(P_P)]\!] \xrightarrow{\tau\langle p:k,k\rangle} \delta \triangleright k[\![P_P \parallel *(P_P)]\!]$ by rule (Rec$_P$)
$\delta \triangleright k[\![*(P_P)]\!] \overset{\tau}{\rightarrow}_{\Omega_P} \delta \triangleright k[\![P_P \parallel *(P_P)]\!]$ by $Def^{\underline{n}}$ of $\rightarrow_{\Omega_P}$

which satisfies our required statement, since the transitions match, and $(\delta \triangleright k[\![P \parallel *P]\!], \delta \triangleright k[\![P_P \parallel *(P_P)]\!]) \in \mathcal{R}$ by $Def^{\underline{n}}$ of system projection.

Proof of (2):

Analogous to the first case.

- $S = k\{\![\mathsf{go}\ u.M]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

Proof of (1):

We know *(i)* $(\delta \triangleright k\{\!\![\mathsf{go}\ l.M]\!\!\}^n)\ \mathcal{R}\ (\delta \triangleright k[\![\mathsf{stop}]\!])$, *(ii)* $\delta \triangleright k\{\!\![\mathsf{go}\ l.M]\!\!\}^n \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \triangleright k[\![\mathsf{stop}]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$.

On inspection of the definition of filter function Ω_P, it is apparent that any monitor action is pruned by the function. Hence, any transition emanating from $\delta \triangleright k\{\!\![\mathsf{go}\ l.M]\!\!\}^n$ is later ignored by Ω_P, implying falsity of statement (ii). Hence, the antecedent of (1) is rendered false, proving statement (1) to be vacuously true.

Proof of (2):

We know *(i)* $(\delta \triangleright k\{\!\![\mathsf{go}\ l.M]\!\!\}^n)\ \mathcal{R}\ (\delta \triangleright k[\![\mathsf{stop}]\!])$, *(ii)* $\delta \triangleright k[\![\mathsf{stop}]\!] \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $((\delta \triangleright k[\![\mathsf{go}\ k.M]\!] \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C''))$.

Given the structure of configuration $\delta \triangleright k[\![\mathsf{stop}]\!]$, we know that no transition exists such that $\delta \triangleright k[\![\mathsf{stop}]\!] \xrightarrow{\alpha} C'$, since stop is a terminal process. By extension, this implies that $\delta \triangleright k[\![\mathsf{stop}]\!] \xrightarrow{\alpha}_{\Omega_P} C'$ is a false statement. This renders the antecedent of (2) in this case, false, rendering the whole statement vacuously true as a result.

Proof for the remaining instances when $S = k\{\!\![M]\!\!\}^n$, as well as when $S = k[\![\mathbf{t}(c, \bar{v}, n)]\!]$ are analogous to the above case, since in all instances (i) the result of projection is $k[\![\mathsf{stop}]\!]$, and (ii) any transition emanating from (these instances of) S are later pruned by Ω_P.

- $S = k\{\!\![u?\bar{x}.M]\!\!\}^n,\ \ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\!\![u!\bar{v}.M]\!\!\}^n,\ \ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\!\![\mathsf{new}\, c.M]\!\!\}^n,\ \ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\!\![\mathsf{if}\ \bar{u} = \bar{v}\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\!\}^n,\ \ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![M \parallel N]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![{*}M]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![\mathsf{setC}(l).M]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k\{\![\mathbf{m}(c, x, k).M]\!\}^n,\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = k[\![\mathbf{t}(c, \bar{v}, n)]\!],\ S'' = S_P = k[\![\mathsf{stop}]\!]$

  Analogous to previous argument.

- $S = S_1 \parallel S_2,\ S'' = (S_1)_P \parallel (S_2)_P.$

Proof of (1):

We know *(i)* $(\delta \rhd S_1 \parallel S_2)\mathcal{R}(\delta \rhd (S_1)_P \parallel (S_2)_P)$, *(ii)* $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $(\delta \rhd (S_1)_P \parallel (S_2)_P \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C'\mathcal{R}C'')$.

Statement $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha}_{\Omega_P} C'$ was derived from an unfiltered transition of the form $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha'} C'$ *s.t.* $\Omega_P(\alpha') = \alpha$. Moreover, we can also infer that α' (and by extension, α) is a process action, since Ω_P successfully mapped the original action (*i.e.*, without pruning it). Conversely, had α' been a monitor or trace action, transition (ii) would have been pruned by Ω_P after filtration. Hence, given the structure of transition $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha'} C'$ and knowledge that α' is a process action, we infer that this latter transition could have been derived using one of six rules:

– By CNTX$_2$:

The above transition hence takes the form $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha'} \delta \rhd S_1' \parallel S_2$. By rule CNTX$_2$ we also deduce that this transition was inferred further from another transition $\delta \rhd S_1 \xrightarrow{\alpha'} \delta \rhd S_1'$. Given that Ω_P is a function, applying $\Omega_P(\alpha')$ must return the same α. Hence, by *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$

we obtain $\delta \triangleright S_1 \xrightarrow{\alpha}_{\Omega_P} \delta \triangleright S_1{}'$.

By the definition of $\mathcal{R}$, we know that S_1 and its projection $(S_1)_P$ are in $\mathcal{R}$ *i.e.,* $S_1\ \mathcal{R}\ (S_1)_P$. Hence, given this statement and transition $\delta \triangleright S_1 \xrightarrow{\alpha}_{\Omega_P} \delta \triangleright S_1{}'$ by the IH we obtain *(iv)* $\delta \triangleright (S_1)_P \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'''$, *(v)* $\delta \triangleright S_1{}'\ \mathcal{R}\ C'''$. Let us assume that $C''' = \delta \triangleright ((S_1)_P)'$ *...(vi)*. Irrespective of whether α is an external or internal action, weak transition (iv) is represented by a transition sequence of the form

$$\delta \triangleright (S_1)_P \xrightarrow{\tau}_{\Omega_P} C_i \xrightarrow{\tau}_{\Omega_P} C_{ii} \xrightarrow{\tau}_{\Omega_P} ...C_j \xrightarrow{\alpha} C_{j+1} \xrightarrow{\tau}_{\Omega_P} ... \xrightarrow{\tau}_{\Omega_P} \delta \triangleright ((S_1)_P)'$$

By the *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ numerous times on the above transition sequence we obtain

$$\delta \triangleright (S_1)_P \xrightarrow{\alpha_1} C_i \xrightarrow{\alpha_2} C_{ii} \xrightarrow{\alpha_3} ...C_j \xrightarrow{\alpha'} C_{j+1} \xrightarrow{\alpha_{j+1}} ... \xrightarrow{\alpha_n} \delta \triangleright ((S_1)_P)'$$

where $\alpha_1...\alpha_n$ represent tagged process τ actions. We next apply rule Cntx$_2$ on each transition in the above sequence, adding system $(S_2)_P$ in each case. Moreover, we apply filter Ω_P once more, giving rise to weak transition

$$\delta \triangleright ((S_1)_P \parallel (S_2)_P) \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} \delta \triangleright (((S_1)_P)' \parallel (S_2)_P)$$

which gives us the required matching transition.

We are left to prove that the residual configurations are in $\mathcal{R}$ in order to prove the required statement in this case. By (v) we know that $\delta \triangleright S_1{}'\ \mathcal{R}\ \delta \triangleright ((S_1)_P)'$. Hence, from this we infer that $(\delta \triangleright S_1{}' \parallel S_2,\ \delta \triangleright (((S_1)_P)' \parallel (S_2)_P)) \in \mathcal{R}$, proving statement (iii).

– By Cntx$_3$: Analogous to the above case.

– By Com$_1$:

We hence infer that $\delta \triangleright S_1 \parallel S_2 \xrightarrow{\tau_{\langle p:k,l\rangle}} \delta' \triangleright \mathsf{new}\,\tilde{b}.(S_1{}' \parallel S_2{}')$. Note that the action modality is set to p, due to previous inference that α' must be a process action. By rule Com$_1$ we also deduce that this transition was inferred from two further transitions *(iv)* $\delta \triangleright S_1 \xrightarrow{(\tilde{b})c!\tilde{d}_{\langle p:k\rangle}} \delta' \triangleright S_1{}'$ *(v)* $\delta \triangleright S_2 \xrightarrow{c?\tilde{d}_{\langle p:l\rangle}} \delta \triangleright S_2{}'$. By *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ twice we obtain transitions $\delta \triangleright S_1 \xrightarrow{(\tilde{b})c!\tilde{d}_{\langle k\rangle}}_{\Omega_P} \delta' \triangleright S_1{}'$ and $\delta \triangleright S_2 \xrightarrow{c?\tilde{d}_{\langle l\rangle}} \delta \triangleright S_2{}'$.

By *Def$^{\underline{n}}$* of $\mathcal{R}$, we know that $(S_1, (S_1)_P) \in \mathcal{R}$ and $(S_2, (S_2)_P) \in \mathcal{R}$. By the IH twice we hence infer *(vi)* $\delta \triangleright (S_1)_P \overset{\widehat{(\tilde{b})c!\tilde{d}_{\langle k\rangle}}}{\Rightarrow}_{\Omega_P} \delta' \triangleright ((S_1)_P)'$, *(vii)* $\delta' \triangleright S_1{}'\ \mathcal{R}\ \delta' \triangleright ((S_1)_P)'$, and *(viii)*

$\delta \rhd (S_2)_P \overset{\widehat{c?\bar{d}_{\langle l \rangle}}}{\Rightarrow} \delta \rhd ((S_2)_P)'$, *(ix)* $\delta \rhd S_2' \mathcal{R} \delta \rhd ((S_2)_P)'$. Using analogous reasoning to that shown for rule Cntx$_2$, the external actions within weak transitions (vi) and (viii) interact to give us another weak transition $\delta \rhd (S_1)_P \parallel (S_2)_P \overset{\widehat{\tau_{\langle p:k,l \rangle}}}{\Rightarrow} \delta' \rhd (\mathsf{new}\,\bar{b}.(((S_1)_P)' \parallel ((S_2)_P)'))$ which gives us the required matching transition. Moreover, by (vii) and (ix) we also infer that $((\mathsf{new}\,\bar{b}.(S_1' \parallel S_2')), (\mathsf{new}\,\bar{b}.(((S_1)_P)' \parallel ((S_2)_P)'))) \in \mathcal{R}$, thus proving statement (iii) in this case.

– By Com$_2$: Analogous to the above case.

– By Mon$_1$:

Rule Mon$_1$ dictates that some trace within S_1 and a monitor within S_2 interact, resulting in a trace action. However, we know that α' must be a process action, implying that transition $\delta \rhd S_1 \parallel S_2 \xrightarrow{\alpha'} C'$ could not have been inferred using rule Mon$_1$.

– By Mon$_2$: Analogous to the above case.

Proof of (2):

Analogous to previous argument.

- $S = \mathsf{new}\,c.S_1, \;\; S'' = \mathsf{new}\,c.(S_1)_P$.

Proof of (1):

We know *(i)* $(\delta \rhd \mathsf{new}\,c.S_1)\mathcal{R}(\delta \rhd \mathsf{new}\,c.(S_1)_P)$, *(ii)* $\delta \rhd \mathsf{new}\,c.S_1 \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $(\delta \rhd \mathsf{new}\,c.(S_1)_P \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C'')$.

Transition $\delta \rhd \mathsf{new}\,c.S_1 \xrightarrow{\alpha}_{\Omega_P} C'$ was derived from an unfiltered transition of the form $\delta \rhd \mathsf{new}\,c.S_1 \xrightarrow{\alpha'} C'$ *s.t.* $\Omega_P(\alpha') = \alpha$. Moreover, we can also infer that α' (and by extension, α) is a process action, since Ω_P successfully mapped the original action (*i.e.*, without pruning it). Conversely, had α' been a monitor or trace action, transition (ii) would have been pruned by Ω_P after filtration. Hence, given the structure of transition $\delta \rhd \mathsf{new}\,c.S_1 \xrightarrow{\alpha'} C'$ and knowledge that α' is a process action, we infer that this latter transition could have been derived using one of two rules:

– By Open$_s$:

The above transition hence takes the form $\delta \triangleright \mathsf{new}\, c.S_1 \xrightarrow{(c,\bar{b})c!\bar{d}_{\langle p:k\rangle}} \delta' \triangleright S_1{}'$. Note that the action modality is set to p, due to previous inference that α' must be a process action. By rule OPEN$_s$ we also deduce that this transition was inferred from another transition of the form $\delta \triangleright \mathsf{new}\, c.S_1 \xrightarrow{(\bar{b})c!\bar{d}_{\langle p:k\rangle}} \delta' \triangleright S_1{}'$. By *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ we obtain transition $\delta \triangleright \mathsf{new}\, c.S_1 \xrightarrow{(\bar{b})c!\bar{d}_{\langle k\rangle}}_{\Omega_P} \delta' \triangleright S_1{}'$.

By *Def$^{\underline{n}}$* of $\mathcal{R}$, we know that $(S_1, (S_1)_P) \in \mathcal{R}$. By the IH we hence infer *(iv)* $\delta \triangleright ((S_1)_P) \overset{\widehat{(\bar{b})c!\bar{d}_{\langle k\rangle}}}{\Longrightarrow}_{\Omega_P} \delta' \triangleright ((S_1)_P)'$ *(v)* $(\delta' \triangleright S_1{}', \delta' \triangleright ((S_1)_P)') \in \mathcal{R}$. Using analogous reasoning to before, we can unravel weak transition (iv) to its sequence of unfiltered transitions, export additional channel b to the output label as necessary, and finally re-filter each transition, resulting in a weak filtered transition of the form $\delta \triangleright \mathsf{new}\, c.((S_1)_P) \overset{\widehat{(c,\bar{b})c!\bar{d}_{\langle k\rangle}}}{\Longrightarrow}_{\Omega_P} \delta' \triangleright ((S_1)_P)'$, which gives us the required transition. Moreover, by (v) we already know that the residual configurations are in $\mathcal{R}$, thus proving the required statement (iii).

– By CNTX$_1$:

The above transition hence takes the form $\delta \triangleright \mathsf{new}\, c.S_1 \xrightarrow{\alpha'} \delta' \triangleright \mathsf{new}\, c.S_1{}'$. By rule CNTX$_1$ we also deduce that this transition was inferred from another transition of the form $\delta \triangleright S_1 \xrightarrow{\alpha'} \delta' \triangleright S_1{}'$. By *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ we obtain transition $\delta \triangleright S_1 \xrightarrow{\alpha}_{\Omega_P} \delta' \triangleright S_1{}'$.

By *Def$^{\underline{n}}$* of $\mathcal{R}$, we know that $(S_1, (S_1)_P) \in \mathcal{R}$. By the IH we hence infer *(iv)* $\delta \triangleright ((S_1)_P) \overset{\widehat{\alpha}}{\Longrightarrow}_{\Omega_P} \delta' \triangleright ((S_1)_P)'$ *(v)* $(\delta' \triangleright S_1{}', \delta' \triangleright ((S_1)_P)') \in \mathcal{R}$. Using analogous reasoning to before, we can unravel weak transition (iv) to its sequence of unfiltered transitions, add channel c on each intermediate transition through numerous applications of rule CNTX$_1$, and finally re-filter each transition, thus resulting in a weak filtered transition of the form $\delta \triangleright \mathsf{new}\, c.((S_1)_P) \overset{\widehat{(c,\bar{b})c!\bar{d}_{\langle k\rangle}}}{\Longrightarrow}_{\Omega_P} \delta' \triangleright ((S_1)_P)'$, which gives us the required transition. Moreover, given truth of (v), we hence infer that $(\delta' \triangleright \mathsf{new}\, c.S_1{}', \delta' \triangleright \mathsf{new}\, c.((S_1)_P)') \in \mathcal{R}$, thus proving the required statement (iii).

Note the impossibility of inferring transition $\delta \triangleright k[\![\mathsf{new}\, c.S_1]\!] \xrightarrow{\alpha'} C'$ using rule OPEN$_t$, since this would require α' to be a trace action, which we have shown to be impossible.

Proof of (2):

Analogous to previous argument.

We next move on the proving veracity of the transfer property for systems placed in parallel with a trace T *i.e.*, proving truth for the second $\mathcal{R}$ clause which allows us to syntactically ignore traces generated by a system and its projection at runtime, as long as they match on both sides.

- $S = S_1 \parallel T,\;\; S'' = S_1{}' \parallel T$, given that $(\delta \rhd S_1)\; \mathcal{R}\; (\delta \rhd S_1{}')$.

Proof of (1):

We know *(i)* $(\delta \rhd S_1 \parallel T)\mathcal{R}(\delta \rhd S_1{}' \parallel T)$, *(ii)* $\delta \rhd S_1 \parallel T \xrightarrow{\alpha}_{\Omega_P} C'$, and are required to prove *(iii)* $(\delta \rhd S_1{}' \parallel T \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'') \wedge (C' \mathcal{R} C'')$.

Clearly, statement $\delta \rhd S_1 \parallel T \xrightarrow{\alpha}_{\Omega_P} C'$ was derived from an unfiltered transition of the form $\delta \rhd S_1 \parallel T \xrightarrow{\alpha'} C'$ *s.t.* $\Omega_P(\alpha') = \alpha$. Moreover, we can also infer that α' (and by extension, α) is a process action, since Ω_P successfully mapped the original action (*i.e.*, without pruning it). Conversely, had α' been a monitor or trace action, transition (ii) would have been pruned by Ω_P after filtration. Hence, given the structure of transition $\delta \rhd S_1 \parallel T \xrightarrow{\alpha'} C'$ and knowledge that α' is a process action, we infer that this latter transition could have been derived using one of two rules:

– By Cntx$_2$:

With the transition under consideration taking the following form $\delta \rhd S_1 \parallel T \xrightarrow{\alpha'} \delta \rhd S_1{}'' \parallel T$. By rule Cntx$_2$ we also deduce that this transition was inferred further from another transition $\delta \rhd S_1 \xrightarrow{\alpha'} \delta \rhd S_1{}''$. Given that Ω_P is a function, applying $\Omega_P(\alpha')$ must return the same α. Hence, by *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ we obtain $\delta \rhd S_1 \xrightarrow{\alpha}_{\Omega_P} \delta \rhd S_1{}''$.

Given that $(\delta \rhd S_1)\; \mathcal{R}\; (\delta \rhd S_1{}')$ and $\delta \rhd S_1 \xrightarrow{\alpha}_{\Omega_P} \delta \rhd S_1{}''$, by the IH we obtain *(iv)* $\delta \rhd S_1{}' \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} C'''$, *(v)* $\delta \rhd S_1{}'' \mathcal{R} C'''$. Let us assume that $C''' = \delta \rhd S_1{}'''$ *...(vi)*. By applying the *Def$^{\underline{n}}$* of $\rightarrow_{\Omega_P}$ numerous times on weak transition (iv) to obtain its unfiltered counterpart, subsequently applying rule (Cntx$_2$) to attach trace T to each configuration within this sequence, and re-applying Ω_P on all (unfiltered) transitions, we obtain weak transition $\delta \rhd S_1{}' \parallel T \overset{\widehat{\alpha}}{\Rightarrow}_{\Omega_P} \delta \rhd S_1{}''' \parallel T$, which gives us the matching transition.

In order to complete the proof, we are required to prove that the residual configurations are in $\mathcal{R}$. By statement (vi) we can rewrite (v) as $\delta \rhd S_1{}'' \mathcal{R} \delta \rhd S_1{}'''$. Hence, by the second clause of $\mathcal{R}$ we can deduce that $\delta \rhd (S_1{}'' \parallel T) \mathcal{R} (\delta \rhd S_1{}''' \parallel T)$, thus completing the proof for (iii).

– By Cntx$_3$:

Rule (Cntx$_3$) dictates that the rightmost system in configuration $\delta \rhd S_1 \parallel T$ must have performed the action. However, given that a trace T can only produce actions tagged with modality t, and from the definition of Ω_P we know that all non-process actions are pruned, then we can safely conclude that $\delta \triangleright S_1 \parallel T \xrightarrow{\alpha'} C'$ could not have been derived by ($\textsc{Cntx}_3$).

Note that rules $\textsc{Com}_1$ and $\textsc{Com}_2$ are eliminated from the offset, since it is impossible for a trace T to communicate using either process or monitor actions. Rules $\textsc{Mon}_1$ and $\textsc{Mon}_2$ are also disallowed, since we inferred that α' must be a process action. Hence, it is impossible for $\delta \triangleright S_1 \parallel T \xrightarrow{\alpha'} C'$ to have been inferred using these latter two rules either.

Proof of (2):

Analogous to previous argument.

□

Bibliography

[1] Charlie Abela, Aaron Calafato, and Gordon J. Pace. Extending wise with contract management. In *WICT 2010*.

[2] Gul Agha. *Actors: A Model of Concurrent Computation*. PhD thesis, MIT, 1986.

[3] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In *Proceedings of the fifth ACM symposium on Operating systems principles*, SOSP '75, pages 67–74, New York, NY, USA, 1975. ACM.

[4] Dr. Thomas Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. 2010.

[5] Bowen Alpern and Fred B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[7] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[8] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, May 2005.

[9] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. pages 55–96, 1993.

[10] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[12] Robert F. Barnes. Interval temporal logic: A note. Springer, November 1981.

[13] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. pages 44–57. Springer, 2004.

[14] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[15] Eric Bodden, Laurie Hendren, Patrick Lam, Ondrej Lhoták, and Nomair Naeem. Collaborative runtime verification with tracematches. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*, pages 22–37. Springer Berlin / Heidelberg, 2007.

[16] Marco Brambilla, Stefano Ceri, Piero Fraternali, and Ioana Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15:360–409, 2006.

[17] Zhou Chaochen, C.A.R Hoare, and Anders P.Ravn. A calculus of durations. *Oxford University Computing Laboratory, Programming Research Group*, 1991.

[18] David Chappell. *Enterprise Service Bus*. O'Reilly Media, June 2004.

[19] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[20] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer Berlin / Heidelberg, 2005.

[21] Gordon J. Pace Christian Colombo and Gerardo Schneider. Resource-bounded runtime verification of java programs with real-time properties. Technical report, Department of Computer Science, University of Malta, 2009.

[22] Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.

[23] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, The MIT Press Massachusetts Insititute Of Technology Cambridge, Massachusetts 02142, 1999.

[24] Séverine Colin and Leonardo Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, pages 525–555. Springer, 2004.

[25] Christian Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University of Malta, 2008.

[26] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[27] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE Computer Society Press, June 2005.

[28] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, 2003.

[29] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13:222–232, 1987.

[30] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[31] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. 12(5):482–498, 2006.

[32] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[33] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[34] Yliès Falcone. You should better enforce than verify. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 89–105, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[36] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting statistics over runtime executions. In *In Proceedings of Runtime Verification (RV'02) [1]*, pages 36–55. Elsevier, 2002.

[37] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. In *In Proceedings of Runtime Verification (RV'01) [1*, pages 44–60, 2001.

[38] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24:342–361, 1998.

[39] Vijay K. Garg. *Concurrent and Distributed Computing in Java*. John Wiley & Sons, 2004.

[40] Dimitra Giannakopoulou and Klaus Havelund. Runtime analysis of linear temporal logic specifications. Technical report, 2001.

[41] Eugen-Ioan Goriac, Dorel Lucanu, and Grigore Roşu. Automating coinduction with case analysis. In *Twelfth International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2010.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[43] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag.

[44] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6:158–173, August 2004.

[45] Klaus Havelund and Grigore Rosu. Java pathexplorer - a runtime verification tool. In *In The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, page 2001, 2001.

[46] Matthew Hennessy. *A Distributed Pi-Calculus*, chapter 2, pages 10–54. Cambridge University Press, 2007.

[47] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA, 2007.

[48] Matthew Hennessy. *A Distributed Pi-Calculus*, chapter 5, pages 124–191. Cambridge University Press, 2007.

[49] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, June 1985.

[50] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[51] Wolfram Kahl, Christopher K. Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In *8th International Conference on Relational Methods in Computer Science, RelMiCS 8*, volume 3929, pages 147–160, 2005.

[52] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[53] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[55] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. In *In Runtime Verification 2001, volume 55 of ENTCS*. Elsevier Science Publishers.

[56] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *ECRTS*, pages 114–122. IEEE Computer Society, 1999.

[57] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Interaction-based runtime verification for systems of systems integration. *Computer Science and Engineering Department, University of California, San Diego USA*, July 2008.

[58] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3:125–143, March 1977.

[59] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[60] Martin Leucker and Christian Schallhart. A brief account of runtime verification, 2008.

[61] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*, page 70, Washington, DC, USA, 1999. IEEE Computer Society.

[62] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006.

[63] J. Lundelius and Jennifer L Welch. Synchronising clocks in a distributed system. Technical report, Cambridge, MA, USA, 1984.

[64] Masoud Mansouri-Samani and Morris Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[65] Patrick Meredith and Grigore Roşu. Runtime verification with the RV system. In *First International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lecture Notes in Computer Science*, pages 136–152. Springer, 2010.

[66] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[67] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[68] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[69] OASIS. *Web Services Business Process Execution Language Version 2.0*, April 2007.

[70] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[71] Gordon Pace, Christian Colombo, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, LNCS 4916. Springer-Verlag, 2008.

[72] Amir Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977.

[73] Paola Quaglia and David Walker. On encoding p-pi in m-pi. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 42–53, London, UK, 1998. Springer-Verlag.

[74] Usa Sammapun and Oleg Sokolsky. Regular expressions for runtime verification. in Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis (ATVA'03), 2003.

[75] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31:15:1–15:41, May 2009.

[76] Davide Sangiorgi and David Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[77] Thomas S.Cook, Doron Drusinksy, and Man-Tak Shing. Specification, validation and run-time moniroting of soa based system-of systems temporal behaviors. In *In System of Systems Engineering (SoSE)*. IEEE Computer Society, 2007.

[78] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *In USENIX Security Symposium*, pages 63–78, 1999.

[79] Koushik Sen, Grigore Roşu, and Gul Agha. Runtime safety analysis of multithreaded programs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 337–346, New York, NY, USA, 2003. ACM.

[80] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. *Software Engineering, International Conference on*, 0:418–427, 2004.

[81] Peter Sewell. Applied pi - a brief tutorial. Technical report, University of Cambridge, Computer Laboratory, July 2000.

[82] A. C. Simpson. *Discrete Mathematics by Example*. McGraw-Hill, 2002.

[83] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[84] Richard W. Stevens. *UNIX Network Programming*. Prentice Hall, October 1997.

[85] Henrik Thane. *Monitoring, Testing and Debugging of distributed real-time systems*. PhD thesis, Mechatronics Laboratory, Department of Machine Design, Royal Institute of Technology, KTH, S-100 44 Stockholm, Sweden, 2000.

[86] Linda Westfall. *The Certified Software Quality Engineer Handbook*. American Society for Quality Press, 2009.

[87] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin Rinard. Runtime checking for program verification. In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 202–213, Berlin, Heidelberg, 2007. Springer-Verlag.

[88] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo, and Insup Lee. Dmac: Distributed monitoring and checking. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 184–201. Springer, 2009.