# Distributed System Contract Monitoring[☆]

Adrian Francalanza, Andrew Gauci, Gordon J. Pace

*Department of Computer Science, University of Malta*

**Abstract**

Runtime verification of distributed systems poses various challenges. A pivotal challenge is the choice of how to distribute the monitors themselves across the system. On one hand, centralised monitoring may result in increased communication overhead and information exposure across locations, while, on the other hand, systems with dynamic topologies and properties are difficult to address using static monitor choreographies. In this paper we present mDPI, a location-aware $\pi$-calculus extension for reasoning about the distributed monitoring scenario. We also define numerous monitoring strategies for a regular expression-based logic, including a novel approach in which monitors migrate to ensure local monitoring. Finally, we present a number of results which emerge from this formalism, justifying our approach.

*Keywords:* runtime verification, distributed systems, migrating monitors, $\pi$-calculus

## 1. Introduction

Distributed systems provide remote services, economise costs by sharing resources, and improve scalability and dependability through replication. This current trend towards system distribution brings forth new challenges to modelling and verification: not only do distributed systems have to be more defensively developed to address new potential points of failure in any of its (non-local) sub-components, but specifications need to cater for additional information, such as locality of data and control. Moreover, existing approaches to system analysis, from testing and debugging to runtime verification and model checking, cannot be applied without substantial adaptation, because of the constraints induced by the distributed nature of the systems under analysis. For instance, in dynamic systems, where access to remote services may be discovered at runtime, one cannot readily extract a closed system for analysis. Similarly, in the case of service contract negotiation, the properties to be monitored may only be known at runtime, meaning that a further degree of dynamicity is required.

Runtime verification[1] is one way of addressing system dependability — by moni-

[☆]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.

[1]The terms *runtime verification*, *runtime monitoring* or sometimes even simply *monitoring* have been used in the literature. We will use these terms interchangeably in this paper.

toring the system's behaviour at runtime and comparing it with its specification, thus enabling the discovery of specification violations and possibly also triggering recovery behaviour. In monolithic systems, where the system resides in a single location, the verifying code typically consists of a new listening subsystem located in the same address-space of the original system. However, in a distributed setting, the choice of location of the verifier is *crucial*, since communication across locations is typically an expensive operation, and may potentially lead to the exposure of sensitive information. Decisions on the locality may also impact on the structure of the verifier. For instance, in order to locally verify different parts of the specification, the verifying code may need to be structured in such a manner so as to be broken down into parts and distributed across locations.

Various alternative solutions have been presented in the literature from fully *orchestrated* solutions, coordinated by a *central monitor* at a specific location, to *choreographed* monitors, *distributed upfront* at different locations. Orchestration is simple and may be adapted directly from the monitoring of monolithic systems. However, it disregards locality of trace analysis, requiring such information to travel across possibly untrusted media; this leads to increased network traffic across remote locations and also unnecessary information exposure. Despite these drawbacks, its simplicity makes it an appealing approach, *e.g.,* in [1], where web-service compositions are monitored in an orchestrated fashion. Choreography-based monitoring [2, 3, 4, 5, 6] can mitigate orchestration limitations However, a characteristic shortcoming with choreography is that local monitors have to be instrumented *statically*, before the analysis starts; this inflexibility prohibits choreographed monitors from adequately dealing with dynamic systems, where network locations come and go, and dynamic properties[2].

In [7] we proposed a novel approach to distributed system monitoring centered around *runtime migration*: monitors are installed locally but then allowed to migrate from one location to another when the need arises. Since monitors can be located where the immediate confidential traces reside, as in choreography-based approaches, migrating monitors avoid unnecessary data exposure. Moreover, they can also migrate to locations determined at runtime, enabling them to preserve local monitoring despite dynamic topologies.

Although there is a substantial body of work on the use of runtime verification for distributed systems e.g. [6, 8], much of the work revolves around tool development and issues of efficiency. We are not aware of any major work attempting to address this area from a more theoretical perspective, answering questions such as expressivity of and equivalence between different monitoring approaches.

In this paper, we present a unified formal framework for studying different monitoring strategies for distributed systems, thereby allowing for their precise comparison — showing for instance, that two monitoring strategies may produce monitors which are distributed in different ways but behave in an equivalent manner; or that one monitoring strategy exposes less information on global channels than another. We present a location-aware calculus supporting explicit monitoring as a first class entity, whilst

[2]By dynamic properties, we refer to properties which are only known at runtime, such as contracts coming with a service which is discovered and used by a system at runtime.

internalising behavioural traces at the operational level of the calculus (as opposed to a meta-level). In this paper, we focus on what the literature variably refers to as *asynchronous*, *offline* or *passive* monitoring — in which the monitors check the traces generated by the system asynchronrously, thus not inhibiting its progress. It is worth noting, however, that the calculus we present can be used to model *synchronous* (also called *online* or *active*) monitoring, in which the system does not proceed until the monitor processes any event generated.

We show the expressivity of the calculus by using it to model different distributed system monitoring strategies from the literature, including migrating monitoring [7]. We later show how behavioral contracts expressed using regular expressions can be automatically translated into monitors using different monitoring strategies. Finally, we formally prove that (i) the various identified monitoring strategies are behaviourally equivalent (up to the location of monitors); (ii) certain distributed monitoring approaches, including migrating monitors, are safe from eavesdropping in that they guarantee that trace analysis is always performed locally and thus do not broadcast logged information beyond location borders.

This paper is an extended and revised version of [7] with proofs of the main results and the following new contributions: (i) We present a revised labelled transition system with a novel technique for dealing with partial traces, allowing us to work with with open systems; (ii) We justify bisimulation equivalence by proving that it is a congruence with respect to parallel composition (Theorem 1); (iii) We prove that the monitors synthesised from a regular expression using orchestrated, choreographed and migrating translations are equivalent (Theorems 2 and 3); and (iv) We prove that whereas orchestrated translations require remote monitoring, choreographed and migrating monitor translations do not incur such a penalty (Theorems 4 and 5).

The paper is organised as follows. In section 2, we outline the contract monitoring strategies for distributed systems from the literature, as well as the novel migrating monitor approach. We then present the monitoring distributed calculus mDPI in section 3.1, the semantics in section 3.2 and bisimulation proof techniques in section 3.4. The calculus is used to model of monitoring contracts expressed as regular expressions in section 4.1. This is followed by section 4.3, presenting formal comparisons between different monitoring strategies. The work is discussed and compared to existing formalisations of distributed monitoring in section 5, finally concluding in section 6.

## 2. Approaches to Distributed System Monitoring

Distributed systems — made up of autonomous, concurrently executing sub-systems communicating through message passing, each with its local memory — poses new challenges to monitoring, which go beyond those posed by the monitoring of monolithic systems. Cross-border interaction is typically an expensive operation and may take place over an unsafe medium. Moreover, these systems are characterised by the *lack of a global clock* when ordering events across location boundaries. Instead, each sub-system admits a local clock, implying a level of *asynchrony* across locations. The topology of such systems may sometimes change at runtime through the addition of new subsystems or the communication of private channels. Most internet-based and service-oriented systems, peer-to-peer systems and Enterprise Service Bus architectures [9] are instances of such systems.

These characteristics impinge on contract monitoring.[3] For instance, the absence of a global clock prohibits precise monitoring for consequentiality properties which refer to behaviour in different locations [10]. From a monitoring perspective, not only is it important that verification uses limited space and computational resources, but it is now also important that the it does not induce an unreasonable communication overhead, since doing so could disrupt the underlying system's computation. Furthermore, distribution impacts on information locality; subsystem events may contain confidential information which must not be exposed globally across unsafe mediums or locations, thereby requiring local monitoring.

Whether monitored contracts are known at compile time or else become known at runtime affects distributed monitoring. Static contracts, ones which are fully known at compile time, are not always expressive enough for distributed systems with dynamic topologies. Dynamic contracts — ones which are only discovered at runtime, or which are only partially known at compile time — tend to be more appropriate for such systems. They are found, for instance, in intrusion detection [11], where suspicious user behaviour is learnt at runtime, and in systems involving service discovery, where the chosen service may come with a fixed or negotiated contract made known only upon discovery.

## *2.1. Classifying Distributed System Monitoring Approaches*

Various approaches have been proposed for monitoring of distributed systems; from centralised architectures, to statically distributed monitoring approaches and, more recently, the use of mobile monitors. In general, existing approaches for distributed system monitoring can be broadly classified into two categories: *orchestration*-based or *choreography*-based. Orchestration-based approaches relegate monitoring responsibility to a central monitor overhearing all necessary information, whereas choreography-based approaches typically distribute monitoring across the subsystems. The main difference between approaches lies in the flow of information; whereas orchestrated approaches require trace information to flow to the (central) monitor, choreographed monitoring requires the verification effort to gather the information across locations. The choice of approach often depends on a number of factors, including the underlying system characteristics, as well as the properties under consideration.

### *2.1.1. Orchestration*

In an orchestrated approach, all monitoring is performed centrally, accessing the data and control information from different locations. This centralisation of monitoring facilitates the handling of dynamic contracts. The approach is depicted in Fig. 1 showing two sub-systems located at $l$ and $k$, each producing a local trace of events

[3]Although the term *contract* has been used in different ways in the literature, in our case we consider contracts to be specifications of the expected behaviour of the system, but which might be violated. This possibility motivates the need for the monitoring of such properties or contracts.

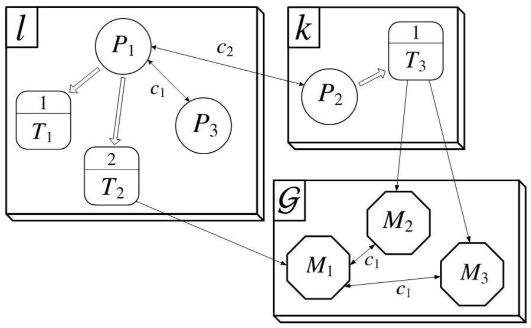

Figure 1: Orchestrated Monitoring

($T_1, T_2$ and $T_3$ respectively), subsequently analysed by monitors $M_1, M_2, M_3$ from remote location $\mathcal{G}$ (acting as a third sub-system). Shortcomings with this technique are immediately apparent; the approach is susceptible to data exposure when contacts concern private information, since local traces are transmitted across locations. Due to the volume of trace information which has to be transmitted remotely for monitoring, scaling up this scenario may also lead to a considerable increase in communication overhead across locations. Finally, the architecture poses a security risk by exposing the monitor as a central point of attack from which sensitive information can be tapped. Nevertheless, an orchestrated monitoring approach can be suitable when dealing with public information available on the communication medium. This approach is used in [1], where pre-determined web-service compositions expressed through pre-BPMN workflows are monitored in a statically orchestrated fashion. The use of a central monitor is facilitated in this case by placing the monitor at the coordinating BPMN engine, through which web-service interactions flow. The approach in [12] is similar, however also supporting orchestrated verification for dynamic properties, by runtime verifying contracts discovered *on-the-fly*.

### *2.1.2. Choreography*

In contrast with orchestrated monitoring, choreography-based approaches push verification locally, as shown in Fig. 2. This scenario depicts three sub-systems at $l, k$ and $h$ each generating local traces, with monitors $M_1$, $M_2$ placed at $l$, and $M_3$ placed at $k$. Monitors $M_2$ and $M_3$ eventually interact in order to synchronise the global monitoring effort. The appeal of localising the monitoring effort is the potential minimisation of data exposure and communication overhead. By verifying locally, we avoid having to transmit trace information to a remote monitor. Moreover, communication between localised monitors is typically less than that induced by the remote monitoring through a central monitor. Choreography is however more complex to instrument, as contracts need to be decomposed into coordinated local monitors. Furthermore, it is more intrusive, by burdening the monitored subsystems with additional local computation, and is thus applicable only when the subsystems allow local instrumentation of monitoring code. Statically choreographed monitors, *i.e.* localised monitors verifying a pre-determined set of properties, are also instrumented upfront, which may lead to

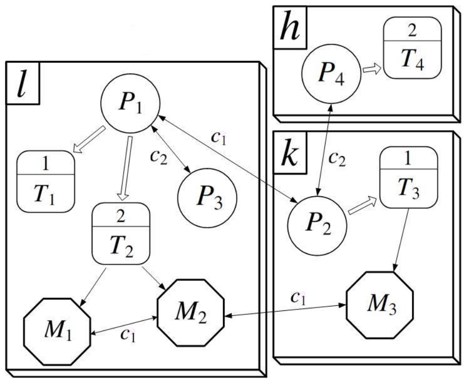

Figure 2: Choreographed Monitoring

redundant local instrumentation in the case of temporal dependencies in a contract; if monitoring at location $k$ is dependent on verification at location $l$, and the check at $l$ is never satisfied, upfront monitor instrumentation at $k$ is never needed. Extensive work has been done in static choreography-based monitoring [2, 3, 4, 5, 6], where communication overhead is mitigated by breaking up contracts into parts which can be monitored independently, synchronising between the monitors only when necessary.

### *2.1.3. Migrating Monitors*

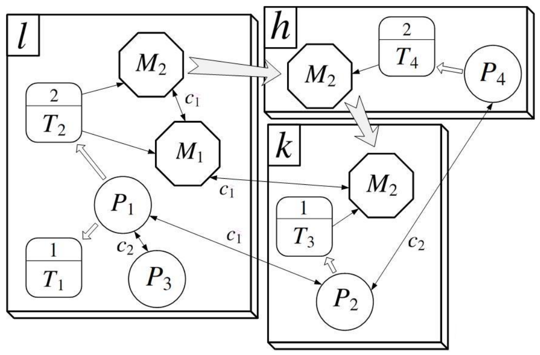

Figure 3: Choreographed Monitoring

An alternate approach to monitoring of distributed systems involves the use of *migrating monitors* [7]. In this approach, monitors reside where the immediate confidential traces occur and migrate to other subsystems, possibly discovered at runtime, when information from elsewhere is required *i.e.* on a *by-need* basis. This monitoring strategy lends itself directly to dynamic topologies and contracts learnt at runtime. Fig. 3 depicts monitor $M_2$, which starts at location $l$, and subsequently migrates to locations $k$ and $h$ during its verification effort. The sequential nature of migration is exploited in the process to extract a temporal order on events monitored across locations. The advantage with a migrating monitor approach is that dynamic contracts can be directly handled, whilst still avoiding orchestration, and thus minimising data exposure. It is for this reason that migrating monitors are considered a dynamic choreography-based strategy.

Nevertheless, the added expressivity and intrusiveness of migrating monitors requires a trust management infrastructure to ensure safe deployment of received monitors. Various solutions can be applied towards this end, from monitors signed by a trusted entity showing that they are the result of an approved contract negotiation process, to proof-carrying monitors which come with a proof guaranteeing what resources they access. Migrating monitors also burden locations with additional computation by running locally, and are intrusive by requiring local instrumentation of monitoring computation. Implementing migrating monitoring on an existing distributed system would require access to the individual systems, and the ability to instrument event detection and processing of contracts. The requirements of the instrumentation phase are not unlike those faced in choreographed monitoring. Although not all architectures may allow for migration of migrating processes, and even less so in a secure and safe manner, this challenge can be addressed by passing the monitors across location boundaries encoded as data objects, and which are interpreted at the location where they are to be evaluated. These issues will not be discussed further here, but are crucial for the practicality of migrating monitors.

#### *2.1.4. Comparing Monitoring Approaches for Distributed Systems*

Although these approaches have been individually studied in the literature, their formal comparison has not. There are a various issues relating to these different monitoring approaches that one would like to be able to formally resolve. For instance, how does one show that two monitors, possibly using different deployment strategies are equivalent, or that choreographed monitoring exposes less information on global channels than orchestrated monitoring? What is required is a common formal framework in which different approaches can be expressed, compared and contrasted.

## 3. A Distributed Monitoring Language

mDPi is an extension and adaptation of the distributed $\pi$-calculus [13], with a notion of (i) explicit locations to host processes; and (ii) monitors, a special form of processes which can (in a non-interfering manner) eavesdrop on the communication taking place on channels. In contrast to many forms of communication used in process calculi, the communication taking place between processes in mDPi leaves a residue which monitors may read at a later stage in a non-destructive manner.

### *3.1. The Syntax*

In mDPi, parallel processes, $P, Q, R \in \textsc{Proc}$, interact by communicating on channels, $c, d, e, b \in \textsc{Chans}$; they are distributed across a flat location structure where hosting locations, $l, k, m \in \textsc{Locs}$, *locally* administer event trace generation. Local trace generation yields totally ordered *local* traces but partially ordered *global* traces, reminiscent of tracing in distributed settings. Monitors, $M, N \in \textsc{Mon}$, then asynchronously analyse these partially ordered traces to determine whether properties are broken. Systems, $S, U, V \in \textsc{Sys}$, range over networks of located processes and monitors.

$$\begin{array}{rcl}
S, U, V & ::= & k[\![P]\!] \mid k[\![T]\!] \mid k\{\![M]\!\}^{(l,i)} \mid S \parallel U \mid \mathsf{new}\, c.S \\
P, Q, R & ::= & \mathsf{stop} \mid u!\overline{w}.P \mid u?\overline{x}.P \mid \mathsf{new}\, c.P \mid \mathsf{if}\ u = w\ \mathsf{then}\, P\ \mathsf{else}\, Q \mid P \| Q \mid *P \\
T & ::= & \mathbf{t}(c, \overline{w}, i) \\
M, N & ::= & \mathsf{stop} \mid u!w.M \mid u?x.M \mid \mathsf{new}\, c.M \mid \mathsf{if}\ u = w\ \mathsf{then}\ M\ \mathsf{else}\ N \mid M \| N \mid *M \\
 & \mid & \mathbf{q}(c, \overline{x}).M \mid \mathsf{sync}(u).M \mid \mathsf{go}\ u.M \mid \mathsf{ok} \mid \mathsf{fail}
\end{array}$$

The syntax, summarised above, assumes denumerable sets of indices $i, j, h \in \textsc{Idx}$ and variables $x, y, z \in \textsc{Vars}$ apart from channels and locations, where identifiers $u, w$ range over $\textsc{Idents} = \textsc{Chans} \cup \textsc{Locs} \cup \textsc{Idx} \cup \textsc{Vars}$. Lists of identifiers $w_1, \ldots, w_n$ are denoted as $\overline{w}$.

The main syntactic class is that of *Systems*, consisting of either located processes, $k[\![P]\!]$, located traces, $k[\![T]\!]$, or located monitors, $k\{\![M]\!\}^{(l,i)}$, that can be composed in parallel, $S \parallel U$, and are subject to scoping of channel names, $\mathsf{new}\, c.S$. Every located monitor carries a *monitoring context*, $(l, i)$ keeping track of the current location, $l$, and local position (index), $i$, of the trace being monitored.

### *3.1.1. Distributed Processes*

*Processes* comprise standard $\pi$-calculus constructs [13] such as output, $c!\overline{v}.P$ where value tuples $\overline{v}$ may include $\textsc{Chans} \cup \textsc{Locs} \cup \textsc{Idx}$, and input, $c?\overline{x}.P$, where variables $\overline{x}$ are bound in the continuation $P$. Processes include other constructs such as name-matching conditional, $\mathsf{if}\ u = w\ \mathsf{then}\, P\ \mathsf{else}\, Q$, replication, $*P$, parallel composition, $P \parallel Q$, and name restriction, $\mathsf{new}\, c.P$. We will sometimes elide $\mathsf{stop}$ and, for example, write $c!\overline{v}$ for $c!\overline{v}.\mathsf{stop}$ and $\mathsf{if}\ B\ \mathsf{then}\ P$ for $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{stop}$ respectively.

**Example 1.** *Consider the system of processes below whereby processes* (1) *and* (4) *are located at location l whereas processes* (2) *and* (3) *are located at location k.*

$$Sys \triangleq \overbrace{l[\![d?x.x!1]\!]}^{(1)} \parallel \overbrace{k[\![d!c.c!2]\!]}^{(2)} \parallel \overbrace{k[\![d!b.b?z.P]\!]}^{(3)} \parallel \overbrace{l[\![c?y.\mathit{if}\ y = 2\ \mathit{then}\ Q_1\ \mathit{else}\ Q_2]\!]}^{(4)}$$

*As in the piCalculus, channels can be communicated as values over other channels in mD*Pi *systems. Process* (1) *is waiting for input on channel d and the value inputted, x, is then used* as a channel *to output the value* 1 *on it.*

$$Sys_1 \triangleq \overbrace{l[\![c!1]\!]}^{(5)} \parallel \overbrace{k[\![c!2]\!]}^{(6)} \parallel \overbrace{k[\![d!b.b?z.P]\!]}^{(3)} \parallel \overbrace{l[\![c?y.\mathit{if}\ y = 2\ \mathit{then}\ Q_1\ \mathit{else}\ Q_2]\!]}^{(4)}$$

*Process* (1) *can receive this value from process* (2)*, in which case the input variable, x, will be instantiated to the channel name c and the entire system evolves to* $Sys_1$ *above. At this point the input on channel c in process* (4) *can non-deterministically react with either the output of process* (5)*, resulting in the system*

$$Sys_1' \triangleq l[\![\mathit{stop}]\!] \parallel k[\![c!2]\!] \parallel k[\![d!b.b?z.P]\!] \parallel l[\![\mathit{if}\ 1 = 2\ \mathit{then}\ Q_1\{^1/_y\}\ \mathit{else}\ Q_2\{^1/_y\}]\!]$$

*or else the output of process* (6)*, resulting in the system*

$$Sys_1'' \triangleq l[\![c!1]\!] \;\|\; k[\![\textsf{stop}]\!] \;\|\; k[\![d!b.b?z.P]\!] \;\|\; l[\![\textsf{if}\ 2{=}2\ \textsf{then}\ Q_1\{^2/_y\}\ \textsf{else}\ Q_2\{^2/_y\}]\!]$$

*Since the values communicated by these outputs differ, the input variable y at* (4) *may be instantiated at different values, i.e. either* 1 *or* 2*, which will in turn affect whether this process will branch to* $Q_1$ *or* $Q_2$*.*

*Alternatively, in the original system Sys, process* (1) *may receive the input on channel d from process* (3)*, in which case the input variable x will be instantiated to the channel b and Sys will evolve to the system* $Sys_2'$*, which clearly has a different behaviour from the former one.*

$$Sys_2 \triangleq \overbrace{l[\![b!1]\!]}^{(7)} \;\|\; \overbrace{k[\![d!c.c!2]\!]}^{(2)} \;\|\; \overbrace{k[\![b?z.P]\!]}^{(8)} \;\|\; \overbrace{l[\![c?y.\textsf{if}\ y{=}2\ \textsf{then}\ Q_1\ \textsf{else}\ Q_2]\!]}^{(4)}$$

*In fact, the new derivative of process* (1)*, i.e. process* (7)*, can not communicate with process* (4) *as in the case of* $Sys_1'$ *above, but may instead communicate with the derivative of process* (3)*, i.e., process* (8)*, yielding the system:*

$$Sys_2' \triangleq l[\![\textsf{stop}]\!] \;\|\; k[\![d!c.c!2]\!] \;\|\; k[\![P\{^1/_z\}]\!] \;\|\; l[\![c?y.\textsf{if}\ y{=}2\ \textsf{then}\ Q_1\ \textsf{else}\ Q_2]\!]$$

### *3.1.2. Distributed Runtime Monitoring*

The behaviour of such distributed systems is hard to analyse statically, due to the inherent non-deterministic nature of concurrent communication and the dynamic instantiation of channel names, as may be apparent from Example 1. In this work we propose how the behaviour of systems of located processes can be asynchronously monitored and verified at runtime; runtime analysis has the advantage of only analysing the current path of execution, thereby side-stepping a large number of problems associated with state-explosion of concurrent system analysis.

The calculus describes distributed, event-based, asynchronous monitoring. Monitoring is asynchronous because it happens in two phases, whereby the mechanism for tracing is detached from that for trace-querying. This two-step setup closely reflects the limits imposed by a distributed setting and lends itself better to the modelling of the various distributed monitoring mechanisms we want to capture. Monitoring is event-based because we focus on recording and analysing discrete events involving communication.

### *3.1.3. Distributed Traces*

Traces, made up of individual *trace records*, $\mathbf{t}(c, \overline{v}, i)$, record communication of values $\overline{v}$ on channel $c$ at timestamp $i$, and are meant to be ordered as a complete log recording past process computation *at a particular location*. For simplicity, traces in mDPI are limited to recording output events, but we conjecture that extensions to more expressive traces recoding other forms of actions such as inputs and name comparisons should be a straightforward task. Note that trace records are located, *e.g.*, $k[\![\mathbf{t}(c, \overline{v}, i)]\!]$ for some location $k$, and when they are composed in parallel, their syntactic ordering is not important, *e.g.*, writing $k[\![\mathbf{t}(c, \overline{v}, i)]\!] \;\|\; k[\![\mathbf{t}(d, \overline{w}, j)]\!]$ is the same as writing $k[\![\mathbf{t}(d, \overline{w}, j)]\!] \;\|\; k[\![\mathbf{t}(c, \overline{v}, i)]\!]$, because parallel composition is commutative. Rather, what is important is the relative ordering of trace records located at the same location, as dictated by their timestamp.

**Example 2.** *Consider the system of four processes, discussed in Example 1. If we consider the first sequence of computations possible,* i.e. *process* (1) *receiving the value c on channel d from process* (2)*, followed by a communication on channel c between processes* (1) *and* (4)*, then we obtain the trace*

$$Trc_1 \triangleq k[\![\mathbf{t}(d, c, i)]\!] \;\|\; l[\![\mathbf{t}(c, 1, j)]\!]$$

*whereas if we consider the communication between process* (1) *and process* (2) *on channel d, followed by the communication between processes* (2) *and* (4) *on channel c we obtain the trace*

$$Trc_2 \triangleq k[\![\mathbf{t}(d, c, i)]\!] \;\|\; k[\![\mathbf{t}(c, 2, i + 1)]\!]$$

*for some index values i and j.*

We note two important aspects of our traces from Example 2. First, motivated by implementation concerns, the recording of an output action as a trace entity *occurs locally*, *e.g.,* $\mathbf{t}$(d,c,i) is located at location $k$ in $\mathrm{Trc}_1$, the location of the process performing the output, even though the receiver resides at a different location $l$. Second, successive output actions at the *same* location are *ordered* by the assigned index *e.g.,* $\mathbf{t}$(d,c,i) and $\mathbf{t}$(c,2,i+1) at $k$ in $\mathrm{Trc}_2$. In contrast, temporally ordered actions that happen *at different locations*, *e.g.,* $\mathbf{t}$(d,c,i) and $\mathbf{t}$(c,1,j) in Trc, loose their ordering as they are assigned unrelated indexes by their respective locations $k$ and $l$, namely $i$ and $j$.

### *3.1.4. Distributed Monitors*

*Monitors* are autonomous computing entities similar in structure to processes, but with additional capabilities for checking and verifying distributed process computation through trace analysis. In a distributed setting, this amounts to a best-effort sound reconstruction of distributed computation from the partial information recorded in the local traces. In mDPi, monitors are allowed to perform this reconstruction from traces (such as those in Example 2) through a trace-querying construct, $\mathbf{q}(c,\tilde{x})$. M, and a trace-realignment construct, $\mathsf{sync}$(k). M; these two constructs embody our notion of asynchronous distributed monitoring.

Although sound, our asynchronous monitoring mechanism of reconstructing temporal order of events across locations is incomplete, and may miss out on detecting property violations when compared to more precise mechanisms such as Vector Clocks [14] and Lamport Timestamps [10]. These (more advanced) mechanisms however come at the price of increased construct complexity while still not guaranteeing completeness (in practice) due to the limits inherent to distributed computing [10]. Given that completeness is not a major concern for this study, we have opted for a construct that is clearly implementable rather going for more expressive, albeit more complex, constructs. In particular, the shortcomings of the constructs $\mathbf{q}(c,\tilde{x})$. M and $\mathsf{sync}$(k). M do not unfavourably affect any particular distributed monitoring strategy we express in our framework and therefore do not impact on the validity of the conclusions we reach in section 4.

The construct $\mathbf{q}(c,\overline{x}).M$ queries traces for the first record describing communication on channel $c$; the list of variables $\overline{x}$ are bound in the continuation $M$. The location, $l$, and index, $i$, of the trace where the record is to be searched for are obtained from the enclosing monitoring-context, $(l, i)$, in a monitor located at $k$, $k\{\![\mathbf{q}(c,\overline{x}).M]\!\}^{(l,i)}$. Traces can be analysed either locally, when $l = k$ or remotely, when $l \neq k$; this flexibility allows us to express both orchestrated monitoring strategies, which require remote monitoring, as well as choreographed strategies, which favour local monitoring. In order to permit modular instrumentation of independent properties, mDPi allows *multiple* monitors to analyse *concurrently* the same trace. Trace records are thus *not consumed* when queried (unlike output messages); instead, every monitor keeps its own position in the trace through the monitoring context.

Monitors can reconstruct a temporal ordering of events across remote traces (which are temporally unrelated) using the realignment construct, $\mathsf{sync}(l).M$. This construct resets the monitoring context, $(m, h)$, of a monitor $k\{\![\mathsf{sync}(l).M]\!\}^{(m,h)}$ to $(l, i)$, where $i$ is the index to be assigned to the *next* generated trace record at location $l$; this allows the monitor to start monitoring for records at $l$ from the present state of the computation onwards. As in the case of querying, monitor re-alignment can be performed both locally, when $l = k$, as well as remotely, when $l \neq k$, which facilitates the encoding of various distributed monitoring strategies.

Our framework permits the monitor allocation across locations to change over the course of computation. In fact, as opposed to processes, monitors can also migrate from their existing location to another location $l$ using the construct $\mathsf{go}\ l.M$. This, in turn, allows us to express a wider variety of monitoring strategies such as the migrating monitors strategy, discussed earlier in Section 2.1.3. The other remaining constructs used exclusively by monitors are $\mathsf{ok}$ and $\mathsf{fail}$, which allow monitors to report success or failure respectively.

**Example 3.** *Consider the property that prohibits outputting an integer that is less than* 2 *on some channel x at location l after this channel x had been earlier outputted as a value on channel d at location k. Monitoring whether the system Sys from Example 1 violates this property can be carried out in a variety of ways:*

$$M^{orch} \triangleq m\{\![\mathbf{q}(d,x).\mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i)}$$

$$M^{chor} \triangleq \mathsf{new}\, b.\Big(k\{\![\mathbf{q}(d,x).\ b!x]\!\}^{(k,i)} \ \parallel\ l\{\![b?x.\ \mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,h)}\Big)$$

$$M^{mig} \triangleq k\{\![\mathbf{q}(d,x).\mathsf{go}\ l.\mathsf{sync}(l).\mathbf{q}(x,y).\mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i)}$$

*$M^{orch}$ describes an* orchestrated approach*, analysing traces* remotely *from a central (main) location m. $M^{chor}$ is a* choreographed monitor*, split into two sub-monitors, each analysing traces* locally *and communicating between them (on the scoped channel b) when necessary. Finally, $M^{mig}$ is a migrating monitor, which starts at location k, locally eavesdropping on channel d, and then migrates to location l once an output on d is recorded (and the channel communicated on d is known). Note that each monitor needs to* dynamically *obtain the channel to query next on the trace at location l from the first query performed on the trace at k, which means that the monitors need to reconstruct this temporal ordering across located traces. When executed over the trace $Trc_1$ all three strategies should be able to raise the violation; they may nevertheless miss to*

*raise this violation because local traces do not provide a total ordering of events — we discuss this point at length in section 3.3. However, when monitoring the trace $\mathit{Trc}_2$ the monitors should never flag a violation.*

There are various reasons why one monitoring strategy may be preferred over the other. $M^{\mathrm{orch}}$ is perhaps the easiest to construct and instrument; it is also the least intrusive as no monitors are instrumented at the location where the the monitored system resides, *i.e.*, the monitor resides at $m$ whereas the monitored system is distributed across $l$ and $k$. On the other hand, $M^{\mathrm{chor}}$ is the option that generates the least amount of network traffic by performing all its trace queries locally at $l$ and $k$. Finally, $M^{\mathrm{mig}}$, is able to perform all its monitoring locally but is less intrusive than $M^{\mathrm{chor}}$, since it only migrates to $l$ if a communication on $d$ at $k$ is observed.

A proper analysis of these different strategies requires us to formalise the behaviour of these monitors first; this is the theme of the following section.

## *3.2. The Semantics of mD*Pi

The semantics of mDPi, is given in terms of a Labelled Transition System (LTS), defined over the following action labels:

$$
\begin{array}{llll}
\mu \in \textsc{pAct} \triangleq & \mathbf{inP}(c,\overline{v}) & \mid\ (\overline{b})\mathbf{outP}(c,\overline{v}) & \text{(process input and output)} \\
\quad\mid & \mathbf{inM}(c,\overline{v}) & \mid\ (\overline{b})\mathbf{outM}(c,\overline{v}) & \text{(monitor input and output)} \\
\quad\mid & \mathbf{inT}(c,\overline{v},l,i) & \mid\ (\overline{b})\mathbf{outT}(c,\overline{v},l,i) & \text{(trace query and availability)} \\
\quad\mid & \tau & \mid\ \mathbf{tick} & \text{(internal and clock actions)}
\end{array}
$$

The different labels for both inputs and outputs allow us to discern whether the action was performed by a process, monitor or trace. The first four labels denote the capability of a process (respectively monitor) to input/output value tuples $\overline{v}$ on channel $c$. As is standard for channel-passing calculi, the list of channel names $(\overline{b})$ in the respective output labels denotes bound names that have been *scope extruded* as a result of the output. The label $(\overline{b})\mathbf{outT}(c,\overline{v},l,i)$ does *not* describe trace record consumption but instead denotes the existence of an $i^{th}$ record in the trace at location $l$ recording a process' output of values $\overline{v}$ on channel $c$ (amongst which channel values $(\overline{b})$ are scoped). Dually, the label $\mathbf{inT}(c,\overline{v},l,i)$ describes a monitor's capability to *analyse* the $i^{th}$ trace record at location $l$ recording a process' output of values $\overline{v}$ on $c$; note that the location of the monitor is not observable. All three output actions $(\overline{b})\mathbf{outP}(c,\overline{v})$, $(\overline{b})\mathbf{outM}(c,\overline{v})$ and $(\overline{b})\mathbf{outT}(c,\overline{v},l,i)$ observe standard conditions such as, $\overline{b} \subseteq \overline{v}$, *i.e.*, scope extruded channels are indeed outputted values, and $c \notin \overline{b}$, *i.e.*, the channel on which the communication occurs is known to the receiver, and hence, not scoped. Finally, label $\tau$ denotes the (standard) silent unobservable action, resulting from some internal computation or interaction whereas **tick** denotes a local clock increment.

As usual, the respective input and output actions are allowed to synchronise with one another in our LTS. To specify this, we describe when an output action is the co-action of an input action, *i.e.* when the labels match *wrt.* their parameters[4]:

$$\overline{\mathbf{inP}(c,\overline{v})} = \mathbf{outP}(c,\overline{v}) \qquad \overline{\mathbf{inM}(c,\overline{v})} = \mathbf{outM}(c,\overline{v}) \qquad \overline{\mathbf{inT}(c,\overline{v},l,i)} = \mathbf{outT}(c,\overline{v},l,i)$$

### *3.2.1. Structural equivalence*

As in [15, 13], we find it convenient to abstract over semantically equivalent systems with different syntactic representations. For instance, in our calculus parallel composition is commutative and, as a result, we would expect the systems $S_1 \parallel S_2$ and $S_2 \parallel S_1$ to denote the same system. We characterise this through the structural equivalence relation, denoted by the symbol $\equiv$, and defined as the least relation satisfying the rules in Figure 4; we use $fn(S)$ to denote the *free names* of $S$ *i.e.* channel names that are not bound by input prefixes and scoping.

$$\textsc{sCom}\ \frac{}{S \parallel U \ \equiv\ U \parallel S} \qquad \textsc{sAss}\ \frac{}{(S \parallel U) \parallel V \ \equiv\ S \parallel (U \parallel V)}$$

$$\textsc{sId1}\ \frac{}{S \parallel k[\![\mathsf{stop}]\!] \ \equiv\ S} \qquad \textsc{sId2}\ \frac{}{S \parallel k\{\!\{\mathsf{stop}\}\!\}^{(l,i)} \ \equiv\ S} \qquad \textsc{sTrc}\ \frac{}{k[\![T]\!] \parallel k[\![T]\!] \ \equiv\ k[\![T]\!]}$$

$$\textsc{sExt}\ \frac{}{S \parallel \mathsf{new}\, c.U \ \equiv\ \mathsf{new}\, c.(S \parallel U)}\ c \notin fn(S) \qquad \textsc{sScp1}\ \frac{}{\mathsf{new}\, c.(k[\![\mathsf{stop}]\!]) \ \equiv\ k[\![\mathsf{stop}]\!]}$$

$$\textsc{sFlp}\ \frac{}{\mathsf{new}\, c.\mathsf{new}\, d.U \ \equiv\ \mathsf{new}\, d.\mathsf{new}\, c.U} \qquad \textsc{sScp2}\ \frac{}{\mathsf{new}\, c.(k\{\!\{\mathsf{stop}\}\!\}^{(l,i)}) \ \equiv\ k\{\!\{\mathsf{stop}\}\!\}^{(l,i)}}$$

$$\textsc{sCtx1}\ \frac{S \ \equiv\ U}{S \parallel V \ \equiv\ U \parallel V} \qquad \textsc{sCtx2}\ \frac{S \ \equiv\ U}{V \parallel S \ \equiv\ V \parallel U} \qquad \textsc{sCtx3}\ \frac{S \ \equiv\ U}{\mathsf{new}\, c.S \ \equiv\ \mathsf{new}\, c.U}$$

Figure 4: mDPi Structural Equivalence Rules

Most of the rules in Figure 4 are standard, such as the commutativity and associativity rules for parallel composition, sCom and sAss. A slightly non-standard aspect is that systems form a commutative monoid *wrt.* parallel composition and two forms of 'identity' systems, namely inert located processes, $k[\![\mathsf{stop}]\!]$, or inert (located) monitors, $k\{\!\{\mathsf{stop}\}\!\}^{(l,i)}$, as described by sId1 and sId2. The channel scoping rules for extrusion, sExt, and swapping, sSwp, are also standard, whereas unused scoped channels can be discarded through either inert located processes or monitors, sScp1 and sScp2. The only rule worth highlighting because it is peculiar to our calculus is sTrc, which allows identical trace records to be consolidated or, dually, replicated. This rule is introduced for technical reasons that will be discussed in Sections 3.2.2 and 3.2.3, and then Section 3.4. Finally, the inductive rules sCtx1, sCtx2 and sCtx3 make structural equivalence a congruence *wrt.* parallel compositions and scoping.

[4]Note that the co-action relation does not include output actions with bound names.

*3.2.2. Configurations*

Our LTS is defined over systems subject to a *local logical clock* at each location. These clocks permit (i) the generation of locally-ordered trace-records, and (ii) the re-alignment of monitors to the current record timestamp in the local trace. Local clocks are modelled as monotonically increasing counters and expressed as a partial function $\delta \in \textsc{Clocks} :: \textsc{Locs} \rightharpoonup \mathbb{N}$, where $\delta(l)$ denotes the next timestamp to be assigned for a trace entity generated at location $l$. We define a clock increment at a particular location, say $k$, using standard function overriding,

$$inc(\delta, k) = \delta[k \mapsto (\delta(k) + 1)].$$

*Configurations*, denoted as $C, D \in \textsc{Conf} :: \textsc{Clocks} \times \textsc{Sys}$, are systems that are subject to a set of localised counters, *i.e.* $\langle \delta, S \rangle$ pairs where $locs(S) \subseteq dom(\delta)$; when such a condition holds, we denote the pair as $\delta \rhd S$. We limit ourselves to well-formed configurations whereby trace records form a local linear order at each location. For this to happen, Definition 1 requires that (*1*) trace records are locally timestamped at an index that is strictly less than the local counter that will be assigned to the next trace record generated at that location and (*2*) trace records referring to the same local timestamp $(k, i)$ must agree on the recorded data.[5]

**Definition 1** (Well-formed Configurations). *A configuration $\delta \rhd S$ is well-formed iff it satisfies the following conditions:*

1. $S \equiv \mathsf{new}\,\overline{c}.(U \parallel k[\![\mathbf{t}(c, \overline{w}, i)]\!])$ *implies* $\delta(k) > i$;
2. $S \equiv \mathsf{new}\,\overline{b}.(U \parallel k[\![\mathbf{t}(c, \overline{w}, i)]\!]) \equiv \mathsf{new}\,\overline{e}.(V \parallel k[\![\mathbf{t}(d, \overline{v}, i)]\!])$ *implies* $c = d$ *and* $\overline{v} = \overline{w}$.

We note that Definition 1 does not require that, in a well-formed configuration $\delta \rhd S$, all trace records for indexed less than $\delta(k)$, for arbitrary $k \in dom(\delta)$, are present in $S$. We also note that Definition 1 does not enforce the existence of a single trace record to be present for a particular timestamp either, but rather it permits replicated trace records, as long as they record the same information. In fact, in what follows, trace records are perhaps best envisaged as partial information relating to past performance that must remain invariant across a configuration.

The reason for these relaxations is that we want our semantics to analyse configurations in an *open world* setting, whereby results determined over local systems (such as equivalences) are preserved in larger system contexts. Dually, these relaxations also allow for *compositional analysis* of configurations, whereby a configuration of the form $\delta \rhd (S_1 \parallel S_2)$ can be analysed from its sub-configurations $\delta \rhd S_1$ and $\delta \rhd S_2$ (see Theorem 1 in Section 3.4). In the meantime, however, note that missing or replicated trace records do not affect the local linear recording of past outputs at a particular location; moreover, replicated trace records can always be consolidated when considering systems up to structural equivalence, using the rule sTrc from Figure 4.

[5]The initial value of every local clock is not important, as long as it is larger than any trace record at that location — its role is merely to order future trace records generated at that location.

### *3.2.3. Transition rules*

Our LTS is defined as the least ternary relation over configurations, $\longrightarrow :: \textsc{Conf} \times \textsc{pAct} \times \textsc{Conf}$, satisfying the rules in Figure 5, Figure 6 and Figure 7; we elide some obvious process rules that can be inferred from the corresponding monitor rules (see Figure 7). Transitions are denoted by the following notation:

$$\delta \rhd S \xrightarrow{\mu} \delta' \rhd S' \tag{1}$$

in lieu of $\langle(\delta \rhd S), \mu, (\delta' \rhd S')\rangle \in \longrightarrow$ and describe the computation step from the configuration $\delta \rhd S$ to the configuration $\delta' \rhd S'$ as a result of some interaction with the environment (some system context), characterised by the action $\mu$. We note that whenever we want to restrict our analysis to a 'closed world' setting, disallowing the possibility of interacting with the environment, we simply restrict configuration transition labels to just the silent action, $\tau$.

$$\textsc{OutP}\ \frac{}{\delta \rhd k[\![c!\bar{v}.P]\!] \xrightarrow{\mathbf{outP}(c,\bar{v})} \mathsf{inc}(\delta, k) \rhd k[\![P]\!] \parallel k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]}$$

$$\textsc{OutM}\ \frac{}{\delta \rhd k\{\![c!v.M]\!\}^{(l,i)} \xrightarrow{\mathbf{outM}(c,\bar{v})} \delta \rhd k\{\![M]\!\}^{(l,i)}}$$

$$\textsc{OutT}\ \frac{}{\delta \rhd k[\![\mathbf{t}(c, \bar{v}, i)]\!] \xrightarrow{\mathbf{outT}(c,\bar{v},k,i)} \delta \rhd k[\![\mathbf{t}(c, \bar{v}, i)]\!]}$$

$$\textsc{InP}\ \frac{}{\delta \rhd l[\![c?x.P]\!] \xrightarrow{\mathbf{inP}(c,\bar{v})} \delta \rhd l[\![P\{^{\bar{v}}/_{x}\}]\!]}$$

$$\textsc{InM}\ \frac{}{\delta \rhd l\{\![c?x.M]\!\}^{(k,i)} \xrightarrow{\mathbf{inM}(c,\bar{v})} \delta \rhd l\{\![M\{^{\bar{v}}/_{x}\}]\!\}^{(k,i)}}$$

$$\textsc{InT}\ \frac{}{\delta \rhd l\{\![\mathbf{q}(c, \bar{x}).M]\!\}^{(k,i)} \xrightarrow{\mathbf{inT}(c,\bar{v},k,i)} \delta \rhd l\{\![M\{^{\bar{v}}/_{\bar{x}}\}]\!\}^{(k,i+1)} \parallel k[\![\mathbf{t}(c, \bar{v}, i)]\!]}\ \delta(k) > i$$

$$\textsc{Skp}\ \frac{}{\delta \rhd l\{\![\mathbf{q}(d, \bar{x}).M]\!\}^{(k,i)} \xrightarrow{\mathbf{inT}(c,\bar{v},k,i)} \delta \rhd l\{\![\mathbf{q}(d, \bar{x}).M]\!\}^{(k,i+1)} \parallel k[\![\mathbf{t}(c, \bar{v}, i)]\!]}\ [c \neq d, \delta(k) > i]$$

Figure 5: mDPi LTS external action rules

Figure 5 describes three output rules. The process output rule, OutP, is central to our monitoring semantics. It differs from standard output rules in two respects; first, it generates a residue trace record at the current location, $k$, after the output occurs, $k[\![\mathbf{t}(c, \bar{v}, \delta(k))]\!]$, recording the channel name, $c$, and the values communicated on it, $\bar{v}$, timestamped by $\delta(k)$; second, it increments the clock at $k$ once the trace record is generated, which is necessary so as to generate a total order on trace records at $k$. Monitor output, OutM, is similar albeit simpler since neither is a trace record generated, nor is the local counter updated. Rule OutT models a trace record's capability to expose information relating to a process output at location $k$, outputting tuple $\overline{v}$ on channel $c$ at timestamp $i$. As opposed to process and monitor outputs, the trace record is not consumed by the action (thereby acting as a broadcast), and its persistence allows for multiple monitors to query it.

Figure 5 also describes three (main) input rules, one for each input label. The rule for process input, InP, is standard: a process that resides at location $l$, can input tuple $\overline{v}$ over channel $c$, substituting $v_i \in \overline{d}$ for $x_i \in \overline{x}$ in the continuation $P$. Note that the location of the process plays no role in this action since process communication is allowed to happen *across locations*. The rule for monitor output, InM, is similar; again, the location, $l$, and the monitoring-context, $(k, i)$, do not affect the monitor input.

The monitoring-context however plays a central role in a (successful) trace-query action, InT. Here, the source location of the trace record, $k$, and time stamp, $i$, of the action label, $\mathbf{inT}(c, \overline{v}, k, i)$, must match those of current monitoring-context $(k, i)$. Since the transition describes the fact that a trace record has been matched by the monitor query, the timestamp index of the monitoring-context is incremented, $(k, i+1)$, in order to progress to the next record in the local trace ordering. The trace-query action can occur only if the corresponding trace record at $(k, i)$ has already been generated: this can be determined from the rule side-condition checking the clock counter at $k$, *i.e.*, $\delta(k) > i$; if the side-condition is not satisfied then the query is blocked until the local clock is incremented accordingly (see rule OutP). Rule InT leaves a residual trace record, $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$, in the resulting configuration; this should not be confused with the trace record generation of OutP, but rather, it should be understood as the generation of an *invariant* ensuring that all future queries of the local trace at $(k, i)$ consistently react with a trace record describing the output of values $\overline{v}$ on channel $c$. Although this may appear somewhat intricate, this mechanism works in tandem with the well-formed conditions of Definition 1 and turns out to be essential in a setting where we want to perform compositional analysis — see example 8 in section 3.3. In a setting where only part of the system is being analysed, the actual trace record may not be part of the sub-system; the mechanism allows us to keep track of the trace record data queried in the current sub-system, and then checked at the composition phase to be *consistent* with the trace records generated (through rule OutP) in the other sub-systems. We recall that when trace records correspond, they can be consolidated using the structural rule sTrc from Figure 4, which allows us to reconstruct the intuition outlined earlier in Section 3.1.3 and Section 3.1.4 that there exists at most one trace record for each located index $(k, i)$ — see example 5.

The final transition rule in Figure 5 is Skp, describing an unsuccessful trace-query action. Here the monitor attempts to query the trace record at $(k, i)$ for outputs on channel $d$, and since the trace record queried happens to describe an output on some other channel, $c$, the monitor query skips this record and proceeds up the chain to the next index, $(k, i + 1)$.[6] The resulting configuration of this transition again generates a

[6]Interestingly, the transition action is the same for that of a successful action, namely $\mathbf{inT}(c, \overline{v}, k, i)$, intuitively because it still denotes the interaction with a trace record $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$, even though the query was

residual trace record describing a system invariant, as in the case of the previous rule $\textsc{in}$T; the motivation for this is the same as for $\textsc{in}$T, namely compositional analysis — see example 5 and example 8 from section 3.3.

$$\textsc{Opn}\ \dfrac{\delta \triangleright S \xrightarrow{(\tilde{b})\alpha} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\, b.S \xrightarrow{(b,\tilde{b})\alpha} \delta' \triangleright S'}\ [b \in (fn(\alpha) \setminus obj(\alpha)), b \notin \tilde{b}]$$

$$\textsc{Scp}\ \dfrac{\delta \triangleright S \xrightarrow{\mu} \delta' \triangleright S'}{\delta \triangleright \mathsf{new}\, b.S \xrightarrow{\mu} \delta' \triangleright \mathsf{new}\, b.S'}\ [b \notin fn(\mu)]$$

$$\textsc{Par}\ \dfrac{\delta \triangleright S \xrightarrow{\mu} \delta' \triangleright S'}{\delta \triangleright S \parallel U \xrightarrow{\mu} \delta' \triangleright S' \parallel U}\ [bn(\mu) \cap fn(U) = \emptyset, \mu \neq \textsc{in}\mathbf{T}(c, \tilde{v}, k, i)]$$

$$\textsc{ParT}\ \dfrac{\delta \triangleright S \xrightarrow{\mathbf{inT}(c,\tilde{v},k,i)} \delta' \triangleright S'}{\delta \triangleright S \parallel U \xrightarrow{\mathbf{inT}(c,\tilde{v},k,i)} \delta' \triangleright S' \parallel U}\ [\nexists U'.\, U \xrightarrow{(\tilde{b})\textsc{out}\mathbf{T}(d,\tilde{w},k,i)} U']$$

$$\textsc{Com}\ \dfrac{\delta \triangleright S \xrightarrow{(\tilde{b})\overline{\mu}} \delta' \triangleright S' \qquad \delta \triangleright U \xrightarrow{\mu} \delta \triangleright U'}{\delta \triangleright S \parallel U \xrightarrow{\tau} \delta' \triangleright \mathsf{new}\, \tilde{b}.(S' \parallel U')}\ [\tilde{b} \cap fn(U) = \emptyset]$$

$$\textsc{Str}\ \dfrac{S \equiv S' \qquad \delta \triangleright S' \xrightarrow{\mu} \delta' \triangleright U' \qquad U' \equiv U}{\delta \triangleright S \xrightarrow{\mu} \delta' \triangleright U} \qquad \textsc{Cntr}\ \dfrac{}{\delta \triangleright S \xrightarrow{\mathbf{tick}} inc(\delta, k) \triangleright S}$$

Figure 6: mDPi LTS Contextual Rules

The first rules in Figure 6 are *contextual* rules, relating to the system contexts of scoping and parallel composition. A subtle but important aspect of our calculus, distinct from related calculi such as [13], is that scope extrusion of channel names may occur both *directly*, through process or monitor communication, but also *indirectly* through trace querying. These three cases of scope extrusion are handled by the rule Opn which uses the action variable $\alpha$ to range over process, monitor and trace output actions carrying no bound names:

$$\alpha \in \textsc{outAct} \triangleq \mathbf{outP}(c, \tilde{v}) \ \mid\ \mathbf{outM}(c, \tilde{v}) \ \mid\ \mathbf{outT}(c, \tilde{v}, l, i)$$

The rule uses two functions in its side condition: $fn(\alpha)$ returns the free names in an

unsuccessful.

action as expected whereas

$$obj(\mathbf{outP}(c,\overline{v})) = obj(\mathbf{outM}(c,\overline{v})) = obj(\mathbf{outT}(c,\overline{v},l,i)) = c.$$

Actions for scoped system that do not involve any extrusion of channel names are handled by the rule Scp.

Actions are preserved by systems in parallel, as stated in rule Par. The side-condition, $bn(\mu) \cap fn(U) = \emptyset$, ensures that scope extruded channel names are fresh[7]; since we assume terms up to $\alpha$-equivalence of bound channel names, these names can be assumed to be distinct from any free variables in the surrounding systems. The only exception to this is the trace querying action, $\mathbf{inT}(c,\overline{v},k,i)$, which requires a further condition stating that for the action to be preserved, systems composed in parallel with it, $U$, must not be able to broadcast the existence of a trace record at $(k,i)$. Otherwise, when $U$ contains the trace record queried for at $(k,i)$, the system $S$ is forced to synchronise with this record in system $U$ through the rule Com.

All three forms of communications — process, monitor and trace — are handled uniformly by the communication rule Com. Communication occurs when configurations $\delta \triangleright S$, $\delta \triangleright U$ are capable of performing action $\mu$ and co-action $\overline{\mu}$ respectively, yielding a silent action $\tau$ as a result. Note that, by our definition of co-action, $\overline{\mu}$ must be a plain (not containing any bound names) output action, *i.e.*, either $\mathbf{outP}(c,\overline{v})$ (process output), $\mathbf{outM}(c,\overline{v})$ (monitor output) or $\mathbf{outT}(c,\overline{v},l,i)$ (trace broadcast), and may therefore involve the scope extrusion of certain channel names, $\overline{b}$, local to system $S$ emitting the output. The side-condition, $\overline{b} \cap fn(U) = \emptyset$, ensures that the scope extruded names are distinct from any free variables in the receiving system and, as a result of the synchronisation, the scope of these names is extended to the residual system of $U$ after communication, *i.e.*, $U'$ in $\mathsf{new}\,\overline{b}.(S' \parallel U')$. It is worth noting that action $\mu$ must be an input action, and from the input rules that can generate this, *i.e.*, InP, InM, InT and Skp, we can conclude that this action does not affect the system clocks $\delta$ in the residual of the respective premise, $\delta \triangleright U'$. On the other hand, when $\overline{\mu}$ is a process output, $\mathbf{outP}(c,\overline{v})$, the local clock of the location where the action occurs is incremented in the residual system of the respective premise, $\delta' \triangleright S'$ (see OutP); this is then is reflected in the resulting configuration for rule Com, $\delta' \triangleright \mathsf{new}\,\overline{b}.(S' \parallel U')$.

Rule Str in Figure 6 lifts transition to structurally equivalent systems. In particular, this is important for our LTS in order to express symmetric rules for the parallel rules Par, ParT and Com through structural rules such as sCom, sAss and sExt, and to discard remnant inert code and unused local channels through rules such as sId1 and sScp1. The last rule in Figure 6 is Cntr and models process communication by some system context in an open world setting, that would increase the timestamp counter at the location where the output occurred.

**Example 4.** *Recall the system below from Example 1.*

$$Sys \triangleq l[\![d?x.x!1]\!] \parallel k[\![d!c.c!2]\!] \parallel k[\![d!b.b?x.P]\!] \parallel l[\![c?y.\mathsf{if}\ y{=}2\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2]\!]$$

[7]Analogous to $fn(S)$, we write $bn(S)$ to denote the *bound names* of $S$.

*When subject to the set of local clocks $\{l \mapsto j, k \mapsto i\}$, the semantics defined by the rules in Figure 5 and Figure 6 allows us to express following behaviour:*

$$\begin{aligned}\{l \mapsto j, k \mapsto i\} \triangleright Sys &\xrightarrow{\tau} \{l \mapsto j, k \mapsto i{+}1\} \triangleright Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!]\\ &\xrightarrow{\tau} \{l \mapsto j{+}1, k \mapsto i{+}1\} \triangleright Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!]\end{aligned}$$

*The first $\tau$-transition describes the process communication of the value c on channel d, discussed in Example 1, where $Sys_1$ was defined as*

$$l[\![c!1]\!] \parallel k[\![c!2]\!] \parallel k[\![d!b.b?z.P]\!] \parallel l[\![c?y.\textsf{if}\ y{=}2\ \textsf{then}\ Q_1\ \textsf{else}\ Q_2]\!]$$

*This transition can be derived using the rules* Com, InP *and* OutP *(together with the parallel composition rule* Par *and structural manipulation of terms using* Str*); see derivation below where $\delta = \{l \mapsto j, k \mapsto i\}$. In particular, rule* OutP *generates the residual trace record $k[\![\mathbf{t}(d,c,i)]\!]$ at location k, as discussed earlier in Example 2, while incrementing the local clock at this location, $k \mapsto i{+}1$.*

$$\dfrac{\dfrac{\dfrac{}{\delta \triangleright k[\![d!c.c!2]\!] \xrightarrow{\textbf{outP}(d,c)} inc(\delta,k) \triangleright k[\![c!2]\!] \parallel k[\![\mathbf{t}(d,c,i)]\!]}\ \textsc{OutP} \quad \vdots \quad \dfrac{}{\delta \triangleright l[\![d?x.x!1]\!] \xrightarrow{\textbf{inP}(d,c)} \delta \triangleright l[\![c!1]\!]}\ \textsc{InP}}{\delta \triangleright k[\![d!c.c!2]\!] \parallel l[\![d?x.x!1]\!] \xrightarrow{\tau} inc(\delta,k) \triangleright k[\![c!2]\!] \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![c!1]\!]}\ \textsc{Com}}{\delta \triangleright Sys \xrightarrow{\tau} inc(\delta,k) \triangleright Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!]}\ \textsc{Par, Str}$$

*The second $\tau$-transition above describes the communication of value* 1 *on channel c resulting in the system $Sys_1'$, defined in Example 1 as*

$$l[\![\textsf{stop}]\!] \parallel k[\![c!2]\!] \parallel k[\![d!b.b?z.P]\!] \parallel l[\![\textsf{if}\ 1{=}2\ \textsf{then}\ Q_1\{^1\!/\!_y\}\ \textsf{else}\ Q_2\{^1\!/\!_y\}]\!].$$

*This transition can also be derived using a combination of same rules just discussed, this time incrementing the local clock at location l. It yields the global trace $k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!]$, i.e., $Trc_1$ from Example 2.*

$$\begin{aligned}\{l \mapsto j, k \mapsto i\} \triangleright Sys &\xrightarrow{\tau} \{l \mapsto j, k \mapsto i{+}1\} \triangleright Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!]\\ &\xrightarrow{\tau} \{l \mapsto j, k \mapsto i{+}2\} \triangleright Sys_1'' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel k[\![\mathbf{t}(c,2,i{+}1)]\!]\end{aligned}$$

*We can also derive a second transition sequence shown above where $Sys_1''$ was defined in Example 1 as the system*

$$l[\![c!1]\!] \parallel k[\![\textsf{stop}]\!] \parallel k[\![d!b.b?z.P]\!] \parallel l[\![\textsf{if}\ 2{=}2\ \textsf{then}\ Q_1\{^2\!/\!_y\}\ \textsf{else}\ Q_2\{^2\!/\!_y\}]\!],$$

*and the residual trace generated, $k[\![\mathbf{t}(d,c,i)]\!] \parallel k[\![\mathbf{t}(c,2,i{+}1)]\!]$, is the one discussed in Example 2, i.e., $Trc_2$. Note that since this sequence of transitions describes two communications where both outputs emanate from the same location, k, the trace records generated are able to describe the relative order of the communication, i.e. from their timestamps, $k[\![\mathbf{t}(d,c,i)]\!]$ precedes $k[\![\mathbf{t}(c,2,i{+}1)]\!]$.*

$$\textsc{SpltM}\ \frac{}{\delta \triangleright k\{\![M \parallel N]\!\}^{(l,i)} \xrightarrow{\tau} \delta \triangleright k\{\![M]\!\}^{(l,i)} \parallel k\{\![N]\!\}^{(l,i)}}$$

$$\textsc{NewM}\ \frac{}{\delta \triangleright k\{\![\mathsf{new}\, c.M]\!\}^{(l,i)} \xrightarrow{\tau} \delta \triangleright \mathsf{new}\, c.(k\{\![M]\!\}^{(l,i)})}$$

$$\textsc{EqM}\ \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^{(l,i)} \xrightarrow{\tau} \delta \triangleright k\{\![M]\!\}^{(l,i)}}\ [u = v]$$

$$\textsc{NeqM}\ \frac{}{\delta \triangleright k\{\![\mathsf{if}\ u\!=\!v\ \mathsf{then}\ M\ \mathsf{else}\ N]\!\}^{(l,i)} \xrightarrow{\tau} \delta \triangleright k\{\![N]\!\}^{(l,i)}}\ [u \neq v]$$

$$\textsc{RecM}\ \frac{\delta \triangleright k\{\![M]\!\}^{(l,i)} \xrightarrow{\mu} \delta \triangleright k'\{\![M']\!\}^{(m,j)}}{\delta \triangleright k\{\![*M]\!\}^{(l,i)} \xrightarrow{\mu} \delta \triangleright k'\{\![M']\!\}^{(m,j)} \parallel k\{\![*M]\!\}^{(l,i)}}$$

$$\textsc{Sync}\ \frac{}{\delta \triangleright k\{\![\mathsf{sync}(l).M]\!\}^{(h,i)} \xrightarrow{\tau} \delta \triangleright k\{\![M]\!\}^{(l,\delta(l))}}$$

$$\textsc{Go}\ \frac{}{\delta \triangleright k\{\![\mathsf{go}\ l.M]\!\}^{(h,i)} \xrightarrow{\tau} \delta \triangleright l\{\![M]\!\}^{(h,i)}}$$

Figure 7: mDPi monitor LTS rules

The rules in Figure 7 describe the remaining monitor transitions; corresponding process transitions such as process splitting and process branching follow the same structure as the monitor rule counterpart, but are simpler as they do not have to cater for the monitoring context — they are also very similar to those found in [13] and are elided here. Most of the rules are self explanatory. The only rules worth noting are the silent action described by rule Sync, which allows monitors to realign with a trace at a particular location to start monitoring for future trace records there, and rule Go, which describes monitor migration as found in calculi such as [13]; presently, in order to keep our framework as simple as possible, only monitors are allowed to migrate.[8]

**Example 5.** *Recall the orchestrated monitor $M^{orch}$ from Example 3,*

$$m\{\![\boldsymbol{q}(d,x).\mathsf{sync}(l).\ \boldsymbol{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k,i)}$$

*verifying whether the monitored system violates a property by first outputting a channel name on channel d at k, followed by an integer output that is less than 2 on this*

[8]Process migrating would have required traces to log this information as well, so as to enable accompanying monitors to trace the migration and thus move along with them. Although this enhancement could have been accommodated by our framework, none of the monitoring strategies surveyed in Section 2 dealt with migrating processes.

*channel at location l. When monitoring the system Sys from Example 4, subject to* $\delta = \{l \mapsto j, k \mapsto i\}$, *our semantics permits the following transition sequence:*

$$
\begin{aligned}
&\delta \triangleright\ Sys \parallel M^{orch} \\
&\xrightarrow{\tau} (\delta' = inc(\delta, k)) \triangleright Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel M^{orch} && (2)\\
&\xrightarrow{\tau} \delta' \triangleright Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\!\!\{\mathsf{sync}(l).\ \mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\}\!\!\}^{(k,i+1)} && (3)\\
&\xrightarrow{\tau} \delta' \triangleright Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\!\!\{\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\}\!\!\}^{(l,j)} && (4)\\
&\xrightarrow{\tau} inc(\delta', l) \triangleright Sys_1' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 1, j)]\!] \parallel m\{\!\!\{\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\}\!\!\}^{(l,j)} && (5)\\
&\xrightarrow{\tau} inc(\delta', l) \triangleright Sys_1' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 1, j)]\!] \parallel m\{\!\!\{\mathsf{if}\ 1 < 2\ \mathsf{then\ fail}\}\!\!\}^{(l,j+1)} && (6)\\
&\xrightarrow{\tau} inc(\delta', l) \triangleright Sys_1' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 1, j)]\!] \parallel m\{\!\!\{\mathsf{fail}\}\!\!\}^{(l,j+1)} && (7)
\end{aligned}
$$

*Transitions* (2) *and* (5) *are those derived earlier in Example 4, except that now they are derived in the context of a monitor using rule* Par. *These transitions are however now interleaved with transitions from the orchestrated monitor's side,* $M^{orch}$.

*In particular,* $\tau$*-transition* (3) *describes the querying of the trace at location k from index i onwards for recorded outputs on channel d. A trace record matching the query is immediately encountered at index i, namely* $k[\![\mathbf{t}(d, c, i)]\!]$, *and the successful reading of this record is derived using the rules* Com *(trace reading),* OutT *(trace broadcast) and* InT *(trace query); this latter rule increments the monitoring context index to* $(k, i + 1)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!] \xrightarrow{\mathbf{outT}(d,c,k,i)} \delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!]}\ \textsc{OutT} \quad \vdots \quad \dfrac{}{\delta' \triangleright M^{orch} \xrightarrow{\mathbf{inT}(d,c,k,i)} \delta' \triangleright m\{\!\!\{\mathsf{sync}(l).\ldots\}\!\!\}^{(k,i+1)} \parallel k[\![\mathbf{t}(d, c, i)]\!]}\ \textsc{InT}}{\delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!] \parallel M^{orch} \xrightarrow{\tau} \delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\!\!\{\mathsf{sync}(l).\ldots\}\!\!\}^{(k,i+1)} \parallel k[\![\mathbf{t}(d, c, i)]\!]}\ \textsc{Com}}{\delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!] \parallel M^{orch} \xrightarrow{\tau} \delta' \triangleright k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\!\!\{\mathsf{sync}(l).\ldots\}\!\!\}^{(k,i+1)}}\ \textsc{Str}}{\delta' \triangleright Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel M^{orch} \xrightarrow{\tau} \delta' \triangleright Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\!\!\{\mathsf{sync}(l).\ldots\}\!\!\}^{(k,i+1)}}\ \textsc{Par}
$$

*The full derivation for the* $\tau$*-transition* (3) *is given above and deserves some comment. In particular, we note that trace querying generates duplicate trace records* $k[\![\mathbf{t}(d, c, i)]\!]$ *immediately after the rule* Com *is applied; these can however be consolidated as one record using rule* Str *and the structural equivalence rule* sTrc *from Figure 4. Even though this may seem redundant at first, it allows us to analyse subsystems in compositional fashion. For instance, in the case of the right axiom in the above derivation, the behaviour of* $M^{orch}$ *was analysed without requiring the trace record form timestamp i at location k. In fact rule* InT *could have even allowed us to derive a different query, say reading a different value v communicated on channel d:*

$$
\dfrac{}{\delta' \triangleright M^{orch} \xrightarrow{\mathbf{inT}(d,v,k,i)} \delta' \triangleright m\{\!\!\{\mathsf{sync}(l).\ldots\}\!\!\}^{(k,i+1)} \parallel k[\![\mathbf{t}(d, v, i)]\!]}\ \textsc{InT}
$$

*Note however how the label matching condition in rule* Com *and the side condition in rule* Par *prevent this from being derived for our full system, which in turn guarantees* *that the configuration remains well-formed. We discuss this issue in more depth in section 3.3 — see example 8 and Lemma 1.*

*The silent transition* (4) *describes the realignment of the orchestrated monitor with the trace at location l so as to start monitoring for future trace records there, and is derived using rule* Sync. *Transition* (6) *is another trace reading computation (this time from the trace at location l) derived again using rules* Com, OutT *and* InT, *whereas the final τ-transition,* (7), *is a branching operation (using a variant of the rule* EqM.*)*

## 3.3. Calculus Expressivity

In this section we argue, though a series of examples, that the operational semantics presented in section 3.2.3 captures the characteristic behaviour expected of asynchronous monitoring systems executing in a distributed setting, which helps us to better understand the situations that may arise when analysing existing monitoring strategies or when designing new ones. In particular, we show how our calculus can model the various distributed monitoring strategies discussed in section 2. We also discuss how, despite all the intricate concurrent interleavings, the semantics still allows us to perform compositional analysis focussing on particular subsystems.

An inherent aspect of asynchronous runtime verification is that violation checks are only carried out on the *current* system execution. This means that an otherwise erroneous concurrent system may produce an interleaved execution that does not yield a violation, in which case no violation must be detected by the monitor. In such concurrent settings, runtime monitoring must also to contend with interleaved events from other executions and must be able to skip trace entries that do not pertain to the property being monitored. Distribution complicates monitoring even further, and may limit detection capabilities across (asynchronous) locations.

**Example 6.** *Recall again $M^{orch}$ from Example 3.*

$$\delta \triangleright \; Sys \parallel k[\![c!5]\!] \parallel k[\![c?x.\mathsf{stop}]\!] \parallel M^{orch}$$

$$\xrightarrow{\tau} (\delta' = inc(\delta, k)) \triangleright Sys \parallel k[\![\boldsymbol{t}(c, 5, i)]\!] \parallel M^{orch} \tag{8}$$

$$\xrightarrow{\tau} (\delta'' = inc(\delta', k)) \triangleright Sys_1 \parallel k[\![\boldsymbol{t}(c, 5, i)]\!] \parallel k[\![\boldsymbol{t}(d, c, i+1)]\!] \parallel M^{orch} \tag{9}$$

$$\xrightarrow{\tau} \delta'' \triangleright \left( \begin{array}{l} Sys_1 \parallel k[\![\boldsymbol{t}(c, 5, i)]\!] \parallel k[\![\boldsymbol{t}(d, c, i+1)]\!] \\ \parallel m\{\![\boldsymbol{q}(d, x).\mathsf{sync}(l).\ \boldsymbol{q}(x, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k, i+1)} \end{array} \right) \tag{10}$$

$$\xrightarrow{\tau} \delta'' \triangleright \left( \begin{array}{l} Sys_1 \parallel k[\![\boldsymbol{t}(c, 5, i)]\!] \parallel k[\![\boldsymbol{t}(d, c, i+1)]\!] \\ \parallel m\{\![\mathsf{sync}(l).\ \boldsymbol{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k, i+2)} \end{array} \right) \tag{11}$$

*During monitoring , trace query matching need not be immediate, as shown in the transition sequence above. Here $M^{orch}$ is instrumented over a slightly more complex system that includes the subsystem $k[\![c!5]\!] \parallel k[\![c?x.\mathsf{stop}]\!]$ apart from Sys; this larger system may yield the altered trace*

$$k[\![\boldsymbol{t}(c, 5, i)]\!] \;\parallel\; k[\![\boldsymbol{t}(d, c, i+1)]\!]$$

*after two transitions,* (8) *and* (9). *In particular, the trace record describing communication on channel d is now assigned a later timestamp, namely $i + 1$. Thus the trace* *query from $M^{orch}$ for outputs on channel d at index i will not match the record at index i, namely $k[\![\mathbf{t}(c, 5, i)]\!]$, since this record describes communication on channel c. As a result, the monitor context of $M^{orch}$ is increase to $i + 1$ in transition* (10) *through a combination of the rules* Com, OutT *and, most importantly,* Skp. *At this point a query can be matched as before, using* Com, OutT *and* inT, (11).

$$\begin{aligned}
&\delta \rhd\ Sys \parallel M^{orch}\\
&\xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\tau} \delta' \rhd Sys_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\![\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j)}\\
&\xrightarrow{\tau} inc(\delta', l) \rhd Sys_1'' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 2, j)]\!] \parallel m\{\![\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j)} &(12)\\
&\xrightarrow{\tau} inc(\delta', l) \rhd Sys_1'' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 2, j)]\!] \parallel m\{\![\mathsf{if}\ 2 < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j+1)} &(13)\\
&\xrightarrow{\tau} inc(\delta', l) \rhd Sys_1'' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 2, j)]\!] \parallel m\{\![\mathsf{stop}]\!\}^{(l,j+1)} &(14)
\end{aligned}$$

*As discussed in Example 4, system Sys is inherently non-deterministic, and after reaching $Sys_1$, it could transition to $Sys_1''$ instead. This behaviour does not lead to a violation of the property monitored by $M^{orch}$ and, accordingly, our transition semantics does not allow the monitor to detect a violation. This is shown in the transitions* (12), (13) *and* (14) *above, which can be derived using the same rules used to derive the corresponding transitions* (5), (6) *and* (7) *above.*

$$\begin{aligned}
&\delta \rhd\ Sys \parallel M^{orch}\\
&\xrightarrow{\tau} (\delta' = inc(\delta, k)) \rhd Sys_2 \parallel k[\![\mathbf{t}(d, b, i)]\!] \parallel M^{orch}\\
&\xrightarrow{\tau} \delta' \rhd Sys_2 \parallel k[\![\mathbf{t}(d, b, i)]\!] \parallel m\{\![\mathsf{sync}(l).\ \mathbf{q}(b, y).\ \mathsf{if}\ y < 2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i+1)} &(15)\\
&\xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\tau} inc(\delta', l) \rhd Sys_2' \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(b, 1, j)]\!] \parallel m\{\![\mathsf{fail}]\!\}^{(l,j+1)}
\end{aligned}$$

*Another possible transition sequence for Sys is the one shown above, discussed earlier in Example 1, transitioning through $Sys_2$ reaching $Sys_2'$ defined as:*

$$\begin{aligned}
Sys_2 &\triangleq l[\![b!1]\!] \parallel k[\![d!c.c!2]\!] \parallel k[\![b?z.P]\!] \parallel l[\![c?y.\mathsf{if}\ y{=}2\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2]\!]\\
Sys_2' &\triangleq l[\![\mathsf{stop}]\!] \parallel k[\![d!c.c!2]\!] \parallel k[\![P\{^1/_z\}]\!] \parallel l[\![c?y.\mathsf{if}\ y{=}2\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2]\!]
\end{aligned}$$

*This time, the channel name communicated on channel d is b, not c, and subsequently, the value* 1 *is outputted on channel b, as would be recorded in the residual trace. Nevertheless, this alternative computation also violates the property monitored for by $M^{orch}$ and accordingly, the transition sequence above shows how $M^{orch}$ can still detect this violation. In particular, after the trace reading transition* (15)*, the subsequent trace query in $M^{orch}$ is set to channel b at l (as opposed to channel c in the previous case), thereby adapting dynamically to the different trace analysis required.*

We note that mDPi focuses on correct monitors (soundness), *i.e.* flagging violations only when these actually happen, rather than monitor precision (completeness), *i.e.* guaranteeing detection whenever violation happens. As discussed earlier, the lack of global clocks in distributed settings prohibit tight synchronisation across locations, forcing monitoring to be inherently *asynchronous*, i.e. trace generation and trace monitoring are *not* in lock-step. Unfortunately, this limitation is inherent in distributed settings [16].

$$\begin{aligned}
&\delta \triangleright\ \text{Sys} \parallel M^{\text{orch}} \\
&\xrightarrow{\tau} (\delta' = \mathit{inc}(\delta, k)) \triangleright \text{Sys}_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel M^{\text{orch}} \\
&\xrightarrow{\tau} \delta' \triangleright \text{Sys}_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel m\{\![\mathsf{sync}(l).\ \mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k,i+1)} \\
&\xrightarrow{\tau} \mathit{inc}(\delta', l) \triangleright \text{Sys}'_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 1, j)]\!] \parallel m\{\![\mathsf{sync}(l).\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ldots]\!\}^{(k,i+1)} \\
&\xrightarrow{\tau} \mathit{inc}(\delta', l) \triangleright \text{Sys}'_1 \parallel k[\![\mathbf{t}(d, c, i)]\!] \parallel l[\![\mathbf{t}(c, 1, j)]\!] \parallel m\{\![\mathbf{q}(c, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(l,j+1)}
\end{aligned}$$

For instance, in the case of the first transition sequence of Example 5, this allows for (4) to potentially occur before (3) as shown above (or even before (2)). This yields a situation whereby, after synchronising with $l$, the monitor starts analysing the local trace at location $l$ at index $j + 1$, missing the trace event $l[\![\mathbf{t}(c, 1, j)]\!]$ as a result. Even though this may lead to the monitor *not detecting* a violation, this lack of precision is not a limitation that is exclusive to our present model but is also an inherent characteristic of distributed monitoring.

More importantly however, our semantics allows us also to derive similar transition sequences to the ones discussed for the orchestrated monitor instrumentation in Example 5, but for the different monitoring strategies discussed earlier in Example 3.

**Example 7.** *Recall the choreographed monitor, $M^{chor}$, defined earlier in as*

$$\mathsf{new}\, b. \Big(k\{\![\mathbf{q}(d, x).\ b!x]\!\}^{(k,i)} \ \parallel\ l\{\![b?x.\ \mathsf{sync}(l).\ \mathbf{q}(x, y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(l,h)}\Big).$$

*As can be shown below, this monitor can also detect the property violation discussed earlier for $M^{orch}$ when the configuration $\delta \triangleright Sys$ transits to $inc(inc(\delta, k), l) \triangleright Sys'_1$. In particular, transition* (16)*, describing trace reading, is derived using the rules* Com1*,* OutT *and* InT *as before, whereas transition* (17)*, describing process communication, is derived using the rules* Com1*,* OutM *and* InM*. Transition* (18) *describes trace alignment using rule* Sync*. Finally* (19) *is another case of a trace reading transition, which* *is followed by monitor branching.*

$$\delta \rhd\ Sys \parallel M^{chor} \xrightarrow{\tau} (\delta' = inc(\delta,k)) \rhd Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel M^{orch}$$

$$\xrightarrow{\tau} \delta' \rhd \begin{pmatrix} Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel \\ \mathsf{new}\, b.\left(k\{\![b!c]\!\}^{(k,i+1)} \parallel l\{\![b?x.\ \mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,h)}\right) \end{pmatrix} \quad (16)$$

$$\xrightarrow{\tau} \delta' \rhd \begin{pmatrix} Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel \\ \mathsf{new}\, b.\left(k\{\![\mathsf{stop}]\!\}^{(k,i+1)} \parallel l\{\![\mathsf{sync}(l).\ \mathbf{q}(c,y).\ \mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,h)}\right) \end{pmatrix} \quad (17)$$

$$\xrightarrow{\tau} \delta' \rhd \begin{pmatrix} Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel \\ \mathsf{new}\, b.\left(k\{\![\mathsf{stop}]\!\}^{(k,i+1)} \parallel l\{\![\mathbf{q}(c,y).\ \mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j)}\right) \end{pmatrix} \quad (18)$$

$$\xrightarrow{\tau} inc(\delta',l) \rhd \begin{pmatrix} Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel \\ \mathsf{new}\, b.\left(k\{\![\mathsf{stop}]\!\}^{(k,i+1)} \parallel l\{\![\mathbf{q}(c,y).\ \mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j)}\right) \end{pmatrix}$$

$$\xrightarrow{\tau} inc(\delta',l) \rhd \begin{pmatrix} Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel \\ \mathsf{new}\, b.\left(k\{\![\mathsf{stop}]\!\}^{(k,i+1)} \parallel l\{\![\mathsf{if}\ 1<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j+1)}\right) \end{pmatrix} \quad (19)$$

$$\xrightarrow{\tau} inc(\delta',l) \rhd Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel \mathsf{new}\, b.\left(k\{\![\mathsf{stop}]\!\}^{(k,i+1)} \parallel l\{\![\mathsf{fail}]\!\}^{(l,j+1)}\right)$$

*The same transition sequence leading to a trace violation detection can be derived as well if the configuration $\delta \rhd Sys$ is instrumented with a migrating-monitor $M^{mig}$; we recall that this monitor was defined earlier in Example 3 as*

$$k\{\![\mathbf{q}(d,x).\mathsf{go}\ l.\mathsf{sync}(l).\mathbf{q}(x,y).\mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i)}.$$

*This transition sequence below looks very much like the one discussed earlier for the orchestrated monitor instrumentation, $M^{orch}$, in Example 5. Transition* (20) *describes trace reading. Transition* (21) *is novel to what we discussed so far, and describes monitor migration; it is derived using rule* Go. *This is followed by a trace realignment transition,* (22)*, a process communication, generating the record $l[\![\mathbf{t}(c,1,j)]\!]$, followed by a trace reading of this record,* (23)*, and the final monitor branching leading to the violation detection.*

$$\delta \rhd\ Sys \parallel M^{mig} \xrightarrow{\tau} (\delta' = inc(\delta,k)) \rhd Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel M^{mig}$$

$$\xrightarrow{\tau} \delta' \rhd Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel k\{\![\mathsf{go}\ l.\mathsf{sync}(l).\mathbf{q}(c,y).\mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i+1)} \quad (20)$$

$$\xrightarrow{\tau} \delta' \rhd Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l\{\![\mathsf{sync}(l).\mathbf{q}(c,y).\mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(k,i+1)} \quad (21)$$

$$\xrightarrow{\tau} \delta' \rhd Sys_1 \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l\{\![\mathbf{q}(c,y).\ \mathsf{if}\ y<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j)} \quad (22)$$

$$\xrightarrow{\tau} \cdot \xrightarrow{\tau} inc(\delta',l) \rhd Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel l\{\![\mathsf{if}\ 1<2\ \mathsf{then}\ \mathsf{fail}]\!\}^{(l,j+1)} \quad (23)$$

$$\xrightarrow{\tau} inc(\delta',l) \rhd Sys_1' \parallel k[\![\mathbf{t}(d,c,i)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel l\{\![\mathsf{fail}]\!\}^{(l,j+1)}$$

*It is worth noting that, in the transition sequences for both $M^{chor}$ and $M^{mig}$ shown above, trace reading is always performed locally, as opposed to the transition sequence for $M^{orch}$. We also note how, in the case of $M^{mig}$, instrumentation at location $l$ is only carried out (through dynamic migration) on a by-need basis. In fact, if the query for* *outputs on channel d at location k is never satisfied, then no monitor is ever instrumented at location l.*

It is important to highlight the fact that the formal analysis carried out so far was performed under a closed world setting, *i.e.* we assume that the system being analyses is the entire system under consideration. In fact, the transitions we considered were in fact $\tau$ transitions, which correspond to more traditional reduction-based operational semantics *i.e.* without any labels. Our semantics, however, allows us to perform a similar analysis under an open world setting i.e. in the presence of other systems executing in parallel. An open-world semantics allows us to analyse a system compositionally.

**Example 8.** *Recall the (closed world) transition sequence* (2) *up to* (7) *from Example 5, or even the sequence from* (8) *up to* (11) *from the same Example. Through external actions, our semantics allows us to model such computation while abstracting away from the system generating the corresponding localised traces (such as Sys* $\parallel k[\![c!5]\!] \parallel k[\![c?x.\mathsf{stop}]\!]$*), thereby focussing on the monitor code in isolation.*

$$\delta \rhd M^{orch} \xrightarrow{\textbf{\textit{tick}}} (\delta' = inc(\delta, k)) \rhd M^{orch} \tag{24}$$

$$\xrightarrow{\textbf{\textit{tick}}} (\delta'' = inc(\delta, k)) \rhd M^{orch} \tag{25}$$

$$\xrightarrow{\textbf{\textit{inT}}(c,5,k,i)} \delta'' \rhd k[\![\mathbf{t}(c,5,i)]\!] \parallel m\{\![\mathbf{q}(d,x).\mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k,i+1)} \tag{26}$$

$$\xrightarrow{\textbf{\textit{inT}}(d,c,k,i+1)} \delta'' \rhd \left( \begin{array}{l} k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \\ \parallel m\{\![\mathsf{sync}(l).\ \mathbf{q}(c,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(k,i+2)} \end{array} \right) \tag{27}$$

$$\xrightarrow{\tau} \delta'' \rhd k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \parallel m\{\![\mathbf{q}(c,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}]\!\}^{(l,j)} \tag{28}$$

$$\xrightarrow{\textbf{\textit{tick}}} inc(\delta'', l) \rhd k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \parallel m\{\![\mathbf{q}(c,y).\ \mathsf{if}\ y < 2 \ldots]\!\}^{(l,j)} \tag{29}$$

$$\xrightarrow{\textbf{\textit{inT}}(c,1,l,j)} inc(\delta'', l) \rhd \left( \begin{array}{l} k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \parallel\parallel l[\![\mathbf{t}(c,1,j)]\!] \\ \parallel m\{\![\mathsf{if}\ 1 < 2\ \mathsf{then\ fail}]\!\}^{(l,j+1)} \end{array} \right) \tag{30}$$

$$\xrightarrow{\tau} inc(\delta'', l) \rhd k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \parallel l[\![\mathbf{t}(c,1,j)]\!] \parallel m\{\![\mathsf{fail}]\!\}^{(l,j+1)} \tag{31}$$

*The transition sequence above is one such example. Transitions* (24) *and* (25)*, derived using rule* CNTR*, model the effect of the corresponding process communication transitions such as* (8) *and* (9)*, performed by the system abstracted away by the open world semantics. The trace records generated by these communications are then queried through the external actions* (26) *and* (27)*, derived using rules* SKP *and* INT *respectively, which correspond to the* $\tau$*-transitions* (10) *and* (11) *discussed earlier. We here note how querying these trace records figuratively brings them in from the abstracted context to form part of the system being analysed, i.e., the system being analysed is now extended with the trace records* $k[\![\mathbf{t}(c,5,i)]\!]$ *and* $k[\![\mathbf{t}(d,c,i+1)]\!]$*. Transition* (28) *corresponds to* (4) *from Example 5 whereas* (29) *and* (30) *model the trace record generation and then the trace query computations described by* (5) *and* (6) *from Example 5.*

*Finally* (31) *corresponds to* (7)*, again from Example 5.*

$$\delta \triangleright M^{orch} \parallel M^{orch} \xrightarrow{\textit{tick}} \cdot \xrightarrow{\textit{tick}} (\delta'' = inc(inc(\delta,k),k)) \triangleright M^{orch} \parallel M^{orch}$$

$$\xrightarrow{\textit{inT}(c,5,k,i)} \delta'' \triangleright \left( \begin{array}{l} k[\![\mathbf{t}(c,5,i)]\!] \parallel M^{orch} \\ \parallel m\{\![\mathbf{q}(d,x).\mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y<2\ \mathsf{then\ fail}]\!\}^{(k,i+1)} \end{array} \right)$$

$$\xrightarrow{\textit{inT}(d,c,k,i+1)} \delta'' \triangleright \left( \begin{array}{l} k[\![\mathbf{t}(c,5,i)]\!] \parallel k[\![\mathbf{t}(d,c,i+1)]\!] \parallel M^{orch} \\ \parallel m\{\![\mathsf{sync}(l).\ \mathbf{q}(c,y).\ \mathsf{if}\ y<2\ \mathsf{then\ fail}]\!\}^{(k,i+2)} \end{array} \right) \qquad (32)$$

*The reason why trace records are brought in as part of the system being analysed becomes apparent from the example derivation above where two parallel copies of the same monitor $M^{orch}$ are considered. An appropriate semantics should ensure that for certain execution interleavings, the two monitors would reach the same verdict because they query the same trace. The four transitions leading to* (32) *above correspond to the earlier transitions* (24) *to* (27)*. At this point, if the second copy of the monitor $M^{orch}$ is to query the trace records from $(k,i)$ onwards, it should not be able to query and introduce records other than $k[\![\mathbf{t}(c,5,i)]\!]$ and $k[\![\mathbf{t}(d,c,i+1)]\!]$. Accordingly, the side-condition of rule* ParT *from Figure 6 prevents us from using external trace querying (through the rules* Skp *and* InT*), and any trace querying can only be derived using* Com*, which forces the second copy of the monitor to react with the already present internal trace records $k[\![\mathbf{t}(c,5,i)]\!]$ and $k[\![\mathbf{t}(d,c,i+1)]\!]$.*

## *3.4. Bisimulation-based Equivalences.*

In this section we use the LTS defined in section 3.2.3 to define a notion of system equivalence in an open-world setting. We then prove the main result of the section, Theorem 1, which states that this equivalence is a congruence *wrt.* parallel composition, and thus a sound equivalence proof technique for our open-world semantics. Before, however, we prove an important sanity check of our LTS, namely that well-formed configurations are preserved by the transition rules which guarantees that, starting from a well-formed configuration, subsequent analysis is also made on well-formed configurations.

**Lemma 1** (Preservation of Well-formed Configurations)**.** *Given a well-formed configuration $\delta \triangleright S$, and transition $\delta \triangleright S \xrightarrow{\mu} \delta' \triangleright S'$, the resulting configuration $\delta' \triangleright S'$ is also well-formed.*

*Proof.* The proof proceeds by rule induction on the transition relation. We here consider two cases.

**OutP:** We know $S = k[\![c!\bar{v}.P]\!]$, $\delta' = \mathsf{inc}(\delta,k)$ and $S' = k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]$ for some $k, c$ and $\bar{v}$. From the structure of $S'$ we know that there is only one trace record to consider, $k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]$, and thus the second condition in Definition 1 is immediately satisfied. This trace record, $k[\![\mathbf{t}(c,\bar{v},\delta(k))]\!]$, also satisfies the first condition: from $\delta' = \mathsf{inc}(\delta,k) = \delta[k \mapsto (\delta(k)+1)]$ we know that the index of this trace record, *i.e.*, $\delta(k)$, is strictly less than $\delta'(k)$, *i.e.*, $\delta(k)+1$.

**Com:** We know $S = S_1 \parallel S_2$, $S' = \mathsf{new}\,\bar{b}.(S'_1 \parallel S'_2)$ with transitions

$$\delta \triangleright S_1 \xrightarrow{(\bar{b})\bar{\mu}} \delta' \triangleright S'_1 \qquad (33)$$

$$\text{and} \qquad \delta \triangleright S_2 \xrightarrow{\mu} \delta \triangleright S'_2 \qquad (34)$$

Given that $\delta \triangleright S_1 \parallel S_2$ is well-formed, then both $\delta \triangleright S_1$ and $\delta \triangleright S_2$ are also well-formed, since removing $S_1$ or $S_2$ from $\delta \triangleright S_1 \parallel S_2$ cannot break any of the conditions in Definition 1. By the inductive hypothesis, we hence infer that $\delta' \triangleright S'_1$ and $\delta \triangleright S'_2$ are also well-formed. We have three subcases to consider:

- $\mu = \mathbf{inP}(c, \overline{v})$: We know that $\overline{\mu} = \mathbf{outP}(c, \overline{v})$ and, by an appropriate sublemma, from (33) we can conclude that $\delta' = \mathsf{inc}(\delta, k)$ for some $k$, and that $S'_1 \equiv (\overline{b})(S''_1 \parallel k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!])$ where $S''_1$ has the same trace records as $S_1$; this means that $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$ is the only new trace record generated. From (34) and an appropriate sublemma we can also deduce that no new trace records are generated in $S'_2$ (apart from possible trace record replication through the structural rule sTrc which does not violate Definition 1). This means that to ensure well-formedness for $\delta' \triangleright \mathsf{new}\,\overline{b}.(S'_1 \parallel S'_2)$, we only need to check that the conditions in Definition 1 are satisfied by the new trace record generated, $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$. Since $\delta \triangleright S_1 \parallel S_2$ is well-formed, then by Definition 1(1), all trace records at $k$ have indexes that are strictly less than $\delta(k)$; this makes $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$ unique in $\delta' \triangleright \mathsf{new}\,\overline{b}.(S'_1 \parallel S'_2)$, thereby trivially satisfying Definition 1(2). Moreover, $k[\![\mathbf{t}(c, \overline{v}, \delta(k))]\!]$ satisfies Definition 1(1), since $\delta' = \mathsf{inc}(\delta, k)$.
- $\mu = \mathbf{inM}(c, \overline{v})$: Analogous to the previous case but simpler as it does not involve the generation of new trace records. More precisely, we know that $\delta' = \delta$ and as a result the well-formedness of $\delta' \triangleright \mathsf{new}\,\overline{b}.(S'_1 \parallel S'_2)$ follows from that of $\delta \triangleright S_1 \parallel S_2$.

$\mu = \mathbf{inT}(c, \overline{v}, l, i)$: We know that $\overline{\mu} = \mathbf{outT}(c, \overline{v}, l, i)$, which, by an appropriate sublemma, (33) implies that $\delta' = \delta$ and that $S_1 \equiv (\overline{b})(S''_1 \parallel l[\![\mathbf{t}(c, \overline{v}, i)]\!]) \equiv S'_1$. From (34) and an appropriate sublemma we know that $S'_2 \equiv (S''_2 \parallel l[\![\mathbf{t}(c, \overline{v}, i)]\!])$ where $S''_2$ has the same trace records as $S_2$. This ensures that Definition 1(2) is satisfied by $\delta \triangleright \mathsf{new}\,\overline{b}.(S'_1 \parallel S'_2)$. Moreover the satisfaction of Definition 1(1) follows from the well-formedness of $\delta \triangleright S_1 \parallel S_2$.

□

Our LTS semantics for mDPi induces an intuitive definition of program equivalences, centered around the concept of bisimulations [13]. We here opt for the most natural version, *i.e.* weak bisimulation, whereby silent actions, $\tau$, are not considered visible; this definition relies on the concept of weak actions, $\overset{\hat{\alpha}}{\Longrightarrow}$, action transitive closures that abstract away from silent actions, defined as $(\xrightarrow{\tau})^*$ if $\alpha = \tau$ and $(\xrightarrow{\tau})^* \cdot \xrightarrow{\alpha} \cdot (\xrightarrow{\tau})^*$ otherwise.

**Definition 2** (Bisimulation)**.** *A family of (binary) relation over systems, indexed by a set of local clocks $\bigcup_{\delta \in \textsc{Clocks}} \mathcal{R}_\delta$ is said to be a bisimulation iff whenever $S_1 \; \mathcal{R}_\delta \; S_2$, $\delta \triangleright S_1$ and $\delta \triangleright S_2$ are configurations, then:*

- *$\delta \triangleright S_1 \xrightarrow{\alpha} \delta' \triangleright S'_1$ implies $\delta_2 \triangleright S_2 \overset{\hat{\alpha}}{\Longrightarrow} \delta' \triangleright S'_2$ such that $S'_1 \; \mathcal{R}'_\delta \; S'_2$;*
- *$\delta \triangleright S_2 \xrightarrow{\alpha} \delta' \triangleright S'_2$ implies $\delta \triangleright S_1 \overset{\hat{\alpha}}{\Longrightarrow} \delta' \triangleright S'_1$ such that $S'_1 \; \mathcal{R}'_\delta \; S'_2$*

Bisimilarity, denoted as $\approx$, is the largest set of indexed relation satisfying Definition 2. It comes equipped with an elegant proof technique: in order to show that two systems $S_1$ and $S_2$ are bisimilar with respect to the local clocks $\delta$, denoted as

$\delta \models S_1 \approx S_2$, *i.e.*, the pair of systems $\langle S_1, S_2\rangle$ is included in the bisimilarity relation, $\approx$ at index $\delta$, it suffices to give a family of relations that includes this pair at the relation indexed by $\delta$, and where this family of relations satisfies the transfer properties of Definition 2. Since bisimilarity is the largest such set of indexed relations satisfying these transfer properties, this implies that are $S_1$ and $S_2$ are bisimilar *wrt.* $\delta$ (see [15, 13] for details).

It is easy to show that bisimilarity is an *equivalence* relation, *i.e.*, it is reflexive, symmetric and transitive. Importantly, we also prove *contextuality*, *i.e.* Theorem 1, which states that whenever we prove that two configurations are bisimilar, they remain so under larger (system) contexts. This theorem in effect ensures that bisimilarity is a congruence (with respect to large systems) and justifies the use of bisimilarity as a sensible behavioural equivalence. We use bisimulation as our touchstone equivalence in the rest of the paper.

**Theorem 1** (Contextuality)**.** *Under any context $C :: \textsc{Sys} \to \textsc{Sys}$, where $\delta \triangleright C(S_1)$ and $\delta \triangleright C(S_2)$ are configurations:*

$$\delta \models S_1 \approx S_2 \quad \textit{implies} \quad \delta \models C(S_1) \approx C(S_2)$$

*Proof.* The proof is by coinduction. We define the family of relations $\bigcup_{\delta\in\textsc{Clocks}} \mathcal{R}_\delta$ as follows:

- $\delta \models S_1 \approx S_2$ implies $\langle S_1, S_2\rangle \in \mathcal{R}_\delta$
- $\langle S_1, S_2\rangle \in \mathcal{R}_\delta$ and $\delta \triangleright (S_1 \parallel S_3)$, $\delta \triangleright (S_2 \parallel S_3)$ are configurations implies $\langle (S_1 \parallel S_3), (S_2 \parallel S_3)\rangle \in \mathcal{R}_\delta$
- $\langle S_1, S_2\rangle \in \mathcal{R}_\delta$ and $\delta \triangleright (S_3 \parallel S_1)$, $\delta \triangleright (S_3 \parallel S_1)$ are configurations implies $\langle (S_3 \parallel S_1), (S_3 \parallel S_2)\rangle \in \mathcal{R}_\delta$
- $\langle S_1, S_2\rangle \in \mathcal{R}_\delta$ implies $\langle \mathsf{new}\, c.S_1, \mathsf{new}\, c.S_2\rangle \in \mathcal{R}_\delta$

We then show that $\bigcup_{\delta\in\textsc{Clocks}} \mathcal{R}_\delta$ satisfies the transfer property of Definition 2, which would imply that $(\bigcup_{\delta\in\textsc{Clocks}} \mathcal{R}_\delta) \subseteq \approx$ and hence that, whenever $\delta \models S_1 \approx S_2$, it remains so under larger system contexts.

We proceed by induction on how each $\mathcal{R}_\delta$ is defined. The base case, *i.e.* when $\langle S_1, S_2\rangle \in \mathcal{R}_\delta$ because $\delta \models S_1 \approx S_2$ is immediate. The other three cases are the inductive cases where we here outline the proof for the more involving subcase of the second case; the subcases for the third case are analogous whereas the subcases for the fourth case are simpler.

We thus consider the case where $\langle (S_1 \parallel S_3), (S_2 \parallel S_3)\rangle \in \mathcal{R}_\delta$ because

$$\langle S_1, S_2\rangle \in \mathcal{R}_\delta, \tag{35}$$

and consider the case where

$$\delta \triangleright S_1 \parallel S_3 \xrightarrow{\alpha} \delta' \triangleright S_4 \tag{36}$$

We are required to show that there exists a transition $\delta \triangleright S_2 \parallel S_3 \overset{\hat{\alpha}}{\Longrightarrow} \delta' \triangleright S_5$ such that $S_4 \;\mathcal{R}_{\delta'}\; S_5$. By case analysis and the structure of the configuration $\delta \triangleright S_1 \parallel S_3$, we know that (36) could have been generated using the rules Com1 (or its dual Com2), Par1 (or its dual Par2), ParT1 (or its dual ParT2), or else Skp1 (or its dual Skp2). We here consider the case when (36) was inferred using rule Com1; the other cases are analogous. From Com1, we know (36) takes the form

$$\delta \triangleright S_1 \parallel S_3 \xrightarrow{\tau} \delta' \triangleright \mathsf{new}\,\overline{b}.(S_1{}' \parallel S_3{}') \quad \textit{s.t.} \;\; \overline{b} \cap \mathit{fn}(S_3') = \emptyset \tag{37}$$

because

$$\delta \triangleright S_1 \xrightarrow{\overline{\alpha}} \delta' \triangleright S_1{}' \tag{38}$$

$$\delta \triangleright S_3 \xrightarrow{\alpha} \delta \triangleright S_3{}' \tag{39}$$

Now by (35), (38) and the inductive hypothesis, we can infer a matching transition from $\delta \triangleright S_2$, *i.e.*,

$$\delta \triangleright S_2 \overset{\overline{\alpha}}{\Longrightarrow} \delta' \triangleright S_2{}' \tag{40}$$

$$\textit{s.t.} \quad S_1' \,\mathcal{R}_{\delta'}\, S_2{}'. \tag{41}$$

Thus, from (40), (39) and Com1 we infer transition

$$\delta_2 \triangleright S_2 \parallel S_3 \overset{\tau}{\Longrightarrow} \delta_2' \triangleright \mathsf{new}\,\overline{b}.(S_2{}' \parallel S_3{}')$$

which is the required matching move. Finally, from (41) we obtain

$$\mathsf{new}\,\overline{b}.(S_1{}' \parallel S_3{}')\;\mathcal{R}_{\delta'}\;\mathsf{new}\,\overline{b}.(S_2{}' \parallel S_3{}')$$

by the second and fourth clauses of the definition of $\mathcal{R}_\delta$. □

Apart from justifying bisimulation as a sensible equivalence relation for our terms, Theorem 1 implies that bisimulation admits compositional analysis. In fact, this theorem allows us to abstract away from common code when exhibiting bisimulations. More precisely, to show that two configurations $\delta_1 \triangleright S_1 \parallel S_3$ and $\delta_1 \triangleright S_2 \parallel S_3$ are bisimilar, it suffices to provide a relation including the pair $\langle \delta_1 \triangleright S_1 \parallel S_3, \delta_1 \triangleright S_2 \parallel S_3\rangle$ without considering the common sub-system $S_3$. We will take advantage of these compositional properties when proving bisimulations in Section 4.3.

**Example 9.** *We can show that the monitors $M^{orch}$ and $M^{chor}$, defined earlier in Example 3, when monitoring the system Sys, defined in Example 1, subject to the local clocks $\delta = \{l \mapsto j, k \mapsto i\}$, are bisimilar:*

$$\delta \models (\mathit{Sys} \parallel M^{orch}) \approx (\mathit{Sys} \parallel M^{chor})$$

*Moreover, by Theorem 1, in order to show this it suffices to show:*

$$\delta \models M^{orch} \approx M^{chor}$$

*This can be shown by giving a concrete bisimulation.*

In general, constructing concrete bisimulation relations for particular cases can be a tedious task. However, the contextuality result of Theorem 1 allows us to reason about monitors by decomposing them into parts. In practice, one usually has compositional ways of synthesising monitors from a logic formula. The contextuality of bisimulation lends itself directly to proving properties about a general synthesis approach as we will show in the next section.

## 4. Comparing Monitoring Strategies

mDPi provides us with mathematical tools enabling us to reason about monitoring of systems. As we have seen in example 3, one can express monitors instrumented in different ways and also to reason about their observational equivalence, despite syntactic and structural differences. Typically, in runtime verification, one expresses properties to be monitored in a logic which is then instrumented on the system in whichever way best suits the setting. In general, a property may thus be translated into different monitors — depending on how one would like the monitoring to take place. Correctness of the instrumentation processes corresponds to saying that starting from any property expressible in the logic, different instrumentations would give observationally equivalent results.

In this section we illustrate this use of mDPi to prove that three different instrumentation strategies for a regular-expression based logic give observationally equivalent results. Furthermore, we also use the calculus, but taking location into account to show that certain instrumentation strategies ensure that monitoring is always performed locally. It should be emphasised that the choice of regular expressions to specify properties is an arbitrary one, and similar results can be shown for other logics.

### *4.1. Regular Expressions for Monitoring*

A regular-expression based logic is used to express properties to be monitored. Well-formed expressions in the logic range over the following syntax:

$$E, F \in \textsc{RegExp} \ ::= \ (c,\overline{v})@k \ \mid \ (c,\exists\overline{x})@k.E \ \mid \ E+F \ \mid \ E^{*} \ \mid \ E.F$$

Standard regular expression combinators are adopted, with union over two expressions being written as $E + F$, repetition as $E^*$ and sequentiality as $E.F$. The basic component $(c,\overline{v})@k$ matches when value $\overline{v}$ is passed over channel $c$ emanating from location $k$, while $(c,\exists\overline{x})@k.E$ matches *any* communication over channel $c$ from location $k$, binding the value to variables $\overline{x}$ in the regular expression $E$. Any communication not matching the regular expression is ignored when matching. For example, the regular expression $(a,7)@l.(b,8)@l$ matches communication traces such as $\langle(a,7)@l,\ (b,8)@l\rangle$, $\langle(a,7)@l,\ (c,7)@l,\ (b,8)@l\rangle$ and $\langle(a,7)@l,\ (a,7)@l,\ (b,8)@l\rangle$ — a trace matches if and only if terminates with $(b,8)@l$ and includes an earlier $(a,7)@l$. We use regular expressions to specify counterexamples in the logic — any trace matching with the regular expression is considered to be a violation.

**Example 10.** *Despite its simplicity, the logic is sufficiently expressive for many useful properties. Consider the property, which states that: "If an alarm is raised (a message is sent on channel alarm) from either location $k_1$ or $k_2$, then no further messages should be sent from location $k_0$ on channel pvt". This can be written as: $((alarm,\langle\rangle)@k_1 + (alarm,\langle\rangle)@k_2).(pvt,\langle\rangle)@k_0$. For simplicity, we use dataless communication, but this can easily be extended using the binding match operator.*

*Now consider the property: "Data passed over channel c emanating from location k may not repeat values". This can be expressed using the binding match operator as the regular expression: $(c,\exists x)@k.\ (c,x)@k$. Similarly, the operator can be used to reason about systems with dynamic topologies e.g. $(c,\exists badloc)@mng.(sys,\langle\rangle)@badloc$*

*would monitor whether the system channel can be accessed from locations which have been reported as bad by the manager at location* mng.

### *4.2. Monitoring of Regular Expressions*

We will now look at different ways of instrumenting an mDPi monitor from a regular expression, which we can then compare for observational equivalence. Unlike existing work on runtime verification with regular expressions for monolithic systems such as [17], we will provide different ways of synthesising a monitor depending on the monitoring strategy we choose to adopt.

#### *4.2.1. Orchestrated Monitoring*

Orchestrated monitoring of a regular expression places all the listening components in a central location, combining the information coming from different locations to try to match the expression.

Consider the continuation-based compilation function *compile* : RegExp × Mon → Mon, where *compile*$(E, M)$ is an mDPi term describing the monitor which, after matching expression $E$ behaves as mDPi term $M$:

$$\begin{array}{rcl}
\mathit{compile}_O & :: & \textsc{RegExp} \times \textsc{Mon} \to \textsc{Mon} \\
\mathit{compile}_O((c, \overline{v})@k, M) & \triangleq & \mathit{read}((c, \overline{v})@k, M) \\
\mathit{compile}_O((c, \exists\overline{x})@k.E, M) & \triangleq & \mathsf{sync}(k).\mathsf{new}\, d.\big(d! \parallel *d?.\mathbf{q}(c, \overline{x}).(\mathit{compile}_O(E, M) \parallel d!)\big) \\
\mathit{compile}_O(E + F, M) & \triangleq & \mathsf{new}\, d.(\mathit{compile}_O(E, d!) \parallel \mathit{compile}_O(F, d!) \parallel *d?.M) \\
\mathit{compile}_O(E^*, M) & \triangleq & \mathsf{new}\, b, d.\big((*d?.\mathit{compile}_O(E, b!)) \parallel (*b?.(d! \parallel M)) \parallel b!\big) \\
\mathit{compile}_O(E.F, M) & \triangleq & \mathit{compile}_O(E, \mathit{compile}_O(F, M))
\end{array}$$

Sequential composition $E.F$ corresponds directly to a continuation, while the binding operator is a combination of the base case and sequential composition, exploiting the binding and substitution associated with querying to instantiate free variables $\overline{x}$ in the continuation *compile*$_O(E, M)$ with the values obtained dynamically from the trace. Choice $E + F$ adds a listener to identify whether $E$ and $F$ has terminated before triggering the continuation. A mathematically equivalent way of expressing *compile*$_O(E + F, M)$ is as *compile*$_O(E, M) \parallel$ *compile*$_O(F, M)$. However, we chose this description since it explicitly refers to a single instance of the continuation monitor, and is therefore closer to the intended implementation. Repetition $E^*$ also adds piping to trigger the continuation repeatedly.

The case of $(c, \overline{v})@k$ is defined exclusively in terms of *read*$((c, \overline{v})@k, M)$, which waits for value $\overline{v}$ over channel $c$ in location $k$ before triggering the continuation $M$, and is defined as follows:

$$\mathit{read}((c, \overline{v})@k, M) \quad \triangleq \quad \mathsf{sync}(k).\mathsf{new}\, d.(d! \parallel *d?.(\mathbf{q}(c, x).\mathsf{if}\ \overline{x} = \overline{v}\ \mathsf{then}\ (M \parallel d!)\ \mathsf{else}\ d!))$$

We can now define the orchestrated monitor of expression $E$ *monitor*$_O(E)$ using *compile*$_O$ with a failing continuation to ensure violation whenever the expression is matched, and starting at an arbitrary (central) location $h$; the monitor resides in this location $h$ throughout its entire lifetime. The definition is given below:

$$\mathit{monitor}_O(E) \quad \triangleq \quad h[\![\mathit{compile}_O(E, \mathsf{fail})]\!]^{(h,1)}$$

**Example 11.** *Consider the example seen earlier with ensuring that once a bad location has been identified by a manager process, it is not used to send a system message:* $E \triangleq (c, \exists badloc)@mng.(sys, \langle\rangle)@badloc$. *A centralised monitor for* $E$ *can be calculated as follows:*

$$
\begin{array}{rl}
 & monitor_O(E) \\
= & h[\![compile_O(E, \mathsf{fail})]\!]^{(h,1)} \\
= & h[\![compile_O((c, \exists badloc)@mng.(sys, \langle\rangle)@badloc, \mathsf{fail})]\!]^{(h,1)} \\
= & h[\![\mathsf{sync}(mng).\mathsf{new}\, d.\big(d! \parallel *d?.\mathbf{q}(c, badloc).(compile_O((sys, \langle\rangle)@badloc, \mathsf{fail}) \parallel d!)\big)]\!]^{(h,1)} \\
= & h[\![\mathsf{sync}(mng).\mathsf{new}\, d.\big(d! \parallel *d?.\mathbf{q}(c, badloc).(read((sys, \langle\rangle)@badloc, \mathsf{fail}) \parallel d!)\big)]\!]^{(h,1)}
\end{array}
$$

*4.2.2. Migrating Monitors*

As discussed in section 2.1.3, instead of listening from a central location, an alternative is to migrate the monitoring code to the location where the next event is expected to occur. As in the case of orchestrated monitoring, we start by defining a continuation-based migrating monitor:

$$
\begin{array}{rcl}
compile_M & :: & \textsc{RegExp} \times \textsc{Mon} \rightarrow \textsc{Mon} \\
compile_M((c, \overline{v})@k, M) & \triangleq & \mathsf{go}\; k.read((c, \overline{v})@k, M) \\
compile_M((c, \exists\overline{x})@k.E, M) & \triangleq & \mathsf{go}\; k.\mathsf{sync}(k).\mathsf{new}\, d.\big(d! \parallel *d?.\mathbf{q}(c, \overline{x}).(compile_M(E, M) \parallel d!)\big) \\
compile_M(E + F, M) & \triangleq & \mathsf{new}\, d.(compile_M(E, d!) \parallel compile_M(F, d!) \parallel *d?.M) \\
compile_M(E^*, M) & \triangleq & \mathsf{new}\, b, d.\big((*d?.compile_M(E, b!)) \parallel (*b?.(d! \parallel M)) \parallel b!\big) \\
compile_M(E.F, M) & \triangleq & compile_M(E, compile_M(F, M))
\end{array}
$$

The schemata follow those used for orchestrated monitoring, but differ for the base cases of the definition: whenever a channel communication is to be monitored via $(c, \overline{v})@k$ or $(c, \exists\overline{x})@k.E$, the monitor *first migrates* to location $k$ where the trace query is to take place (followed by a synchronisation to that location), thereby guaranteeing that monitoring is always performed locally.

As before, we can now define the complete monitor by using a failing continuation to mark violations:

$$
monitor_M(E) \;\triangleq\; k[\![compile_M(E, \mathsf{fail})]\!]^{(k,1)}
$$

**Example 12.** *Once again, we will use the example disallowing system messages from a bad location identified at runtime:* $E \triangleq (c, \exists badloc)@mng.(sys, \langle\rangle)@badloc$. *A migrating monitor can be calculated as:*

$$
\begin{array}{rl}
 & monitor_M(E) \\
= & k[\![compile_M(E, \mathsf{fail})]\!]^{(k,1)} \\
= & k[\![compile_M((c, \exists badloc)@mng.(sys, \langle\rangle)@badloc, \mathsf{fail})]\!]^{(k,1)} \\
= & k[\![\mathsf{go}\; mng.\mathsf{sync}(mng).\mathsf{new}\, d. \\
 & \qquad \big(d! \parallel *d?.\mathbf{q}(c, badloc).(compile_M((sys, \langle\rangle)@badloc, \mathsf{fail}) \parallel d!)\big)]\!]^{(k,1)} \\
= & k[\![\mathsf{go}\; mng.\mathsf{sync}(mng).\mathsf{new}\, d. \\
 & \qquad \big(d! \parallel *d?.\mathbf{q}(c, badloc).(\mathsf{go}\; badloc.read((sys, \langle\rangle)@badloc, \mathsf{fail}) \parallel d!)\big)]\!]^{(k,1)}
\end{array}
$$

*Note that the only difference between this and orchestrated monitoring, is that the monitor migrates to the location where the communication will take place before eavesdropping.*

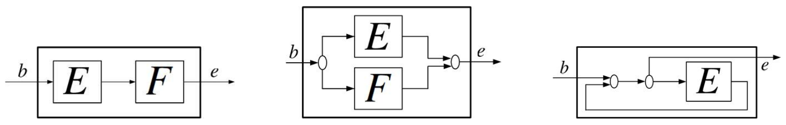

Figure 8: Compiling $E.F$, $E + F$ and $E^*$ (respectively).

*4.2.3. Static Choreography*

The third option for instrumenting a monitor for a regular expression is that of a choreographed approach — statically breaking the property down into communicating components each of which resides in the location where the monitored communication is to take place. The compilation decomposes the monitor into parallel components, resulting in a compilation strategy not unlike standard approaches used in hardware compilation of regular expressions e.g. [18]. As typically done in these approaches, we use two additional channels: $b$ and $e$, with $b$ signalling when to *begin* matching the regular expression, and $e$, signalling the *end* of a match with the regular expression. The resulting structure of the compilation follows the pattern shown in the block diagrams of Figure 8.

The compilation scheme is a function of the form $compile_C^{b\to e}(E)$, with $b$ and $e$ being the begin and end channels:

$$\begin{array}{rcl}
compile_C & :: & \textsc{Chans} \times \textsc{Chans} \times \textsc{RegExp} \times \textsc{Mon} \rightarrow \textsc{Mon} \\
compile_C^{b\to e}((c,\overline{v})@k) & \triangleq & k[\![ *b?.read((c,\overline{v})@k, e!) ]\!]^{(k,1)} \\
compile_C^{b\to e}(E.F) & \triangleq & \mathsf{new}\, d.\Big(compile_C^{b\to d}(E) \parallel compile_C^{d\to e}(F)\Big) \\
compile_C^{b\to e}(E+F) & \triangleq & \mathsf{new}\, c,d.\Big(compile_C^{c\to e}(E) \parallel compile_C^{d\to e}(F) \parallel h[\![ *b?.(c! \parallel d!) ]\!]^{(h,1)}\Big) \\
compile_C^{b\to e}(E^*) & \triangleq & \mathsf{new}\, c,d.\Big(compile_C^{c\to d}(E) \parallel h[\![ (*d?.(c! \parallel e!)) \parallel (*b?.d!) ]\!]^{(h,1)}\Big)
\end{array}$$

For simplicity, all additional machinery used to synchronise the monitors is placed at an arbitrary location $h$, although this could be changed without affecting the proofs in the coming sections, given that this code does not involve any tracing. Also note, that due to the static nature of the monitoring approach, properties discovered at runtime and dynamic locations cannot be handled locally and thus the binding operator $(c, \exists\overline{x})@k.E$ is not supported.

The installation of a choreographed monitoring of expression $E$ can be expressed in terms of $compile_C^{b\to e}(E)$ by triggering the monitor right at the start, and failing upon a match:

$$monitor_C(E) \quad \triangleq \quad \mathsf{new}\, b,e.\Big(compile_C^{b\to e}(E) \parallel h[\![ b! \parallel *e?.\mathsf{fail} ]\!]^{(h,1)}\Big)$$

**Example 13.** *Consider the property:* $E \triangleq ((alarm, \langle\rangle)@k_1 + (alarm, \langle\rangle)@k_2).(pvt, \langle\rangle)@k_0$. *Compiling $E$ will result in three monitoring components located at $k_0$, $k_1$ and $k_2$ with* *additional communication handling processes which can be placed at any location h:*

$$
\begin{array}{ll}
& monitor_C(E) \\
= & \mathsf{new}\, b, e.\Big(compile_C^{b\to e}(E) \parallel h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)}\Big) \\
= & \mathsf{new}\, b, e.\Big(compile_C^{b\to e}(((alarm, \langle\rangle)@k_1 + (alarm, \langle\rangle)@k_2).(pvt, \langle\rangle)@k_0) \parallel h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)}\Big) \\
= & \mathsf{new}\, b, e. \\
& \quad \left( \begin{array}{l} \mathsf{new}\, d_0. \\ \quad compile_C^{b\to d_0}(((alarm, \langle\rangle)@k_1 + (alarm, \langle\rangle)@k_2)) \parallel \\ \quad compile_C^{d_0\to e}((pvt, \langle\rangle)@k_0) \end{array} \right) \parallel \\
& \quad h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)} \\
= & \mathsf{new}\, b, e. \\
& \quad \left( \begin{array}{l} \mathsf{new}\, d_0. \\ \quad \left( \begin{array}{l} \mathsf{new}\, d_1, d_2. \\ \quad compile_C^{d_1\to d_0}((alarm, \langle\rangle)@k_1) \parallel \\ \quad compile_C^{d_2\to d_0}((alarm, \langle\rangle)@k_2) \parallel \\ \quad h\{\![{*b?}.(d_1! \parallel d_2!)]\!\}^{(h,1)} \end{array} \right) \parallel \\ \quad compile_C^{d_0\to e}((pvt, \langle\rangle)@k_0) \end{array} \right) \parallel \\
& \quad h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)} \\
= & \mathsf{new}\, b, e. \\
& \quad \left( \begin{array}{l} \mathsf{new}\, d_0. \\ \quad \left( \begin{array}{l} \mathsf{new}\, d_1, d_2. \\ \quad k_1\{\![{*d_1?}.read((alarm, \langle\rangle)@k_1, d_0!)]\!\}^{(k_1,1)} \parallel \\ \quad k_2\{\![{*d_2?}.read((alarm, \langle\rangle)@k_2, d_0!)]\!\}^{(k_2,1)} \parallel \\ \quad h\{\![{*b?}.(d_1! \parallel d_2!)]\!\}^{(h,1)} \end{array} \right) \parallel \\ \quad k_0\{\![{*d_0?}.read((pvt, \langle\rangle)@k_0, e!)]\!\}^{(k_0,1)} \end{array} \right) \parallel \\
& \quad h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)}
\end{array}
$$

*Pushing the channel declarations to the top level and reorganising, the monitor is thus:*

$$
\begin{array}{l}
\mathsf{new}\, b, e, d_0, d_1, d_2. \\
\quad k_0\{\![{*d_0?}.read((pvt, \langle\rangle)@k_0, e!)]\!\}^{(k_0,1)} \parallel \\
\quad k_1\{\![{*d_1?}.read((alarm, \langle\rangle)@k_1, d_0!)]\!\}^{(k_1,1)} \parallel \\
\quad k_2\{\![{*d_2?}.read((alarm, \langle\rangle)@k_2, d_0!)]\!\}^{(k_2,1)} \parallel \\
\quad h\{\![{*b?}.(d_1! \parallel d_2!)]\!\}^{(h,1)} \parallel \\
\quad h\{\![b! \parallel {*e?}.\mathsf{fail}]\!\}^{(h,1)}
\end{array}
$$

Note that the compilation schemata given may lose completeness since they all perform synchronisation every time a value is to be read. For example, the compilation of $(c, 2)@k.(c@1)@k$ results in two synchronisations, which means that although location $k$ may have output 2 followed by 1 on channel $c$, by the time the second sychronisation takes place, the value 1 may have been missed by the monitor. There are however ways around this within the calculus itself; for instance, in the case of orchestration and migrating-monitor strategies, a more complete approach would be not to synchronise if the location remains unchanged. Both schemata would thus replace a synchronisation with location $k$ followed by a monitoring term $M$ (where *curr* is a free variable in $M$) with the following:

$$tmp?curr.\mathsf{if}\ k = curr\ \mathsf{then}\ M\ \mathsf{else}\ (\mathsf{sync}(k).\mathsf{new}\, tmp'.\Big(tmp'!k \parallel tmp'?curr.M)\Big)$$

The respective adjustments for the choreographed approach are slightly more involved and are outlined in [7]. The proofs in the next section can be adapted for these extensions.

## *4.3. Equivalence of Monitoring Strategies*

In the previous section we have seen how different monitors can be obtained from a regular expression. Formalising the monitors in terms of mDPI allows us to analyse them for correctness. One can prove their correctness in that a monitor synthesised from a regular expression matches exactly the strings covered by the regular expression. Another important question is that of relative correctness — are monitors produced via different synthesis techniques equivalent modulo locality of monitoring? We focus on the latter question, since it allows us to illustrate the use of mDPI bisimulation techniques to the utmost. In this section we thus prove this relative correctness result for the three monitoring strategies presented.

The main results, given in theorems 2 and 3, state that given any regular expression $E$, the derived orchestrated, migrating and choreographed monitors obtained for the expression (i.e. $monitor_M(E)$, $monitor_M(E)$ and $monitor_C(E)$) are bisimilar. The equivalence results follow by inductively (over the structure of the regular expression) proving how the compilation schemas $compile_O$, $compile_M$ and $compile_C^{b \to e}$ are related to each other.

We start by relating orchestrated and migrating monitoring approaches.

**Lemma 2.** *The orchestrated and migrating monitor compilation of an expression E are bisimilar, assuming bisimilar continuations M and M′. If* $\delta' \models k[\![M]\!]^{(k,i)} \approx h[\![M']\!]^{(k,i)}$*, then:*

$$\delta \models l[\![compile_M(E, M)]\!]^{(l,j)} \approx h[\![compile_O(E, M')]\!]^{(l,j)}$$

*Proof.* By induction on the structure of $E$, where each case is proven by showing co-inductively that the respective compilations are bisimilar. The resulting monitors for base case, $E = (c, \overline{v})@k$, are almost identical, apart from ($i$) an additional $\mathsf{go}\ k$ in case of the migrating monitor compilation, and ($ii$) differences in starting location: the additional migration move for ($i$) is a silent action which can be matched by an empty move by the orchestrated monitor, whereas the location discrepancies of (ii) are (purposefully) not reflected in the labels of the LTS.

The inductive cases follow easily from bisimulation equivalences obtained from the inductive hypotheses and Theorem 1 (Contextuality). For example, consider the compilations for $E^*$. We are hence required to prove that

$$\begin{aligned} \delta \ \models\ & l[\![\mathsf{new}\, s, f.(*s?.compile_M(E, f!) \parallel *f?.s!.M \parallel f!)]\!]^{(l,j)} \approx \\ & \mathcal{G}[\![\mathsf{new}\, s, f.(*s?.compile_O(E, f!) \parallel *f?.s!.M \parallel f!)]\!]^{(l,j)} \end{aligned}$$

It is trivially true that $\delta' \triangleright k[\![f!]\!]^{(k,i)}$ can act identically to $\delta' \triangleright \mathcal{G}[\![f!]\!]^{(k,i)}$. By the inductive hypothesis we hence infer that $\delta \models l[\![compile_M(E, f!)]\!]^{(l,j)} \approx \mathcal{G}[\![compile_O(E, f!)]\!]^{(l,j)}$.

It can also be shown that $\delta \models l[\![*f?.s!.M]\!]^{(l,j)} \approx \mathcal{G}[\![*f?.s!.M]\!]^{(l,j)}$, since $M$ and $M'$ are bisimilar and differences in monitor location are not reflected in the LTS. Hence, using the contextuality of $\approx$ shown in Theorem 1, we can infer the bisimilarity under $\delta$ of $\mathsf{new}\, s, f.(l[\![compile_M(E, f!)]\!]^{(l,j)} \parallel l[\![*f?.s!.M]\!]^{(l,j)} \parallel l[\![f!]\!]^{(l,j)})$ and $\mathsf{new}\, s, f.(\mathcal{G}[\![compile_O(E, f!)]\!]^{(l,j)} \parallel \mathcal{G}[\![*f?.s!.M]\!]^{(l,j)} \parallel \mathcal{G}[\![f!]\!]^{(l,j)})$, from which the desired result follows. The cases for $E_1 + E_2$ and $E_1.E_2$ are analogous. □

This theorem enables us to prove the equivalence of $monitor_M(E)$ and $monitor_O(E)$:

**Theorem 2.** *The migrating monitor and orchestrated compilations of a regular expression E are bisimilar:*

$$\delta \models monitor_M(E) \approx monitor_O(E)$$

The proof of Theorem 3 is more complex as static choreography relies on triggers. We therefore employ an intermediary modified version of a "triggered" orchestration to make Lemma 3 go through[9]. Proper orchestration can be then recovered using Lemma 4.

We now turn our attention to relating choreographed and orchestrated compilation.

**Lemma 3.** *For a continuation M, the choreographed compilation of a regular expression E without initial triggering is bisimilar to a "triggered" orchestrated compilation:*

$$\delta \models \mathsf{new}\, e.(compile_C^{b\to e}(E) \parallel h\{\!\!\{ *e?.M \}\!\!\}^{(h,1)}) \approx h\{\!\!\{ *b?.compile_O(E, M) \}\!\!\}^{(h,1)}$$

*Proof.* By induction on the structure of $E$, where each case is proved by showing co-inductively that the respective compilations are bisimilar. Consider the case for catenation $E.F$. The orchestrated compilation is $(\delta \rhd h\{\!\!\{ *b?.compile_O(E.F, M) \}\!\!\}^{(h,1)})$, which can be expanded to $(\delta \rhd h\{\!\!\{ *b?.compile_O(E, (compile_O(F, M))) \}\!\!\}^{(h,1)})$. By I.H. on $E$ this acts identically to $(\delta \rhd \mathsf{new}\, d.(compile_C^{b\to d}(E) \parallel h\{\!\!\{ *d?.compile_O(F, M) \}\!\!\}^{(h,1)}))$. By I.H. on $F$ and contextuality, this acts bisimilarly to $(\delta \rhd \mathsf{new}\, d.(compile_C^{b\to d}(E) \parallel \mathsf{new}\, e.(compile_C^{d\to e}(F) \parallel h\{\!\!\{ *e?.M \}\!\!\}^{(h,1)})))$ which, up to standard structural manipulation of terms, $\equiv$, (see [13]) is equivalent to $(\delta \rhd \mathsf{new}\, e.(compile_C^{b\to e}(E.F) \parallel h\{\!\!\{ *e?.M \}\!\!\}^{(h,1)}))$. Hence, for local clocks $\delta$, we can show that

$$\delta \models \mathsf{new}\, e.(compile_C^{b\to e}(E) \parallel h\{\!\!\{ *e?.M \}\!\!\}^{(h,1)}) \approx h\{\!\!\{ *b?.compile_O(E, M) \}\!\!\}^{(h,1)}$$

□

This result allows us to prove, together with co-induction, that triggering the monitor once still gives the same result:

**Lemma 4.** *The standard orchestrated compilation for E is bisimilar to a scoped, "singly-triggered" orchestrated compilation :*

$$\delta \models h\{\!\!\{ compile_O(E, M) \}\!\!\}^{(h,1)} \approx \mathsf{new}\, b.(h\{\!\!\{ b! \}\!\!\}^{(h,1)} \parallel h\{\!\!\{ *b?.compile_O(E, M) \}\!\!\}^{(h,1)})$$

Finally, this result can be used to prove equivalence between choreographed and orchestrated monitors.

**Theorem 3.** *The choreographed and orchestrated compilations of a regular expression E are bisimilar:*

$$\delta \models monitor_C(E) \approx monitor_O(E)$$

[9]Trigger channels $b, e, \ldots$ used by compilations are always assumed to be fresh.

*Proof.* By Lemma 3 and contextuality (using the context $\mathsf{new}\, b.(h\{\![b!]\!\}^{(h,1)} \parallel -)$) we have

$$\delta \models monitor_C(E) \approx \mathsf{new}\, b.(h\{\![b!]\!\}^{(h,1)} \parallel h\{\![*b?.compile_O(E, \mathsf{fail})]\!\}^{(h,1)})$$

and by Lemma 4 we have

$$\delta \models \mathsf{new}\, b.(h\{\![b!]\!\}^{(h,1)} \parallel h\{\![*b?.compile_O(E, \mathsf{fail})]\!\}^{(h,1)} \approx monitor_O(E)$$

The required result then follows by transitivity of $\approx$. □

Theorems 2 and 3, together with transitivity of the bimulation relation allow us to conclude that all three forms of monitoring are, in fact, pairwise equivalent.

## *4.4. Migrating Monitoring Preserves Locality*

Although the monitoring approaches exhibit the same behaviour, there are reasons for choosing one over another — primarily due to how the monitors are distributed across locations. In the case of choreographed and agent migration approaches, this should ensure that the monitors listen to channels locally.

A monitor's intention to read a trace remotely can be inferred when it is syntactically of the form $\delta \rhd k\{\![\mathbf{q}(c, \overline{x}).M']\!\}^{(l,i)}$, *i.e.* when its current location, $k$, and that of monitoring context, $l$, do not match. We can thus inductively define the following predicate over systems signalling remote trace violations through these rules.

$$\frac{}{k\{\![\mathbf{q}(c,\overline{x}).M']\!\}^{(l,i)} \rightarrow_{err}}\; k \neq l \qquad \frac{S \rightarrow_{err}}{\mathsf{new}\, c.S \rightarrow_{err}} \qquad \frac{S \rightarrow_{err}}{S \parallel U \rightarrow_{err}} \qquad \frac{S \rightarrow_{err}}{U \parallel S \rightarrow_{err}}$$

**Definition 3.** *A system $S$ is said to be* local*, written $local(S)$, if for any counter values $\delta$, err is not reachable from $S$:*
$local(S) \triangleq \forall \delta \neg\Big(\exists \alpha_1, \ldots, \alpha_n, \delta', S'$ *such that* $\delta \rhd S \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} \delta' \rhd S'$ *where* $S' \rightarrow_{err}\Big)$

**Proposition 1.** (*i*) *Any system $S$ which does not contain a sub-term of the form $\mathbf{q}(c, \overline{x}).M'$ is local* i.e. *$local(S)$.* (*ii*) *Locality is, in some sense, contextual: $local(S)$ and $local(U)$ implies $local(S \parallel U)$ and $local(\mathsf{new}\, c.S)$.*

It is easy to show that orchestrated monitoring does not always preserve locality. We can also prove that migrating and choreographed monitoring always occurs locally.

**Theorem 4.** *There are regular expressions $E$ for which $\neg(local(monitor_O(E)))$.*

*Proof.* By counter example. Consider the regular expression $(c, \overline{v})@k$, with $k \neq h$ where $h$ is the global orchestration location. One can easily verify that the respective compilation can transition into the following non-local system:

$$\mathsf{new}\, d.\Big(\; h\{\![\mathbf{q}(c,x).\mathsf{if}\ \overline{x} = \overline{v}\ \mathsf{then}\ (\mathsf{fail} \parallel d!)\ \mathsf{else}\ d!]\!\}^{(k,\delta(k))} \parallel h\{\![*d?.(\ldots)]\!\}^{(k,\delta(k))}\;\Big) \rightarrow_{err}$$

□

As before, we prove that locality is preserved for the compilation schemata $\mathit{compile}_M$ and $\mathit{compile}_C$, from which we can then conclude locality preservation by the monitors.

**Lemma 5.** *Compiling regular expression $E$ into a migrating monitor with a local continuation $M$ yields a local monitor: $\mathit{local}(h\{\![M]\!\}^{(l,i)})$ implies $\mathit{local}(\mathit{compile}_M(E, M))$.*

*Proof.* By induction on the structure of $E$. The base case, $(c, \overline{v})@k$, yields the compiled monitor

$$h\{\![\mathsf{go}\ k.\mathsf{sync}(k).\mathsf{new}\,d.(s! \parallel *s?.(\mathbf{q}(c, x).\mathsf{if}\ \overline{x} = \overline{v}\ \mathsf{then}\ (M \parallel s!)\ \mathsf{else}\ s!))]\!\}^{(l,i)}$$

which never produces an error since all queries are preceded by a single synchronising operation $\mathsf{sync}(k)$, which is, in turn, preceded by a migration to that location, $\mathsf{go}\ k$.

For the inductive case $E + F$, we need to show that $\mathit{err}$ is not reachable from $k\{\![\mathsf{new}\,d.(\mathit{compile}_M(E, d!) \parallel \mathit{compile}_M(F, d!) \parallel *d?.M)]\!\}^{(l,j)}$. From Proposition 1(i) we know that $\mathit{local}(k\{\![d!]\!\}^{(l,j)})$ and $\mathit{local}(k\{\![*d?.M]\!\}^{(l,j)})$ and by I.H. on $E$ and $F$ we conclude $\mathit{local}(k\{\![\mathit{compile}_M(E, d!)]\!\}^{(l,j)})$ and $\mathit{local}(k\{\![\mathit{compile}_M(F, d!)]\!\}^{(l,j)})$. The result follows from Proposition 1(ii). The other inductive cases are analogous. □

Similarly, for choreographed monitoring locality is preserved:

**Lemma 6.** *Compiling a regular expression $E$ into a choreographed monitor results in a local monitor: $\mathit{local}(\mathit{compile}_C^{b\to e}(E))$.*

*Proof.* By induction on the structure of $E$. The base case, $(c, v)@k$ yields the monitor

$$k\{\![*b?.\mathit{read}((c, \overline{v})@k, e!)]\!\}^{(k,1)}$$

which can be easily shown not to result in $\mathit{err}$.

For the inductive case of $E + F$, the compilation yields $\mathsf{new}\,c, d.(\mathit{compile}_C^{c\to e}(E) \parallel \mathit{compile}_C^{d\to e}(F) \parallel h\{\![*b?.(c! \parallel d!)]\!\}^{(h,1)})$. One can show that this system is local by using similar reasoning to that used for the respective inductive case of Lemma 5. The other inductive cases are analogous. □

These two lemmata allow us to conclude locality of migrating and choreography-based monitoring.

**Theorem 5.** *Both migrating monitors and choreography-based monitoring guarantee locality of monitoring: for any regular expression $E$, both $\mathit{local}(\mathit{monitor}_M(E))$ and $\mathit{local}(\mathit{monitor}_C(E))$ hold.*

*Proof.* We conclude that the migrating monitor compilation is local from Lemma 5 and the fact that $\mathit{local}(k\{\![\mathsf{fail}]\!\}^{(k,j)})$. We conclude that the choreographed compilation is local from Lemma 6, Proposition 1 (i) (for the outer plumbing code) and then Proposition 1(ii). □

## 5. Discussion and Related Work

This paper formalises the distributed monitoring scenario, allowing for a comparison of instrumentation strategies. This approach is somewhat different than other frameworks presented in the literature, which typically focus on formalising and implementing one particular strategy for a particular scenario.

Much work appearing in the field of runtime verification [19, 20] focusses on the verification of techniques for the analysis of traces of events generated by the system. The distinguishing features of the subset of work in the area focussing on distributed systems are that (i) in a distributed setting, different locations generate separate traces, and given that one usually lacks a global clock one has to make do with a limited notion of consequentiality [16]; and (ii) beyond the question of how to instrument monitoring code, one is faced with the question of where to instrument the code, since different locations give rise to different communication behaviour. Since the contribution of this work lies in these features, in this section we focus on related work on runtime verification for distributed systems.

To the best of our knowledge, no other generic formal framework for reasoning about issues such as monitor correctness and instrumentation strategies in a distributed system setting appears in the literature. Even though there are numerous process calculi that address locations and distribution — the closest to our work being [21, 13] — none of these model trace generation and monitoring as part of the computation; rather, traces are often a meta-construct aiding system analysis. The work by Zavattaro *et al.* [22, 23] studies more expressive contract languages for Service Oriented Computing. Their aims differ from ours in that they are concerned with the analysis of contract mechanisms such as error handling and compensations with respect to notions of correctness such as service compliance. In contrast, our work focusses on the instrumentation and monitoring, using contracts as a vehicle for explaining the issues that arise in distributed settings.

DiAna [2] is one of the more formal attempts based on the actor model [24], adopting *knowledge vectors* (which extend vector clocks [14]) for monitoring causal properties of distributed systems. More specifically, this approach recognises the impracticality of monitoring distributed systems in a centralised fashion due to unreasonable bandwidth overheads. To this effect, the framework adopts a pre-compiled monitor per location, imposing a (reasonable) overhead on each across-border interaction, which is exploited in order to share knowledge vector instances between monitors. Using a simple update strategy, the monitor on the receiving end is guaranteed to obtain the latest known expression evaluations, extracting a causal order on remote events in the process. By verifying causal assertions — a well-understood subset of temporal properties — DiAna offers a more elegant solution towards the statically choreographed monitoring of distributed architectures. However, this approach cannot handle dynamic architectures, since knowledge vectors are based on the assumption that contributing nodes are known at compile time. It is for this reason that mDPi extracts temporal orderings across locations by exploiting monitor execution, as opposed to underlying system interactions as is the case with DiAna. One should also note that this latter framework specifies causal properties through PT-DTL, a distributed extension to PT-LTL; a logic with a proven track record in a runtime verification setting [25, 26, 20]. This points to the need for the study of more expressive logics within the setting of mDPi, and is left as future work. Finally, it is worth noting that DiAna ignores the issue of data exposure by sharing knowledge vectors across locations.

Two other tool-oriented frameworks include DMaC [6] and GEM [5]. The former, an extension of the MaC [8] framework (originally developed for monolithic systems), makes use of declarative networking techniques for verifying system behaviour against formalised requirements. In order to do so, PEDL scripts (a language for defining primitive events within the system) are exploited in order to convert MEDL (similar to past-time Linear Temporal Logic) specifications to declarative queries, run over distributed tables, and is hence choreography based. Although DMaC offers an implementation-independent approach, it fails to offer solutions to the problem of asynchrony across locations. Moreover, the approach depends on cost-based optimisations for query placement which remain unchanged during execution, a problem which is known to be NP-hard — although various dynamic programming techniques and heuristics are used, there is a risk of generating inefficient query plans resulting in unreasonable bandwidth overheads. Issues of dynamic architectures and information confidentiality are not addressed. GEM also deals with distributed monitoring, adopting an interpreted rule-based language for monitoring temporal properties across locations. Crucially, it assumes the availability of a global clock, relieving itself of substantial difficulties inherent with the monitoring of distributed architectures. The framework offers basic solutions to the loss of event orderings across locations, including the specification of a *tolerated* limit on event delays per rule, keeping an event history in the meanwhile (beyond which subsequent events are ignored). This is achieved by delaying the triggering of rules for a specified amount of time, with the rule evaluated on the collected event history. Clearly, these solutions are applicable due to the assumed synchrony across locations. Although GEM allows for the loading of new rules at runtime (being an interpreted language), it does not consider dynamic architectures. Moreover, local pertinent events are disseminated to remote nodes, exposing information.

## 6. Conclusions

In this paper we presented mDPi, a location-aware calculus with explicit monitoring capabilities, whilst internalising the local tracing of process behaviour. This calculus' purpose is the formalisation of the distributed monitoring scenario, allowing for the comparison of competing strategies. Apart from presenting mDPi's syntax and LTS-based semantics, we justify our approach in a number of ways:

1. We justify a bisimulation-based semantics for mDPi through the concept of compositionality, Theorem 1;
2. We provide orchestrated, choreographed and migrating monitor compilations for a simple regular-expression based language, Section 4.1;
3. The three compilations are proved to be equivalent up to monitoring location, in Theorem 2 and Theorem 3;
4. We prove that, with respect to this logic, migrating monitors and static-choreography minimise information leaks by ensuring local monitoring, whereas orchestration does not, Theorem 4 and Theorem 5;

From the point of view of the migrating monitor strategy, these results formally justify it as equally expressive to existing distributed monitoring strategies, while giving additional guarantees regarding potential information leakage through remote monitoring. Needless to say, migrating monitors still does not solve the problem of information leaks — for instance, the migration pattern of monitors and the content of the monitor continuations could still be used to deduce information about the local events. While the latter aspect can be addressed using standard encryption techniques[10], the former aspect poses an interesting challenge that can be tackled in future work.

We also plan to extend mDPI to address issues such as clock boundaries and real-time operators, which will allow us to devise and study monitoring strategies that are more complete. At present, the calculus also ensures that monitoring is non-intrusive, in that it reads events from the system but does not otherwise interact with it. To be able to handle reparations triggered upon contract violation, and to express correctness through monitor-oriented programming [27], this constraint needs to be relaxed; this is another direction we would like to explore in the future.

## References

[1] F. Barbon, P. Traverso, M. Pistore, M. Trainotti, Run-time monitoring of instances and classes of web service compositions, in: ICWS '06: Proceedings of the IEEE International Conference on Web Services, IEEE Computer Society, Washington, DC, USA, 2006, pp. 63–71.

[2] K. Sen, A. Vardhan, G. Agha, G. Roşu, Efficient decentralized monitoring of safety in distributed systems, International Conference on Software Engineering (2004) 418–427.

[3] T. S.Cook, D. Drusinksy, M.-T. Shing, Specification, validation and run-time moniroting of soa based system-of systems temporal behaviors, in: System of Systems Engineering (SoSE), IEEE Computer Society, 2007.

[4] I. H. Krüger, M. Meisinger, M. Menarini, Interaction-based runtime verification for systems of systems integration, Computer Science and Engineering Department, University of California, San Diego USA.

[5] M. Mansouri-Samani, M. Sloman, Gem: a generalized event monitoring language for distributed systems, Distributed Systems Engineering 4 (2) (1997) 96–108.

[6] W. Zhou, O. Sokolsky, B. T. Loo, I. Lee, Dmac: Distributed monitoring and checking., in: Runtime Verification 09, Vol. 5779 of LNCS, Springer, 2009, pp. 184–201.

[10]For example, the continuations can be encrypted with the public key of the next listener, and signed for authenticity.

[7] A. Francalanza, A. Gauci, G. J. Pace, Distributed system contract monitoring, in: Fifth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11), Vol. 68 of Electronic Proceedings in Theoretical Computer Science (EPTCS), 2011.

[8] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Java-mac: a run-time assurance tool for java programs, in: In Runtime Verification 2001, volume 55 of ENTCS, Elsevier Science Publishers.

[9] D. Chappell, Enterprise Service Bus, O'Reilly Media, 2004.

[10] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Softw. Eng. 3 (1977) 125–143.

[11] D. E. Denning, An intrusion-detection model, IEEE Transactions on Software Engineering 13 (1987) 222–232.

[12] C. Abela, A. Calafato, G. J. Pace, Extending wise with contract management, in: WICT 2010.

[13] M. Hennessy, A Distributed Pi-Calculus, Cambridge University Press, New York, NY, USA, 2007.

[14] C. Fidge, Timestamps in message-passing systems that preserve the partial ordering, in: Proceedings of the 11th Australian Computer Science Conference, Vol. 10, 1988, pp. 56–66.

[15] D. Sangiorgi, D. Walker, $\pi$-Calculus: A Theory of Mobile Processes, Cambridge University Press, New York, NY, USA, 2001.

[16] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565. doi:http://doi.acm.org/10.1145/359545.359563.

[17] U. Sammapun, O. Sokolsky, Regular expressions for run-time verification, in: Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis (ATVA'03), Taipei, Taiwan, 2003.

[18] K. Claessen, G. J. Pace, An embedded language framework for hardware compilation, in: Designing Correct Circuits '02, Grenoble, France, 2002.

[19] S. Colin, L. Mariani, Run-time verification, in: Model-Based Testing of Reactive Systems, Springer, 2004, pp. 525–555.

[20] M. Leucker, C. Schallhart, A brief account of runtime verification, Journal of Logic and Algebraic Programming 78 (5) (2009) 293–303.
URL `http://dx.doi.org/10.1016/j.jlap.2008.08.004`

[21] M. Berger, K. Honda, The two-phase commitment protocol in an extended pi-calculus, in: L. Aceto, B. Victor (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 39, Elsevier, 2003, proc. of EXPRESS'00.

[22] M. Bravetti, G. Zavattaro, A theory of contracts for strong service compliance, Mathematical. Structures in Comp. Sci. 19 (3) (2009) 601–638.

[23] C. Guidi, I. Lanese, F. Montesi, G. Zavattaro, On the interplay between fault handling and request-response service invocations, in: ACSD, IEEE, 2008, pp. 190–198.

[24] G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, Cambridge, MA, USA, 1986.

[25] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, Springer, 2004, pp. 44–57.

[26] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, O. Sokolsky, Formally specified monitoring of temporal properties., in: ECRTS, IEEE Computer Society, 1999, pp. 114–122.

[27] F. Chen, G. Roşu, Towards monitoring-oriented programming: A paradigm combining specification and implementation, in: Runtime Verification (RV'03), Vol. 89(2) of ENTCS, 2003, pp. 108 – 127.

## Appendix A. Bisimulation Proof for Example 9

In example 9, we looked at monitors $M^{\text{orch}}$ and $M^{\text{chor}}$ (as defined in Example 3) when monitoring a system Sys (as defined in Example 1). When subject to the local clocks $\delta = \{l \mapsto j, k \mapsto i\}$ we wanted to show their bisimilarity:

$$\delta \models (\text{Sys} \parallel M^{\text{orch}}) \approx (\text{Sys} \parallel M^{\text{chor}})$$

By Theorem 1, in order to show this it suffices to show:

$$\delta \models M^{\text{orch}} \approx M^{\text{chor}}$$

This can in turn be proved by constructing the witness family of relations over systems indexed by clocks below (we here denote them as a single relation over triples $\textsc{Clocks} \times \textsc{Sys} \times \textsc{Sys}$). This relation assumes the following definitions:

$$\begin{aligned}
M^o &\triangleq \mathbf{q}(d,x).\mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\\
M^o_1 &\triangleq \mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\\
M^o_2 &\triangleq \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}\\
M^o_3 &\triangleq \mathsf{if}\ y < 2\ \mathsf{then\ fail}\\
M^k &\triangleq \mathbf{q}(d,x).\ b!x\\
M^l &\triangleq b?x.\ \mathsf{sync}(l).\ \mathbf{q}(x,y).\ \mathsf{if}\ y < 2\ \mathsf{then\ fail}
\end{aligned}$$

The witness relation shown in Figure A.9 is made up of six groups of triples, and assumes the following ordering amongst local clocks:

$$\delta' \leq \delta \ \triangleq\ dom(\delta') = dom(\delta) \wedge (k \in dom(\delta) \Rightarrow \delta'(k) \leq \delta(k))$$

The first group of triples describe the unsuccessful queries of trace records by the respective monitors through actions $\mathbf{inT}(c_g, v_g, k, g)$ derived using rule Skp, preceded by a series of $\tau$-transitions increasing the local counters through rule Cntr; every unsuccessful query increases the trace monitor index of the respective monitor and introduces the respective trace record queried as part of the system.

The second group of triples describe the successful querying of a trace record at index $(k, j')$ and are transitioned to from triples of the first group through $\mathbf{inT}(c_g, v_g, k, g)$ actions derived using rule InT. The third group relates the communication on the scoped channel $b$ from the choreographed monitor with a no-transition from the orchestrated monitor side. Again, the fourth group describes unsuccessful trace record queries at location $l$, which then transition to the fifth group of triples once a successful query is made. Finally, the sixth group of triples describe the possible branches that the monitor condition may take, based on the value inputted from the trace record queried.

$$
\left\{
\begin{array}{l|l}
\left\langle \begin{array}{l} \delta',\ m\llbracket M^{o}\rrbracket^{(k,i'+1)} \parallel T_k, \\ \qquad \mathsf{new}\, b.(k\llbracket M^{k}\rrbracket^{(k,i'+1)} \parallel l\llbracket M^{l}\rrbracket^{(l,h)}) \parallel T_k \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ T_k \triangleq \prod_{g=i}^{i'} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \\ \forall g\ s.t.\ i \le g \le i'.\ c_g \ne d \end{array} \\[3em]
\left\langle \begin{array}{l} \delta',\ m\llbracket M_1^{o}\{v/x\}\rrbracket^{(k,i'+1)} \parallel T \parallel T_k, \\ \qquad \mathsf{new}\, b.(k\llbracket b!v\rrbracket^{(k,i'+1)} \parallel l\llbracket M^{l}\rrbracket^{(l,h)}) \parallel T \parallel T_k \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ T_k \triangleq \prod_{g=i}^{i'-1} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \\ \forall g\ s.t.\ i \le g < i'.\ c_g \ne d \\ T \triangleq k\llbracket \mathbf{t}(d, v, i')\rrbracket \end{array} \\[3em]
\left\langle \begin{array}{l} \delta',\ m\llbracket M_1^{o}\{v/x\}\rrbracket^{(k,i'+1)} \parallel T_k, \\ \qquad\qquad l\llbracket M_1^{o}\{v/x\}\rrbracket^{(l,h)} \parallel T_k \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ T_k \triangleq \prod_{g=i}^{i'} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \end{array} \\[3em]
\left\langle \begin{array}{l} \delta',\ m\llbracket M_2^{o}\{v/x\}\rrbracket^{(l,j'+1)} \parallel T_k \parallel T_l, \\ \qquad\quad l\llbracket M_2^{o}\{v/x\}\rrbracket^{(l,j'+1)} \parallel T_k \parallel T_l \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ j'' \le j' < \delta'(l), \\ T_k \triangleq \prod_{g=i}^{i'} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \\ T_l \triangleq \prod_{f=j''}^{j'} l\llbracket \mathbf{t}(d_f, v_f, f)\rrbracket \\ \forall f\ s.t.\ j'' \le f \le j'.\ d_f \ne v \end{array} \\[3em]
\left\langle \begin{array}{l} \delta',\ m\llbracket M_3^{o}\{v/x\}\{w/y\}\rrbracket^{(l,j'+1)} \parallel T_k \parallel T \parallel T_l, \\ \qquad\quad l\llbracket M_3^{o}\{v/x\}\{w/y\}\rrbracket^{(l,j'+1)} \parallel T_k \parallel T \parallel T_l \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ j'' \le j' < \delta'(l), \\ T_k \triangleq \prod_{g=i}^{i'} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \\ T_l \triangleq \prod_{f=j''}^{j'-1} l\llbracket \mathbf{t}(d_f, v_f, f)\rrbracket \\ \forall f\ s.t.\ j'' \le f < j'.\ d_f \ne v \\ T \triangleq k\llbracket \mathbf{t}(v, w, j')\rrbracket \end{array} \\[3em]
\left\langle \begin{array}{l} \delta',\ m\llbracket M\rrbracket^{(l,j'+1)} \parallel T_k \parallel T_l, \\ \qquad\quad l\llbracket M\rrbracket^{(l,j'+1)} \parallel T_k \parallel T_l \end{array} \right\rangle
& \begin{array}{l} \delta' > \delta,\ i' < \delta'(k), \\ j'' \le j' < \delta'(l), \\ T_k \triangleq \prod_{g=i}^{i'} k\llbracket \mathbf{t}(c_g, v_g, g)\rrbracket \\ T_l \triangleq \prod_{f=j''}^{j'} l\llbracket \mathbf{t}(d_f, v_f, f)\rrbracket \\ M = \mathsf{fail} \text{ or } M = \mathsf{stop} \end{array}
\end{array}
\right\}
$$

Figure A.9: Bisimulation triples