Distributed System Contract Monitoring

Adrian Francalanza Andrew Gauci Gordon J. Pace
Department of Computer Science, University of Malta *
{adrian.francalanza|agau0006|gordon.pace}@um.edu.mt

The use of behavioural contracts, to specify, regulate and verify systems, is particularly relevant to runtime monitoring of distributed systems. System distribution poses major challenges to contract monitoring, from monitoring-induced information leaks to computation load balancing, communication overheads and fault-tolerance. We present mDPı, a location-aware process calculus, for reasoning about monitoring of distributed systems. We define a family of Labelled Transition Systems for this calculus, which allow formal reasoning about different monitoring strategies at different levels of abstractions. We also illustrate the expressivity of the calculus by showing how contracts in a simple contract language can be synthesised into different mDPı monitors.

1 Introduction

As systems continue to grow in size and complexity, the use of behavioural contracts is becoming crucial in specifying, regulating and verifying correctness. Various notions of contracts have been used, but most prevailing variants enable the regulation of the behaviour of a system, possibly with consequences in case of violations. Such contracts can then be used in multiple ways, from system validation and verification, to conflict analysis of the contract itself. One important use of contracts is in runtime monitoring: system traces are analysed at runtime to ensure that any contract violating behaviour is truncated before it leads to any further consequences, possibly applying reparations to recover from anomalous states.

More and more systems are deployed in a distributed fashion, whether out of our choice or necessity. Distribution poses major design challenges for runtime monitoring of contracts, since monitors themselves can be distributed, and trace analysis can be carried out remotely across location. This impacts directly various aspects of the system being monitored, from the security of sensitive information, to resource management and load balancing, to aspects relating to fault tolerance. Various alternative solutions have been presented in the literature, from fully orchestrated solutions where monitors are located at a central location, to statically distributed monitors where the contract monitor is statically decomposed into different components hosted at the location where system traces are generated.

The primary contribution of this paper is a unified formal framework for studying different monitoring strategies. We present a location-aware calculus supporting explicit monitoring as a first class entity, and internalising behavioural traces at the operational level rather than at a meta-level. We show the expressivity of the calculus by using it to model different distributed system monitoring strategies from the literature. We also present a novel architecture in which contract monitors *migrate* across locations to keep information monitoring local, while limiting remote monitor instrumentation in certain situations. The versatility of the contract-supporting calculus is later illustrated by showing how it can model different instrumentation strategies. In particular, we show how behavioral contracts expressed using regular expressions can be automatically translated into monitors of different monitoring strategies.

*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.

Submitted to:
FLACOS 2011

The paper is organised as follows. In section 2, we outline the contract monitoring strategies for distributed systems from the literature. We present a monitoring distributed calculus in sections 3 and 4, and illustrate its use for monitoring behavioural contracts expressed as regular expressions in section 5. We discuss related work and conclude in section 6.

2 Monitoring Distributed Systems

Monitoring distributed systems is distinct from monolithic monitoring. These systems are usually characterised by the absence of a global clock when ordering events across location boundaries. They often consist of autonomous, concurrently executing subsystems communicating through message passing, each with its local memory, where communication across subsystems is considerably slower than local communication. 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 [4] are instances of such systems.

These characteristics impinge on contract monitoring. For instance, the absence of a global clock prohibits precise monitoring for consequentiality contracts across locations [11]. Distribution impacts on information locality; subsystem events may contain confidential information which must not be exposed globally thereby requiring local monitoring. The possibility of distributing monitors also introduces concerns for monitor load balancing and monitor replication for fault tolerance.

Whether monitored contracts are known at compile time or else become known at runtime also affects distributed monitoring. Static contracts, ones which are fully known at compile time, are typically not expressive enough for distributed systems with dynamic topologies. Dynamic contracts, ones which are partially known at compile time, tend to be more appropriate for such systems. They are found in intrusion detection [8], 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 upon discovery.

2.1 Classifying Distributed System Monitoring Approaches

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 typically distribute monitoring across the subsystems. Orchestration, used traditionally in monolithic systems, is relatively the simplest strategy and its centralisation facilitates the handling of dynamic contracts. The approach is however susceptible to data exposure when contacts concern private information; it also leads to considerable communication overhead across locations, and poses a security risk by exposing the monitor as a central point of attack. By contrast, choreography-based approaches push verification locally, potentially minimising data exposure and communication overhead. Communication between localised monitors is typically substantially less than that induced by the remote monitoring of a central monitor. Choreography is however more complex to instrument, as contracts need to be decomposed into coordinating local monitors, is more intrusive, burdening monitored subsystems with additional local computation, and is applicable only when the subsystems allow local instrumentation of monitoring code. Choreographed monitors are also instrumented upfront, which may lead to redundant local instrumentation in the case of consequential contracts; 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.

Static orchestration verifying pre-determined contracts is a common approach, *e.g.,* [3], where web-service compositions are monitored in an orchestrated fashion. By contrast, [1] uses orchestration to monitor for dynamic properties: web services are centrally monitored against BPMN workflow specifications, facilitating the verification of contracts (representing system properties) discovered at runtime *on-the-fly*. Extensive work has also been done in static choreography monitoring [13, 7, 10, 12, 14], where communication overhead is mitigated by breaking up contracts into parts which can be monitored independently and locally, synchronising between the monitors only when necessary. However, to the best of our knowledge, these approaches cannot fully handle dynamic contracts with runtime contract decomposition and distribution, nor do they tackle monitor-induced data exposure.

An alternative approach is that of using *migrating monitors*, which adequately supports dynamic contracts whilst still avoiding orchestration; in particular, it limits instrumentation of distributed monitors in cases were monitoring is dependent on computation. Using this approach, monitors reside where the immediate confidential traces reside, and migrate to other subsystems, possibly discovered at runtime, when information from elsewhere is required *i.e.,* on a *by-need* basis. This enhanced expressivity also permits support for dynamic topologies and contracts learnt at runtime.

***Example* 2.1.** *Consider the hospital system contract:*

> *A nurse will have access to a patient's records after requesting them, as long as his or her request is approved by a doctor assigned to the patient.*

We assume that (i) the nurse requests (and eventually accesses) the patient's data from a handheld device, (ii) the information about which doctors have been assigned to which patients resides at the central site, and (iii) the patient's information is stored on the doctors' private clinic systems, where doctors can also allow nurses permission to access patients' data.

A migrating monitor starts on the nurse's system; upon receiving a patient-information request, it migrates to the hospital system, decomposes, and spreads to the systems of the patient's assigned doctors to check for permissions allowing the nurse access to the records. Finally, if the permission is given, the decomposed monitors migrate back to the nurse's device to check that the records are available. As with choreographed monitoring, and in contrast to orchestration, migrating monitors can ensure that monitoring is performed locally. The main difference is that instrumentation of monitors can be performed at runtime. For instance, when monitoring the hospital contract clause, no monitor is installed on a doctor's system unless a nurse has made a request for information about a patient assigned to that doctor, which is less intrusive.

The added expressivity 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 procedure, to proof-carrying monitors which come with a proof guaranteeing what resources they access. This issue will not be discussed further here, but is crucial for the practicality of migrating monitors.

There are a number of issues relating to these different monitoring approaches that are unresolved. For instance, it is somewhat unclear, at least from a formal perspective, what added benefits migration brings to distributed monitoring. There are also issues relating to the monitoring of consequential properties across locations, which cannot be both sound and complete: in Example 2.1, by the time the monitor migrates to the doctor's system, the doctor may have already approved the nurse's request. Distribution precludes precise analysis of the relative timing of the traces, and one has the option of taking a worst or best case scenario, avoiding false positives or false negatives respectively. This problem is also prevalent in both orchestrated and choreographed approaches. We therefore require a *common formal framework* where all three approaches can be expressed. This would, in turn, permit rigorous analysis and evaluation with respect to these issues.

3 A Distributed Monitoring Language

We present mDPI, an adaptation of the distributed π-calculus in [9], where processes are partitioned across a flat organisation of locations; their behaviour is amenable to monitoring through traces, administered at a local level. The syntax, presented in Figure 1, assumes denumerable sets of channel names $c, d \in \text{CHANS}$, location names $l, k, h \in \text{LOCS}$, indices $n, m, o \in \text{IDX}$ and variables $x, y \in \text{VARS}$; identifiers u, v range over $\text{IDENTS} = \text{CHANS} \cup \text{LOCS} \cup \text{IDX} \cup \text{VARS}$, with identifier list $v_1, \ldots, v_n$ denoted as $\tilde{v}$.

$$
\begin{array}{rcl}
S, R \in \text{SYS} & ::= & k[\![P]\!] \mid S \parallel R \mid \mathsf{new}\, c.S \\
P, Q \in \text{PROC} & ::= & \mathsf{stop} \mid u!v.P \mid u?x.P \mid \mathsf{new}\, c.P \mid \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P \| Q \mid *P \mid \{M\}^{(l,n)} \mid T \\
T \in \text{TRC} & ::= & \mathbf{t}(c, \tilde{d}, n) \\
M, N \in \text{MON} & ::= & \mathsf{stop} \mid u!v.M \mid u?x.M \mid \mathsf{new}\, c.M \mid \mathsf{if}\ u = v\ \mathsf{then}\ M\ \mathsf{else}\ N \mid M \| N \mid *M \\
 & \mid & \mathbf{q}(c, \tilde{x}).M \mid \mathsf{sync}(u).M \mid \mathsf{getl}(x, y).M \mid \mathsf{setl}(u, v).M \mid \mathsf{go}\ u.\ M \mid \mathsf{ok} \mid \mathsf{fail}
\end{array}
$$

Figure 1: mDPI Syntax

Systems, S, R, are made up of *located processes* $k[\![P]\!]$ (the tag k denotes the current location hosting P) which can be composed in parallel and subject to scoping of channel names.

Located entities are partitioned into three syntactic categories: *Processes*, PROC, comprise the standard communication constructs for output, c!$\tilde{d}$.P, and input, c?$\tilde{x}$.P (variables $\tilde{x}$ are bound in the continuation P), together with the name-matching conditional, replication, parallel composition and name restriction; *Traces*, are made up of individual trace entities, $\mathbf{t}(c, \tilde{d}, n) \in \text{TRC}$ recording communication of values $\tilde{d}$ on channel c at timestamp n - they are meant to be ordered as a complete log recording past computation at a particular location; *Monitors*, MON, are similar in structure to processes, and are delimited at the process level by enclosing brackets $\{M\}^{(k,n)}$, where (k, n) denotes the *monitoring context i.e., the location and log position of the trace being monitored.* In addition, they can:

- query traces for records of communication on channel c, $\mathbf{q}$(c,$\tilde{x}$). M, where the index and location of the trace monitored is inferred from the enclosing monitoring context, (k, n).
- get the information relating to the current monitoring context, $\mathsf{getl}(x, y).M$ (x, y bound in M), and set the monitoring context to specific values k and n, $\mathsf{setl}(k, n).M$, or else update to the current timestamp of a location k, $\mathsf{sync}(k).M$,
- migrate to location k, go k. M, and
- report success, ok, or failure, fail.

Shorthand: We often elide trailing stop processes. We thus represent asynchronous outputs such as c!$\tilde{v}$.stop as c!$\tilde{v}$ and branches such as if u=v then P else stop as if u=v then P. We also denote $k\{\!|M|\!\}^{(l,n)}$ as syntactic sugaring for $k[\![\{M\}^{(l,n)}]\!]$.

Our calculus describes distributed, event-based, asynchronous monitoring. Monitoring is *asynchronous* because it happens in two phases, whereby the operational mechanism for tracing is detached from the operational mechanism for querying the trace. This two-set setup closely reflects the limits imposed by a distributed setting and is more flexible with respect to the various monitoring mechanisms we

want to capture. Monitoring is *event-based* because we chose only to focus on recording and analysing discrete events such as communication.

For simplicity, our calculus only records effected output process events in traces (which dually imply effected inputs); we can however extend traces to record other effects of computation in straightforward fashion. In order to extract realistic temporal ordering of traces across locations, the calculus provides two mechanisms for monitor re-alignment: the coarser (and real-time) sync operation is used to start monitoring from a particular instant in time; the more explicit context update setl enables hard-coded control of relative timing at the level of monitors. For instance, together with getl, it enables decomposed monitoring to hand over tracing at a specific index in a local trace. This mechanism gives more control and can improve distributed monitoring precision, but may also lead to unsound monitoring.

Tracing: Whenever communication occurs (possibly across two locations) on some channel c with values $\bar{v}$, a trace entity of the form $\mathbf{t}(c,\bar{v},n)$ is produced at the location where *the output resides*. The output location, which keeps a local counter, assigns the timestamp n to this trace entity and increments its counter to $n+1$. Local timestamps induce a partial-order amongst all trace entities *across the system*. In particular, we obtain a finite (totally ordered) chain of traces *per location*.

***Example* 3.1** (Distributed Tracing)**.** *Consider the distributed system of outputs:*

$$l[\![c_1!v_1]\!] \;\|\; l[\![c_2!v_2]\!] \;\|\; k[\![c_3!v_3]\!]$$

Assuming that locations l and k have the respective timestamp counters n and m, once all outputs are consumed we can obtain either of the following possible sets of trace entities (for simplicity, we assume that $v_1 \neq v_2$):

$$l[\![\mathbf{t}(c_1,v_1,n)]\!],\ l[\![\mathbf{t}(c_2,v_2,n+1)]\!],\ k[\![\mathbf{t}(c_3,v_3,m)]\!] \tag{1}$$

$$l[\![\mathbf{t}(c_1,v_1,n+1)]\!],\ l[\![\mathbf{t}(c_2,v_2,n)]\!],\ k[\![\mathbf{t}(c_3,v_3,m)]\!] \tag{2}$$

The timestamps of trace-set (1) record the fact that the output on c_1 was consumed before *that on c_2, whereas those of trace-set (2) record the opposite. However, in both of these trace-sets, the timestamp assigned to the trace-entity relating to c_3, recorded at location k, does not indicate the order it was consumed, relative to the outputs on c_1 and c_2, which occurred at location l.*

Concurrent Monitoring: In our model, traces may be queried by multiple concurrent entities, which allows for better separation of concerns when monitors are instrumented. Trace querying is performed exclusively by monitors, $l\{\![M]\!\}^{(k,n)}$, parameterised by the monitoring context (k,n); indicating that monitor M is interested in analysing the n^{th} trace record at location k.

***Example* 3.2** (Parallel Monitoring)**.** *Consider the trace-set (1) from Example 3.1. A (local) monitor determining whether, from timestamp n onwards, a value v_2 was communicated on the first output on channel c_2 at location l, can be expressed as:*

$$l\{\![\mathbf{q}(c_2,x).\mathit{if}\ x = v_2\ \mathit{then\ ok\ else\ fail}]\!\}^{(l,n)}$$

The counter n of this monitor indicates that it starts analysing the trace-set (1) from the trace entity $l[\![\mathbf{t}(c_1,v_1,n)]\!]$ and continues moving up the chain of trace entities until the first trace entity describing outputs on channel c_2 is encountered. Since $l[\![\mathbf{t}(c_1,v_1,n)]\!]$ states that the event occurred on another channel, namely c_1, the monitor skips the irrelevant trace entity and its index is incremented to $n+1$ i.e.,

$l\{\!|\mathbf{q}(c_2,x).\mathit{if}\ x = v_2\ \mathit{then}\ \mathsf{ok}\ \mathit{else}\ \mathit{fail}|\!\}^{(l,n+1)}$. *The monitor analyses the trace entity with the incremented timestamp* $n+1$ *i.e.,* $l[\![\mathbf{t}(c_2,v_2,n+1)]\!]$*, which happens to match the required event on channel* c_2*; the monitor thus substitutes* v_2*, obtained from the trace entity* $l[\![\mathbf{t}(c_2,v_2,n+1)]\!]$*, for* x *and proceeds with the monitor processing, which should eventually yield* ok*. Traces can be concurrently queried by multiple monitors. For instance, consider another monitor running in parallel with the previous one of the form:*

$$l\{\!|\mathbf{q}(c_1,x).\mathbf{q}(c_2,y).\mathit{if}\ x = y\ \mathit{then}\ \mathsf{ok}\ \mathit{else}\ \mathit{fail}|\!\}^{(l,n)}$$

which checks that equal values are communicated on channels c_1 *and* c_2*. Thus it is important that trace entities, such as* $l[\![\mathbf{t}(c_2,v_2,n+1)]\!]$*, are persistent and* not *consumed once analysed, as in the case of outputs in a message passing setting.*

Distributed Monitoring: Distribution adds another dimension of complexity to monitor instrumentation, in terms of how to partition monitors across locations and how this partitioning evolves as computation progresses. In our calculus, *remote* querying can be syntactically expressed as $k\{\!|\mathbf{q}(c,x).M|\!\}^{(l,n)}$ for some c and n, where $k \neq l$. We can also describe the different classifications of distributed monitoring outlined earlier in Section 2.

***Example* 3.3.** *Consider a distributed system*

$$\mathit{Sys} \triangleq l[\![c_1?x.c_2!x]\!] \parallel \mathit{new}d.(\, k[\![d?x.c_2!x]\!] \parallel l[\![c_1!v.d!v.c_2?x.\mathsf{stop}]\!]\,)$$

where $c_1?x.c_2!x$ *and* $c_1!v.d!v.c_2?x.\mathsf{stop}$ *are located at* l *whereas* $d?x.c_2!x$ *is located at* k*. Moreover, process* $d?x.c_2!x$ *shares a scoped channel* d *with* $c_1!v.d!v.c_2?x.\mathsf{stop}$*. For some timestamps* n *and* m*,* Sys *non-deterministically produces either of the traces (3) or (4) below; the non-determinism is caused by the competition for the output on channel* c_2 *by respective inputs at* l *and* k*:*

$$\mathit{new}d.(\, l[\![\mathbf{t}(c_1,v,n)]\!],\ l[\![\mathbf{t}(d,v,n+1)]\!],\ l[\![\mathbf{t}(c_2,v,n+2)]\!]\,) \tag{3}$$

$$\mathit{new}d.(\, l[\![\mathbf{t}(c_1,v,n)]\!],\ l[\![\mathbf{t}(d,v,n+1)]\!],\ k[\![\mathbf{t}(c_2,v,m)]\!]\,) \tag{4}$$

The preservation of the property 'Whenever a value e *is communicated on the first output on* c_1 *at location* l*, then this value is* ***not*** *output on a subsequent output on channel* c_2 *at* k*' in general cannot be adequately determined statically, due to the non-deterministic nature of the computation, as exhibited by the possible traces (3) and (4). However, the property can be monitored at runtime in a number of ways:*

$$\begin{array}{lcl} M^{orch} & \triangleq & h\{\!|\mathbf{q}(c,x).\ \mathit{if}\ x = e\ \mathit{then}\ \mathsf{sync}(k).\ \mathbf{q}(c,y).\ \mathit{if}\ x = y\ \mathit{then}\ \mathit{fail}|\!\}^{(l,n)} \\ M^{chor} & \triangleq & \mathit{new}d'.(\, l\{\!|\mathbf{q}(c,x).\ \mathit{if}\ x = e\ \mathit{then}\ d'!x|\!\}^{(l,n)} \parallel k\{\!|d'?x.\ \mathsf{sync}(k).\ \mathbf{q}(c,y).\ \mathit{if}\ x = y\ \mathit{then}\ \mathit{fail}|\!\}^{(k,m)}\,) \\ M^{mig} & \triangleq & l\{\!|\mathbf{q}(c,x).\ \mathit{if}\ x = e\ \mathit{then}\ \mathit{go}\ k.\ \mathsf{sync}(k).\mathbf{q}(c,y).\mathit{if}\ x = y\ \mathit{then}\ \mathit{fail}|\!\}^{(l,n)} \end{array}$$

M^{orch} *monitors for this property in* orchestrated *fashion, querying traces at both* l *and* k *from a remote central location* h*; this monitor is well-aligned with location* l *to start with, but has to explicitly re-align with location* k *once monitoring shifts to that location.* M^{chor} *is an instance of a* choreographed *monitor setup, instrumenting local monitors at each location where trace querying needs to be performed, namely* l *and* k*. These local monitors synchonise between them using remote communication on the scoped channel* d'*. Note that the monitor at* k *updates its context upon channel synchronisation on* d' *to ensure a temporal ordering on analysed trace records; without synchronisation, the monitor would potentially be reading past parts of the trace which may lead to unsound sequentiality conclusions. Finally,* M^{mig} *is a case of a* migrating monitor*, that starts monitoring at location* l *but then migrates to location* k *if it needs to continue monitoring there, re-aligning its index to that of the destination location.*

All three distributed monitors in Example 3.3 are sound *wrt.* the property stated, in the sense that they never falsely flag a violation. They are nevertheless incomplete, and may miss out on detecting property violations. For instance, M^{orch} may realign with location k after the trace $k[\![\mathbf{t}(c_2, v, m)]\!]$ is generated by k, which sets the monitor timestamp index to $(k, m+1)$. This forces the monitoring to start querying the trace at k from index $m+1$ and will therefore skip the relevant trace item $k[\![\mathbf{t}(c_2, v, m)]\!]$. This aspect is however not a limitation of our encoding, but rather an inherent characteristic of distributed computing as discussed earlier in Section 2.

4 Monitoring Semantics

We define the semantics of mDPı in terms of a number of related Labelled Transition Systems (LTSs), which are then used to compare systems through the standard notion of weak-bisimulation equivalence, denoted here as $\approx$. This framework allows us to state and prove properties from a behavioural perspective about our monitored systems. For instance, we could express the fact that, ignoring monitoring location, M^{orch} and M^{chor} from Example 3.3 monitor for the same properties *wrt. Sys*, using the statement:

$$Sys \parallel M^{\text{orch}} \approx Sys \parallel M^{\text{chor}} \tag{5}$$

Using an LTS that does not express observable monitor actions, the property that a monitor, say M^{orch}, does not affect the observable behaviour of the system *Sys* could be stated as:

$$Sys \approx Sys \parallel M^{\text{orch}} \tag{6}$$

Using different LTSs, the same system could be assigned more restricted behaviour. For instance, this is useful to ensure that the monitor M^{mig} of Example 3.3 does not perform remote querying at any stage during its computation by establishing the comparison:

$$Sys \parallel M^{\text{mig}} \approx ((Sys \parallel M^{\text{mig}}) \text{ without remote monitoring}) \tag{7}$$

where the lefthand system is subject to an LTS allowing remote querying whereas the righthand monitor is subject to an LTS that prohibits it. Intuitively, if the behaviour is preserved when certain internal moves are prohibited, this means that these moves are not used (in any useful way) by the monitor.

4.1 Deriving LTSs Modularly

Closer inspection of the comparisons (5), (6) and (7) reveals that the different LTSs required are still expected to have substantial common structure; typically they would differ with respect to either the information carried by actions and/or the type of actions permitted. For instance, in (5) we would want actions that *restrict* information relating to the location of where monitoring is carried out, as this additional information would distinguish between the two monitors. On the other hand, for (7) we would want to *prohibit* actions relating to remote monitoring.

We therefore construct these related LTSs in modular fashion through the use of a *preLTS*, *i.e.*, an LTS whose transitions relate more systems, and whose action labels carry more information than actually needed. The excess transitions and label information are then pruned out as needed by a *filter function* from actions in the preLTS to actions in the LTS required.

4.2 A preLTS for mDPi

Our preLTS is defined over systems subject to a *local logical clock* at every location used by the system, which are used to generate ordered trace-entities and to re-align monitors. These clocks are modelled as monotonically increasing counters and expressed as a partial function $\delta \in \Delta :: \text{Locs} \rightharpoonup \mathbb{N}$, where $\delta(l)$ denotes the next timestamp to be assigned for a trace entity generated at l. Moreover, the counter increment is defined using standard function overriding, $inc(\delta, k) = \delta[k \mapsto (\delta(k)+1)]$.

$$\textsc{OutP}\ \frac{}{\delta \rhd k[\![c!d.P]\!] \xrightarrow{c!\bar{d}_{\langle p:k,l\rangle}} \mathsf{inc}(\delta,k) \rhd k[\![P]\!] \parallel k[\![\mathbf{t}(c,\bar{d},\delta(k))]\!]} \qquad \textsc{InP}\ \frac{}{\delta \rhd l[\![c?x.P]\!] \xrightarrow{c?\bar{d}_{\langle p:k,l\rangle}} \delta \rhd l[\![P\{\bar{d}/\bar{x}\}]\!]}$$

$$\textsc{OutT}\ \frac{}{\delta \rhd k[\![\mathbf{t}(c,\bar{d},n)]\!] \xrightarrow{c!\bar{d}_{\langle t:k,l:n\rangle}} \delta \rhd k[\![\mathbf{t}(c,\bar{d},n)]\!]} \qquad \textsc{InT}\ \frac{}{\delta \rhd l\{\![\mathbf{q}(c,\bar{x}).M]\!\}^{(k,n)} \xrightarrow{c?\bar{d}_{\langle t:k,l:n\rangle}} \delta \rhd l\{\![M\{\bar{d}/\bar{x}\}]\!\}^{(k,n+1)}}$$

$$\textsc{OutM}\ \frac{}{\delta \rhd k\{\![c!d.M]\!\}^{(l,n)} \xrightarrow{c!\bar{d}_{\langle m:k,h\rangle}} \delta \rhd k\{\![M]\!\}^{(l,n)}} \qquad \textsc{InM}\ \frac{}{\delta \rhd l\{\![c?x.M]\!\}^{(k,n)} \xrightarrow{c?\bar{d}_{\langle m:h,l\rangle}} \delta \rhd l\{\![M\{\bar{d}/\bar{x}\}]\!\}^{(k,n)}}$$

$$\textsc{Open}\ \frac{\delta \rhd S \xrightarrow{(\bar{b})c!\bar{d}_\gamma} \delta' \rhd S'}{\delta \rhd \mathsf{new}\, b.S \xrightarrow{(b,\bar{b})c!\bar{d}_\gamma} \delta' \rhd S'}\ [b \neq c, b \in \bar{d}] \qquad \textsc{Res}\ \frac{\delta \rhd S \xrightarrow{\mu} \delta' \rhd S'}{\delta \rhd \mathsf{new}\, b.S \xrightarrow{\mu} \delta' \rhd \mathsf{new}\, b.S'}\ [b \notin \mathsf{FN}(\mu)]$$

$$\textsc{Com1}\ \frac{\delta \rhd S \xrightarrow{(\bar{b})c!\bar{d}_\gamma} \delta' \rhd S' \qquad \delta \rhd R \xrightarrow{c?\bar{d}_\gamma} \delta \rhd R'}{\delta \rhd S \parallel R \xrightarrow{\tau_\gamma} \delta' \rhd \mathsf{new}\, \bar{b}.(S' \parallel R')}\ [\bar{b} \cap \mathsf{FN}(R) = \emptyset]$$

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

$$\textsc{SetI}\ \frac{}{\delta \rhd k\{\![\mathsf{setl}(h,m).M]\!\}^{(l,n)} \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \rhd k\{\![M]\!\}^{(h,m)}} \qquad \textsc{Sync}\ \frac{}{\delta \rhd k\{\![\mathsf{sync}(l).M]\!\}^{(h,n)} \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \rhd k\{\![M]\!\}^{(l,\delta(l))}}$$

$$\textsc{GetI}\ \frac{}{\delta \rhd k\{\![\mathsf{getl}(x,y).M]\!\}^{(l,n)} \xrightarrow{\tau_{\langle m:k,k\rangle}} \delta \rhd k\{\![M\{^{l,n}/_{x,y}\}]\!\}^{(l,n)}} \qquad \textsc{Go}\ \frac{}{\delta \rhd k\{\![\mathsf{go}\ l.\ M]\!\}^{(h,n)} \xrightarrow{\tau_{\langle m:k,l\rangle}} \delta \rhd l\{\![M]\!\}^{(h,n)}}$$

Figure 2: mDPi preLTS main rules

A *Configuration* $C, D \in \text{Conf} :: \Delta \times \text{Sys}$ is thus a system subject to a set of localised counters. The preLTS is a ternary relation $\rightarrow :: \text{Conf} \times \text{pAct} \times \text{Conf}$, denoted using the suggestive notation $C \xrightarrow{\mu} D$, where $\mu \in \text{pAct}$ is a preLTS action label of the form, τ_γ, an internal action, $(\bar{b})c!\bar{d}_\gamma$, an output action, or $c?\bar{d}_\gamma$, an input action. In case of the output action, $\bar{b}$ denotes the (possibly empty) set of channel names exported during an eventual interaction. These actions are standard [9], but are decorated with additional information γ which can be of the following three formats:

$\langle p : l,k\rangle-$ This states that it is a *process* (p) action, involving locations l and k.

$\langle m : l,k\rangle-$ This states that it is a *monitor* (m) action, involving locations l and k.

$\langle t : l,k : n\rangle-$ This states that it is a *trace* (t) action at timestamp n, involving locations l and k, .

The main rules defining the relation $C \xrightarrow{\mu} D$ are outlined in Figure 2. The rule for process input, InP, is standard, except for the additional label tag $\langle p : k,l\rangle$ encoding the fact that the input is a process input, it resides at location l, and is reading from some location k (when communication is local, then $l = k$). A central rule to our monitoring semantics is OutP. Apart from the additional label decoration, it differs from standard output rules in two respects: first it generates a trace entity, $k[\![\mathbf{t}(c,\bar{d},\delta(k))]\!]$, recording the channel name, c, the values communicated, d, timestamped by $\delta(k)$, and second, it increments the clock at k once the trace entity is generated, necessary for generating a total order of trace-entities at k. Monitor communication, defined by rules OutM and InM, is similar albeit simpler since neither trace entities are generated, nor is the local counter updated[1].

Rule OutT models trace actions as output labels with tags $\langle t : k,l : n\rangle$, where the timestamp of the trace, n, is recorded in the tag as well. Crucially, the trace entity is not consumed by the action (thereby acting as a broadcast), and its persistence allows for multiple monitors to query it. This action can be matched by a query action, InT, expressed as an input action with a matching tag $\langle t : k,l : n\rangle$ where the source location of the trace entity, k, and time stamp n must match the current monitoring context (k,n). Since the action describes the fact that a trace entity has been matched by the monitor query, the timestamp index of the monitoring context is incremented, $(k,n+1)$ to progress to the next entitity in the local trace log.

Scope extrusion of channel names may occur both directly, through process or monitor communication, or else indirectly through trace querying; these are both handled by the standard scoping rules Open and Res. All three forms of communication *i.e.,* process, monitor and trace, are also handled uniformly, this time by the communication rule Com1 (we here elide its symmetric rule). Communication yields a silent action τ_γ that is decorated with the corresponding tagging information from the constituent input and output actions of the premises. This tagged information must match for both input and output actions and, in the case of the trace tags, $\langle t : k,l : n\rangle$, this also implies a matching of the timestamp n. When, for a particular timestamp, querying does not match the channel of the trace entity at that timestamp, rule Skip allows the monitor to increase its timestamp index and thus querying to move up the trace-chain at that location. Finally, Sync allows monitors to realign with a trace at a particular location, GetI and SetI allow for explicit manipulations of the monitoring context whereas Go describes monitor migration.

4.3 Filter Functions

Although necessary to encode extended information of system execution, the preLTS presented is too discriminating. For instance, the internal action τ_γ is now compartementalised into distinct silent actions, each identified by the tag information γ, which complicates their use for weak actions when verifying bisimilar configurations. Similarly, external actions differentiating between a process or a monitor carrying out that action may also be deemed to discriminating. Finally, we may also want to disallow certain actions such as remote trace querying.

We obtain LTSs with the necessary level of discriminating actions using (i) the preLTS of Section 4.2, together with (i) a filter function, Ω. This function maps actions in the preLTS, $\mu \in$ pAct, to actions in

[1]Note that rule OutM refers to an indeterminate location h, to match a reader in any such location.

the required LTS, $\alpha \in \text{Act}$, through the rule:

$$\textsc{Fltr}\ \frac{C_1 \xrightarrow{\mu} C_2}{C_1 \xrightarrow{\alpha}_{\Omega} C_2}\ [\Omega(\mu) = \alpha]$$

Notation: Note that filter function applications are essentially abstractions of the preLTS. LTSs obtained in this manner can effectively be indexed by their respective filter function, Ω, and for clarity we denote a configuration C subject to a behaviour obtained from the preLTS and a filter function Ω as C_Ω. We also denote transitions obtained in this form as $C_1 \xrightarrow{\alpha}_{\Omega} C_2$.

***Example* 4.1** (Filter Functions)**.** *Consider the following filter function definitions:*

$$\begin{array}{lcl} \Omega_{NTg}(\tau_\gamma) & \triangleq & \tau \\ \Omega_{NTg}(c!\bar{d}_\gamma) & \triangleq & c!\bar{d} \\ \Omega_{NTg}(c?\bar{d}_\gamma) & \triangleq & c?\bar{d} \end{array} \qquad \begin{array}{lcl} \Omega_{Prc}(\tau_\gamma) & \triangleq & \tau \\ \Omega_{Prc}(c!\bar{d}_{\langle p:l,k\rangle}) & \triangleq & c!\bar{d}_{\langle l,k\rangle} \\ \Omega_{Prc}(c?\bar{d}_{\langle p:l,k\rangle}) & \triangleq & c?\bar{d}_{\langle l,k\rangle} \end{array} \qquad \begin{array}{lcl} \Omega_{LTr}(\tau_{\langle t:l,l:n\rangle}) & \triangleq & \tau \\ \Omega_{LTr}(\tau_{\langle \_:l,k\rangle}) & \triangleq & \tau \\ \Omega_{LTr}(c!\bar{d}_\gamma) & \triangleq & c!\bar{d} \\ \Omega_{LTr}(c?\bar{d}_\gamma) & \triangleq & c?\bar{d} \end{array}$$

Ω_{NTg} removes all tags from decorated actions, which in turn allows for a straightforward definition of weak actions, $\stackrel{\hat{\alpha}}{\Longrightarrow}$, as $(\xrightarrow{\tau})^*$ if $\alpha = \tau$ and $(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^*$ otherwise. In addition to stripping τ action tags, the second filter function, Ω_{Prc}, allows only process external actions, filtering out the p component in this case. The function is partial as it is undefined for all other preLTS actions. This is useful when we do not want to discriminate configurations based on the tracing and monitoring actions. The final filter function, Ω_{LTr}, removes all tags but uses them to prohibit silent tracing actions where the two locations in the tag are distinct; this in effect rules out remote trace querying, thereby enforcing localised trace monitoring.

We assume a certain well-formedness criteria on our filter functions, such as that they do not change the form of an action (*e.g.*, an output action remains an output action), and that whenever they map to silent actions, τ, these are not decorated; the filter functions in Example 4.1 satisfy these criteria. Through this latter requirement we re-obtain the standard silent τ-action at the LTS level.

4.4 Behavioural Equivalence

The technical development in sections 4.2 and 4.3 allows us to immediately apply weak bisimulation [9] as a coinductive proof technique for equivalence between LTSs obtained for our preLTS and well-formed filter functions. Two (filtered) LTSs, C_{Ω_1} and D_{Ω_2}, are bisimilar, denoted as $C_{\Omega_1} \approx D_{\Omega_2}$, if they match each other's transitions; we use the weak bisimulation variant, $\approx$, as this abstracts over internal τ-actions which yields a more natural extensional equivalence.

***Example* 4.2.** *Using the filter functions defined in Example 4.1, we can formally state and prove equivalences (5), (6) and (7) outlined earlier, for a localised clock-set δ including locations l and k:*

$$(\delta \rhd Sys \parallel M^{orch})_{\Omega_{NTg}} \quad \approx \quad (\delta \rhd Sys \parallel M^{chor})_{\Omega_{NTg}} \tag{8}$$

$$(\delta \rhd Sys)_{\Omega_{Prc}} \quad \approx \quad (\delta \rhd Sys \parallel M^{orch})_{\Omega_{Prc}} \tag{9}$$

$$(\delta \rhd Sys \parallel M^{mig})_{\Omega_{NTg}} \quad \approx \quad (\delta \rhd Sys \parallel M^{mig})_{\Omega_{LTr}} \tag{10}$$

Equivalence (8) formalises the behaviour expected for (5) using an LTS whose actions prohibit distinctions based on action tags; including monitoring location, i.e., Ω_{NTg}. Since in (6) we wanted to analyse

how monitors affect process computation, in its corresponding equivalence (9) we use an LTS that tags process external actions with location information while prohibiting any actions relating to tracing or monitoring. Finally (10) compares M_{mig} with itself, subject to a restricted semantics where remote monitoring is prohibited, i.e., Ω_{LTr}.

5 Instrumentation for Distributed Monitoring

The instrumentation of contracts as distributed monitors is non-trivial and can easily lead to unsound contract monitoring. In this section we illustrate how the instrumentation of contracts, expressed using a simple regular expression-based temporal logic specifying violation traces, can be safely automated according to different monitoring approaches. The syntax of the contract language is:

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

Basic events have the form $(c, \bar{v})@k$ indicating that a communication on channel c with value $\bar{v}$ occurs at location k. We adopt a semantics allowing for multiple matches, rather than opt only for the shortest match[2] and thus any trace terminating with a communication $c!\bar{v}$ at location k is considered to be a violating trace. The other operators are the standard ones used in regular expressions: $E.F$ corresponds to the traces which can be split into two, with the first matching E, and the second matching F; expression E^* corresponds to traces which can be split into a number (possibly zero) parts, each of which satisfies E; and $E + F$ corresponds to the set of traces which match either E or F.

Notation: $\sum_{e \in I} E$ corresponds to the generalised choice over finite I, which is equal to $E\{^{i_1}/_e\} + E\{^{i_2}/_e\} + \ldots E\{^{i_n}/_e\}$ (where $I = \{i_1, i_2 \ldots, i_n\}$).

Despite the apparent simplicity of this expository contract language, we can already express interesting contracts.

***Example* 5.1.** *Consider a simplification of the contract outlined in Example 2.1:* "The release of a patient's record must be approved by supervising doctors." *Stated in terms of what leads to a violation, we get:* "If a patient's medical record is released regardless of a doctor's disapproval, the contract is violated" *which can be expressed as the regular expression:*

$$\sum_{p \in Patient} (\mathsf{req}, ())@p. \sum_{d \in Doctor} (\mathsf{withhold}, p)@d \;.\; (\mathsf{send}, p)@h$$

where p, d and h are locations referring to the patient's, the doctor's and the hospital domain, channel names req, withhold *and* send *denote actions requesting, withholding and sending medical records, and sets Patient and Doctor range over the finite patients and doctors in the system.*

There are different ways in which one may transform a regular expression into an mDPi term. For instance, $(c_1, \overline{v_1})@k_1.(c_2, \overline{v_2})@k_2$ may be matched by either one monitoring process, M_1, or by the split monitors, M_2, below:

$$M_1 \triangleq \mathsf{sync}(k_1).\mathbf{q}(c_1, \overline{x_1}).\mathsf{if}\ \overline{x_1} = \overline{v_1}\ \mathsf{then}\ (\mathsf{sync}(k_2).\mathbf{q}(c_2, \overline{x_2}).\mathsf{if}\ \overline{x_2} = \overline{v_2}\ \mathsf{then}\ \mathsf{fail})$$

$$M_2 \triangleq (\mathsf{sync}(k_1).\mathbf{q}(c_1, \overline{x_1}).\mathsf{if}\ \overline{x_1} = \overline{v_1}\ \mathsf{then}\ m!) \parallel (m?.\mathsf{sync}(k_2).\mathbf{q}(c_2, \overline{x_2}).\mathsf{if}\ \overline{x_2} = \overline{v_2}\ \mathsf{then}\ \mathsf{fail})$$

[2] In any case, when runtime monitoring one may choose to halt the system on the shortest match.

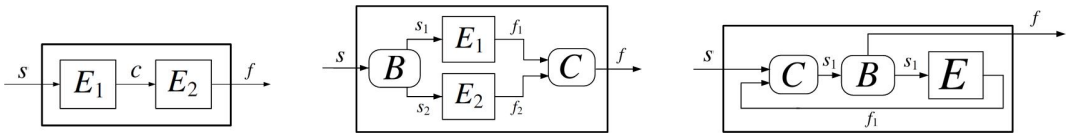

Figure 3: Compiling $E.F$, $E+F$ and E^* (respectively) where C, B correspond to comb, bifurc.

We find that the second translation, M_2, lends itself better towards illustrating how monitors can be distributed in different ways across locations — for example, in an orchestrated approach, we would place all the monitoring processes in a single location, while in a choreographed approach, we would distribute the processes as required. A sequential approach such as M_1, may be more approriate in an orchestrated approach (since it avoids unnecessary parallelism), but would not be possible to distribute to enable choreographed monitoring without further manipulation.

In this paper, we adopt the maximally parallelised approach, primarily to be able to observe similarities and distinctions between different compilation approaches. In particular we use this translation for a compilation strategy corresponding closely to standard approaches used in hardware compilation of regular expressions [6], producing circuits with two additional wires: an input which is signaled upon to start matching the regular expression, and an output wire which the circuit uses to signal a match with the regular expression. In our case, the wires correspond to channels: basic events $(c,\overline{v})@k$ with start channel s and match channel f would be translated into an expression which waits for input on channel s, then outputs on channel f when an instance of c with $\overline{v}$ occurs at location k. Our translations also employ two standard monitor organisations for funneling two output signals into one and forking channel communication onto two separate ones; these are expressed below as the macros comb and bifurc:

$$\text{comb}(f_1,\ f_2,\ f) \triangleq *(f_1?\bar{x}.f!\bar{x}) \parallel *(f_2?\bar{x}.f!\bar{x}) \qquad \text{bifurc}(s,\ s_1,\ s_2) \triangleq *s?\bar{x}.(s_1!\bar{x} \parallel s_2!\bar{x})$$

We define three compilation strategies, ψ_O, ψ_C and ψ_M, corresponding respectively to monitoring using orchestration, static choreography and migrating monitoring as discussed in section 2. The compilation procedures use three parameters: two control channels (used to notify when the regular expression is to start being matched, and to notify when it has matched) and the expression to be compiled. For simplicity, all three translations follow a similar pattern shown by the block diagrams in Figure 3, varying only in the location placement of the monitors and synchronisation strategy.

5.1 Orchestration-Based Monitoring Translation

Orchestration places the monitor at some predefined central location, h. As stated earlier, the lack of a global clock prevents it from deducing with certainty the order of events happening across different locations. Nevertheless, our translation attempts to mitigate this imprecision for sequence of events occurring *at the same location* using the following mechanism: when a basic event is matched, the monitoring context, (k,n), at that moment is recorded using getl and passed as arguments on the match signaling channel; this allows subsequent matching to explicitly adjust the monitoring context to these values using setl in cases where the location of subsequent events does not change; in cases where the location changes, this information is redundant and alignment is carried out using the coarser sync command.

$$\psi'_O(s, f, ((c,\overline{v})@k)) \triangleq *(s?(x_{\text{loc}}, x_{\text{idx}}).\text{if } k = x_{\text{loc}} \text{ then } (\text{setl}(x_{\text{loc}}, x_{\text{idx}}).\text{trg}(c,\overline{v},f)) \text{ else } (\text{sync}(k).\text{trg}(c,\overline{v},f)))$$

In either case, a listener triggers a signal with the updated index of the trace on the match channel for every event c with data $\bar{v}$. Specifically, the macro $\mathrm{trg}(c, \bar{v}, f)$ repeatedly reads from channel c, outputting the monitoring-context information on the matching channel f every time the data traced matches $\bar{v}$:

$$\mathrm{trg}(c, \bar{v}, f) \triangleq *(\mathbf{q}(c, \bar{x}).\mathsf{if}\ \bar{x} = \bar{v}\ \mathsf{then}\ \mathsf{getl}(x_{\mathrm{loc}}, x_{\mathrm{idx}}).f!(x_{\mathrm{loc}}, x_{\mathrm{idx}}))$$

The compilation of the regular expression operators matches the compilation schemata of figure 3:

$$\begin{array}{rcl}
\psi'_O(s, f, (E.F)) & \triangleq & \mathsf{new}\, m.(\ \psi'_O(s, m, E) \parallel \psi'_O(m, f, F)) \\
\psi'_O(s, f, E^*) & \triangleq & \mathsf{new}\, c, s', f'.\ \mathrm{comb}(s, f', c) \parallel \mathrm{bifurc}(c, s', f) \parallel \psi'_O(s', f', E) \\
\psi'_O(s, f, (E + F)) & \triangleq & \mathsf{new}\, s_1, s_2, f_1, f_2.\Big(\mathrm{bifurc}(s, s_1, s_2) \parallel \psi'_O(s_1, f_1, E) \parallel \psi'_O(s_2, f_2, F) \parallel \mathrm{comb}(f_1, f_2, f)\Big)
\end{array}$$

The combined monitors are located at the predefined central location h, with a dummy initial monitor context continuation parameters $(h, 1)$. The monitor induced for a contract E is thus:

$$\psi_O(E) \triangleq h\{\!|\ \mathsf{new}\, s, f.(\ s!(h, 1) \quad \parallel \quad \psi'_O(s, f, E) \quad \parallel \quad f?\bar{x}.\mathsf{fail}\ )\ |\!\}^{(h,1)}$$

5.2 Choreography-Based Monitoring Translation

Instead of instrumenting the whole monitor at a single central location, a choreography-based approach decomposes the monitor into parts, possibly placing them at different locations. Once again, monitors are made up of two kinds of components: (i) the event listeners; and (ii) the choreography control logic made up of comb and bifurc components. The event listeners are located locally, where the event takes place, but are otherwise exactly the same as in the orchestrated approach:

$$\psi'_C(s, f, ((c, \bar{v})@k)) \triangleq k\{\!|[\psi'_O(s, f((c, \bar{v})@k))]|\!\}^{(k,1)}$$

On the other hand, the choreography control logic can be placed at any location. For instance one may choose to locate them at the node where the next input will be expected, or where the last one occurred. For a particular choice of locations l and h, choice $E + F$ is compiled as follows:

$$\mathsf{new}\, s_1, s_2, f_1, f_2.\Big(l\{\!|[\mathrm{bifurc}(s, s_1, s_2)]|\!\}^{(l,1)} \parallel \psi'_C(s_1, f_1, E) \parallel \psi'_C((s_2, f_2), F) \parallel h\{\!|[\mathrm{comb}(f_1, f_2, f)]|\!\}^{(h,1)}\Big)$$

Finally, we add the necessary start signal (from some start location k) to initiate the monitoring:

$$\psi_C(E) \triangleq \mathsf{new}\, s, f.\ k\{\!|[s!(k, 1) \parallel f?\bar{x}.\mathsf{fail}]|\!\}^{(k,1)} \parallel \psi'_C(s, f, E)$$

Note that unless all the locations enable the execution of new (monitoring) process at runtime, the contracts must be known at compile-time, which is guaranteed in the simple regular expression logic we are using.

5.3 Migrating Monitors Translation

For the migrating monitors technique, we use a simplified translation where the monitors generated are similar to the ones used in orchestration, except that the monitor migrates when required to the relevant location (using the go operator). ψ'_M is defined identical to ψ'_O except for basic events:

$$\psi'_M(s, f, ((c, \bar{v})@k)) \triangleq *(s?(x_{\mathrm{loc}}, x_{\mathrm{idx}}).\mathsf{go}\, k.\ \mathsf{if}\ k = x_{\mathrm{loc}}\ \mathsf{then}\ (\mathsf{setl}(x_{\mathrm{loc}}, x_{\mathrm{idx}}).\mathrm{trg}(c, \bar{v}, f))\ \mathsf{else}\ \mathsf{sync}(k).\mathrm{trg}(c, \bar{v}, f))$$

Note how migration (thus monitor instrumentation) is delayed and happens only once the start signal on channel s is received. Initially, the monitor can be chosen to reside anywhere. For a particular location choice h, the migrating monitor approach for a contract E would be the following:

$$\psi_M(E) \triangleq h\{\!\![\ \mathsf{new}\, s, f.(\ s!(h,1) \quad \| \quad \psi'_M(s,f,E) \quad \| \quad f?\bar{x}.\mathsf{fail}\ )\ ]\!\}^{(h,1)}$$

Despite the resemblances resulting from our simplistic translations, migration improves on an orchestrated approach by avoiding remote tracing. As in the choreographed approach, one can also choose to explicitly run the combining and bifurcation processes at particular locations by adding explicit migration instructions. A better approach would be to nest all the monitors within each other to avoid monitors migrating or installed before they are actually required. For example, monitoring for an expression of the form: $(c_1, \overline{v_1})@l.(c_2, \overline{v_2})@k.(c_3, \overline{v_3})@h$ would be transformed into a monitor of the form:

$$\mathsf{go}\ l.\ (\mathbf{q}(c_1, \overline{x_1}).\mathsf{if}\ \overline{x_1} = \overline{v_1}\ \mathsf{then}\ \mathsf{go}\ k.\ (\mathbf{q}(c_2, \overline{x_2}).\mathsf{if}\ \overline{x_2} = \overline{v_2}\ \mathsf{then}\ \mathsf{go}\ h.\ (\mathbf{q}(c_3, \overline{x_3}).\mathsf{if}\ \overline{x_3} = \overline{v_3}\ \mathsf{then}\ \mathsf{fail})))$$

Note that using this approach entails minimal local monitor instrumentation since this happens on a by-need basis: the translation avoids installing any monitor at location k unless $c_1!\overline{v_1}$ happens at l.

Even within this simplistic formal setting, migrating monitors can be seen to be more versatile than a choreographed approach. For instance, if our contract language is extended with variables and a binding construct, $\exists x.E$, we could express a more dynamic form of contract such as $\exists x.(c_1, x)@k.(c_2, v)@x$; in such a contract the location of the second event *depends* on the location communicated in the first event and, more importantly, this location is not known at compile time. Because of this last point, this contract cannot be handled adequately by traditional choreographed approaches which would need to preemptively instrument monitors at *every location*. However, in a migrating monitor approach, this naturally translates to a single runtime migration.

5.4 The Approaches and Limitations

We have shown how one can formulate different monitoring strategies of the same contract using mDPi. The contract language and its compilation procedure have intentionally been kept simple to avoid their complexity from obscuring the underlying monitoring choices. The different approaches mostly differ only in the location of the monitors. The migrating monitor approach also allows for straightforward setting up of new contracts at runtime, including references to locations not known at compile-time. Furthermore, the migrating approach procrastinates from setting up monitors in remote locations until necessary. In contrast, on a choreographed approach, monitors are set up at all locations, even though some of them may never be triggered.

Formalising the compilation of regular expression contracts into mDPi also gives us opportunities to formally verifying certain properties. For instance, as a generalisation of (8) we can state and prove that, for arbitrary expression E, different compilation approaches give the same monitoring result. We can state this as:

$$\delta \rhd Sys \,\|\, \psi_O(E))_{\Omega_{NTg}} \quad \approx \quad (\delta \rhd Sys \,\|\, \psi_C(E))_{\Omega_{NTg}} \quad \approx \quad (\delta \rhd Sys \,\|\, \psi_M(E))_{\Omega_{NTg}}$$

and prove it by giving witness bisimulations defined by induction on the structure of E. One can prove similar results on the lines of the equivalences given in section 4.4.

6 Conclusions

We have presented a novel unified framework, using a process calculus approach, where distributed contract monitoring can be formalised and rigorously compared. We have shown it to be expressive enough to encode various distributed monitoring strategies from the literature. To the best of our knowledge, the process calculus we present is also unique in that it internalises traces as first class computational entities rather than meta-constructs, together with non-intrusive monitoring. We modularly developed various semantics for this calculus, using transition abstraction techniques that enable selective reasoning about aspects such as locality of communication and distinctions between monitor and process actions.

We are currently working on an implementation in Erlang [2], guided by the design decisions made for our calculus. This should give us insight into practical issues, such as that of addressing trust issues when installing monitors and the avoidance of indirect data exposure due to monitoring. We are also studying mDPı further, addressing issues such as clock boundaries and real-time operators. As the calculus stands, the monitoring component is non-intrusive, in that it reads system events but does not otherwise interact with it. To handle reparations and enforcements upon contract violation, and to be able to express monitor-oriented programming [5] we require potentially intrusive monitoring. We believe that our bisimulation approach can also handle reasoning about monitor intrusiveness.

References

[1] C. Abela, A. Calafato & G.J. Pace: *Extending WISE with Contract Management*. In: *WICT 2010*.

[2] Joe Armstrong (2007): *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

[3] Fabio Barbon, Paolo Traverso, Marco Pistore & Michele Trainotti (2006): *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, pp. 63–71.

[4] David Chappell (2004): *Enterprise Service Bus*. O'Reilly Media.

[5] Feng Chen & Grigore Roşu (2003): *Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*. In: *Runtime Verification (RV'03)*, *ENTCS* 89(2), pp. 108 – 127.

[6] Koen Claessen & Gordon J. Pace (2002): *An Embedded Language Framework for Hardware Compilation*. In: *Designing Correct Circuits '02, Grenoble, France*.

[7] T. Cook, D. Drusinksy & M.-T. Shing (2007): *Specification, Validation and Run-time Moniroting of SOA Based System-of Systems Temporal Behaviors*. In: *System of Systems Engineering*, IEEE Computer Society.

[8] Dorothy E. Denning (1987): *An intrusion-detection model*. *IEEE Transactions on Software Engineering* 13, pp. 222–232.

[9] Matthew Hennessy (2007): *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA.

[10] I. H. Krüger, M. Meisinger & M. Menarini (2008): *Interaction-based Runtime Verification for Systems of Systems Integration*. *Computer Science and Engineering Dept., University of California, San Diego USA* .

[11] L. Lamport (1977): *Proving the Correctness of Multiprocess Programs*. *IEEE Trans. Softw. Eng.* 3, pp. 125–143.

[12] Masoud Mansouri-Samani & Morris Sloman (1997): *GEM: a generalized event monitoring language for distributed systems*. *Distributed Systems Engineering* 4(2), pp. 96–108.

[13] Koushik Sen, Abhay Vardhan, Gul Agha & Grigore Roşu (2004): *Efficient Decentralized Monitoring of Safety in Distributed Systems*. *International Conference on Software Engineering* , pp. 418–427.

[14] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo & Insup Lee (2009): *DMaC: Distributed Monitoring and Checking*. In: *Runtime Verification 09*, *LNCS* 5779, Springer, pp. 184–201.