Runtime Monitoring of Distributed Systems*

Adrian Francalanza
CS, ICT
University of Malta
adrian.francalanza@um.edu.mt

Andrew Gauci
CS, ICT
University of Malta
agau0006@um.edu.mt

Gordon J. Pace
CS, ICT
University of Malta
gordon.pace@um.edu.mt

ABSTRACT

Distributed and component-based architectures are becoming more prevalent computer systems. The increased complexities introduced by the distribution hampers dependability, emphasising the need for verification techniques tailored for a distributed setting. Runtime verification has proven to be a viable approach for verifying correctness, by focussing on the adherence of the runtime-generated trace to the desired properties. We present a broad taxonomy of current techniques to distributed monitoring, culminating in the proposal of a novel migrating monitor approach. We argue for certain situations where this approach presents clear advantages over current techniques.

1. INTRODUCTION

As systems become more complex, monolithic architectures are becoming less common, and distributed and component-based systems are becoming more mainstream. Distributed architectures introduce additional complexities such as computation/memory distribution, concerns for information confidentiality and architecture dynamicity. Such issues hamper system dependability and robustness, emphasising the need for techniques guaranteeing correctness tailored for distributed systems.

Software verification techniques traditionally include testing, model checking and runtime verification. Although testing is partly attractive due to its scalability, it is insufficient due to (i) its lack of program coverage, whereby testing can only find the presence and not prove the absence of bugs [8], (ii) the difficulty of effective test case generation, which when combined, offer 'reasonable' coverage of possible system behaviour. Model checking provides the highest guarantees but the state space explosion required by the modeling of even moderately-sized systems makes this approach impractical in most cases. This issue is further compounded by the asynchrony and concurrency inherent in distributed architectures [13].

Runtime verification [4, 11] is concerned with formally verifying the system trace generated at runtime. This process is carried out through an executable monitor verifying the generated trace against a set of desirable properties, with the system guaranteed to never go beyond a bad state undetected. Advantages with runtime verification include (i) the fact that it ensures that the system may be stopped the moment issues are identified in a tractable manner, (ii) trace generation is left to the system, (iii) verification continues beyond system deployment.

The following paper investigates the application of runtime verification to a distributed setting, while proposing a novel *migrating monitor* approach we believe is advantageous for monitoring certain scenarios of distributed systems. Section 2 presents distributed system characteristics pertinent to the design of a prospective monitoring framework, while also introducing a motivating example. These act as our basis for comparing the various monitoring approaches in Section 3, where we also outline situations where each approach is best suited. Finally, section 4 concludes the paper with directions for future work.

2. SYSTEM CHARACTERISTICS

We consider distributed systems with a set of autonomous, concurrently executing sub-systems communicating through message passing. Each sub-system has (i) its own execution thread, (ii) local memory, and (iii) confidential local information. Most internet-based and service-oriented systems readily fit in the above architecture, as do systems adhering to the Enterprise Service Bus architectures [6]. Characteristics pertinent to this form of architecture affect the design of a prospective monitoring framework, and are discussed below:

Computation and memory distribution: Distributed system architectures entail computation that is distributed across its various computational entities, as well as the partitioning of the system's global state amongst a set of remote partitions. Crucially, the system's global state is not readily available and global state projection is often impractical, due to the voluminous information transfer involved and the restrictions on the communication medium (see next point).

The communication medium: Communication between physically distributed subsystems is considerably slower than local communication, limited by finite bandwidth restrictions. Hence, an efficient distributed system should focus on *minimising remote communication*. Moreover, some communication media may not preserve message order during communication. Other characteristics that one should consider include communication synchrony (synchronous vs asynchronous communication), as well as the potential for non-lossy communication.

*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.

Information locality: Distributed systems may be composed of subsystems each containing confidential local information. This is especially true in heterogenous environments, where issues of *trust* are prevalent. It is the responsibility of any prospective monitoring framework to respect information locality, since failure to do leads to so additional *data exposure*. Data exposure can take the form of (i) exposure through remote communication across unsafe mediums, (ii) exposure of confidential information between non-privileged subsystems.

System topology: This relates to the system's *configuration dynamicity i.e.,* whether the system admits an architecture which 'evolves' during execution. System architecture may evolve in one of two ways; (i) the number of contributing computational entities changes during execution, (ii) the communication pattern between sub-systems changes. Systems which admit dynamic configurations include peer-to-peer systems, as well as service oriented architectures using brokers for service lookup.

2.1 A Motivating Example

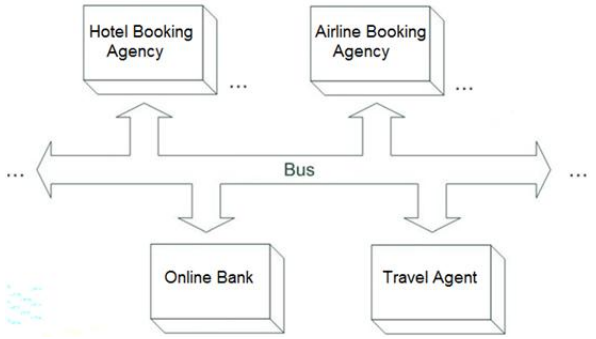

Figure 1: The travel agent.

Figure 1 depicts a typical distributed system whereby a travel agent is responsible for booking holidays on the client's behalf. Given a set of client requests and financial limitations, the agent's task is to search for deals across multiple hotel and airline booking agencies, booking the best deal (through the client's online bank) given the specified restrictions.

We shall be looking at two variants of the above scenario. In the first (simpler) scenario, the travel agent is to communicate with a pre-determined set of online banks, as well as hotel and airline booking agencies. The second scenario involves the agent acting as a broker, dynamically searching (based on parameters such as destination country, budgets etc) for hotel and airline booking agencies according to the client's requests, as well as communicating with the appropriate bank responsible for the client's account. Clearly, the second scenario is more flexible and could potentially return better results (due to dynamic lookup). However, additional capabilities come at the cost of complexity; whereas all contributing entities (i.e. banks and booking agencies) in the first scenario are known a priori, entities involved in the second scenario can only be discovered at runtime. We shall later see how both scenarios affect the applicability of distributed monitoring approaches.

The protocol adhered to by the travel agent while catering for a client request is as follows:

1. The client provides the travel agent with (i) bank account details (for future transactions), and (ii) a set of parameters regarding the desired holiday (destination country, cost etc).

2. Based on this information, the travel agent interacts with (i) the client's online bank, (ii) candidate flight and hotel booking agencies.

3. Agencies are chosen based on some optimal selection policy (possibly effected dynamically based on some service lookup), and queried for a quotation of desired bookings.

4. With the quotations in hand, the travel agent subsequently queries the bank if the client's bank account can afford the provided package. If so, the travel agent returns to the client for confirmation.

5. If confirmed, transactions are triggered on the client's behalf, else the process restarts by choosing different flight and airline booking agencies.

The travel agent example exhibits the characteristics discussed above. Clearly, the system is *distributed*, since both the computational entities (agencies, bank, travel agent) and the system's memory space is partitioned into a set of distributed sub-systems. Moreover, the notion of *information confidentiality* is of considerable importance. Both the online bank (private acccount details, transactions), as well booking agencies (booking information) admit local information whose preservation of locality is paramount. Data exposure can take both forms i.e. exposure of confidential information (such as bank account information) across unreliable mediums (in this case, the internet), as well as exposure across entities (for example, rival booking agencies). Given that inter-system communication happens online, this implies that the global system operates within *restricted bandwidth* limitations. Finally, whereas the first travel agent scenario admits a *static topology* (all entities are known prior to computation), the second scenario involving service lookup admits a *dynamic topology*, since participating entities depend on the client's requests and details.

System correctness in the example above is of critical importance. One property expected to hold is that of *progress i.e.,* each client request submitted to the travel agent is eventually countered by an offer from hotel and flight booking agencies within a certain time frame. Another correctness property involves ensuring that the cost of proposed bookings does not exceed the client's bank balance. Although both properties specify a form of *event sequentiality*, they vary on one subtle point; whereas the former refers to generally non-confidential information (booking offers are usually publicly available online), the latter requires the handling of confidential information (bank details).

3. DISTRIBUTED MONITORING

Monitoring correctness w.r.t the two properties outlined in Section 2.1 is heavily influenced by both the underlying distributed architecture as well as the property's nature. *Computation/memory* distribution forces monitoring

be carried out across unsafe mediums. Moreover, the monitoring framework no longer has ready access to the system's global state, meaning that evaluating properties over the partitioned state is challenging. The *communication medium* may also potentially introduce new problems. It is the responsibility of all prospective monitoring frameworks to minimise bandwidth overhead induced by the framework - failure to do so could interfere with the system's integration effort, possibly altering the system's behaviour. Moreover, lack of communication order preservation could induce monitors remotely observing system behaviour into incorrectly validating broken properties, or vice versa (see [13]). The monitoring framework is also responsible for preserving *information locality* in the presence of confidential local information. Finally, the issue of system configuration dynamicity presents an additional complexity, since this requires the monitoring framework to 'keep up' with the often unpredictable runtime changes the system undergoes during execution.

Presently, there exist a number of architectures and tools for distributed runtime verification and monitoring. These are best understood and categorised according to the following two criteria.

Choreography vs orchestration: Current approaches to distributed monitoring can be broadly classified as *orchestration* or *choreography-based*. In orchestration-based approaches, verification responsibility lies firmly with a central monitor overhearing all information pertinent to the system's global correctness, as seen in figure 2. Although this approach works seamlessly on monolithic systems, its application is not as straightforward in a distributed environment. In the case where the monitored property concerns only public communication between subsystems, this approach works well by constructing a monitor overhearing all such communication, modifying its state accordingly. However, when the system property involves local subsystem information, this approach is less than ideal. Firstly, communication of local confidential information across remote locations leads to data exposure. Moreover, the volume of information required for centralised monitoring is substantial, often resulting in unreasonable bandwidth overhead. Finally, orchestrated approaches pose a security risk by presenting a central point of attack, in the form of the monitor, through which sensitive information can be tapped.

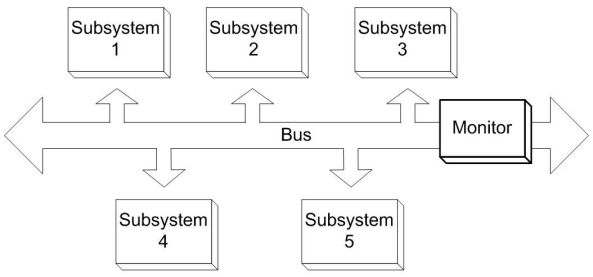

Figure 2: An orchestration-based approach.

Choreography-based monitoring takes a more *dataflow dependent* approach, whereby (sub)system events drive the execution control flow of the monitoring process, often leading to a distribution of monitoring functionality across the distributed system. In general, choreography-based monitoring can mitigate shortcomings posed by orchestrated approaches. A choreography-based approach can be optimised to push verification to occur locally, minimising data exposure. Also, monitoring localisation eliminates the requirement of subsystem information transfer back to a central monitor. This does not stop localised monitors from communicating over the communication medium, however the *volume* of information for monitor synchronisation is usually substantially less than that required by a central monitor, implying that a choreography based approach potentially reduced bandwidth overhead in certain situations. Finally, removing the central monitor eradicates the security risk of providing a central attack point. Nevertheless, although choreography-based approaches boast advantages over orchestrated counterparts, applying choreography is often more complex especially since the nodes in which monitoring is to be set up must somehow enable the instrumentation of monitoring code, and should hence be used only in scenarios where it is advantageous to do so.

Static vs dynamic properties: Static properties include properties whose specification is entirely known at compile time, and remains unchanged during system execution. On the other hand, dynamic properties involve system properties (i) whose characterisation is not entirely known (or changes) at runtime, or (ii) may be learnt entirely during execution — thus making the monitor parametrised by properties which may only available at runtime. One finds dynamic properties, for instance, in security-related intrusion detection scenarios [7], where suspicious user behaviour can only be learnt at runtime after observating the system to learn what typical behaviour looks like. Dynamic properties can also be *contextual* i.e. properties which evolve according to collected information. Taking the travel agent scenario, it is conceivable that different banks would require the verification of different security policies, or that the security policy to be verified depends on the amount of money involved in the transaction (hence, a dynamic property which depends on its context — the bank and the amount of money involved). In such cases, the property can be seen as either a complex conditional static property or a number of simpler properties, only one of which is triggered dynamically at runtime. Although the issue of static vs dynamic properties is not bound to distributed monitoring frameworks, dynamic properties have a particular affinity to distributed systems due to the possibility of dynamic configurations — distributed systems whose configuration evolves during execution may require properties which change accordingly so as to monitor the dynamically changing architecture. Clearly, although dynamic properties are more expressive than their static counterparts, they also represent a class of considerably more complex properties to monitor, and should only be considered when necessary.

These criteria lead to the possibility of four categories for distributed monitoring, namely *static orchestration*, *static*

choreography, *dynamic orchestration* and *dynamic choreography*. The choice of approach often depends on necessity, depending on both underlying system characteristics and the property under consideration.

3.1 Static Orchestration

Conceptually the simplest approach, static orchestration involves employing a central monitor overhearing information over the communication medium, and verifying a set of pre-determined properties. This approach is evidenced in [3], where web service compositions implemented in BPEL [2] are monitored in orchestrated fashion. Advantages with this approach include (i) its simplistic nature, both in concept and usually in application, and (ii) its applicability when monitoring properties dealing with public information over the communication medium. However, static orchestration admits prevalent issues discussed above. Namely, static orchestration may lead to data exposure, poses a security risk, could also potentially result in unreasonable bandwidth overhead and is also incapable of handling dynamic properties (hence, no dynamic configurations).

Nevertheless, a static orchestration-based approach is applicable in certain scenarios. One could, for example, monitor the travel agent (assuming the first scenario involving a static topology) for the progress property, by installing a central monitor which overhears public client requests and offers made by the booking agencies, and verifying that each request is met with a response. However, verifying the second travel agent scenario (admitting a dynamic topology) is unattainable, since static property specifications are incapable of expressing properties over dynamic architectures.

3.2 Static Choreography

Static choreography involves converting system properties at compile time into a set of distributed monitors encompassing the global monitoring framework, as seen in figure 3. These monitors observe system behaviour locally, and synchronise remotely to achieve global verification of the system. Current static choreography based approaches include [13, 12, 9, 10, 14].

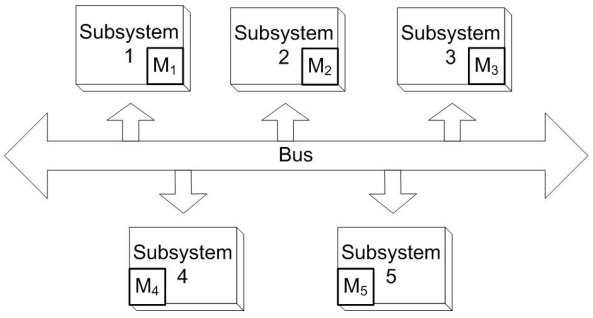

Figure 3: A static choreography-based approach.

Advantages with static choreography include those discussed for choreography in general, *i.e.* (i) preservation of locality, (ii) possible bandwidth overhead reduction, and (iii) the removal security risks related to central attack points.

If we had to monitor the first travel agent scenario for the property stating that bookings do not exceed the client's financial limitations, one could employ local monitors (one at each booking agency and online bank), with monitors at the booking agencies notifying the monitor located at the client's particular bank of proposed bookings, which can then verify locally that cost does not exceed limitations. Notice how using this approach no confidential client information leaves the bank's location, as opposed to a static orchestrated approach which would require transfer of client information remotely to the central monitor. The volume of information transfer for monitoring purposes is also decreased, since monitor synchronisation upon booking generation involves less information than the transfer of relevant bank, hotel and flight booking information to a central monitor.

Given that monitor distribution occurs once a priori to system execution, a static choreography approach is incapable of handling evolving system properties, or properties learnt at runtime. Handling dynamic properties through a static choreography based approach would require (i) recompilation, and (ii) re-distribution of the monitoring framework upon each update the the monitored properties, which is generally unfeasible. A direct implication of this statement is that static choreography is incapable of handling dynamic topologies, since one would require dynamic properties capable of quantifying over evolving system architectures. This implies that monitoring the second travel agent scenario is unattainable using static choreography.

Finally, note how the handling of the progress property discussed for static orchestration is also attainable using static choreography, by matching a client request at the travel agent with booking responses at hotel and flight booking agencies. However, no apparent advantage is gained by applying static choreography over static orchestration in this scenario (information locality is a non issue when dealing with public communication over the bus), making the application of choreography an unnecessary complication.

3.3 Dynamic Orchestration

Dynamic orchestration-based approaches involve the adoption of a central monitor remotely observing sub-system behaviour, which however allows for the monitoring of dynamic properties. An instance of dynamic orchestration is seen in [1], involving the centralised monitoring of web services through the specification of BPMN work flows [5]. Moreover, this approach allows for the deployment of the verification of contracts (representing system properties) *on-the-fly*, allowing for the verification of dynamic properties.

The main advantage of dynamic orchestration over its static counterpart is the capability to handle dynamic properties. A dynamic orchestrated approach may for example monitor the second travel agent scenario for both previously discussed properties, with the central monitor listening to information from new agencies discovered by the travel agent accordingly. Nevertheless, dynamic orchestration still suffers from data exposure, is potentially inefficient due to unreasonable bandwidth overhead, and still represents a security risk by presenting a privileged entity, in the form of the monitor, through which information can be tapped.

Although possible, a dynamic orchestrated approach for monitoring the second travel agent scenario is hence unsuitable, since using such an approach would still require remote transfer of sensitive client bank details to the central monitor. A dynamic orchestrated approach appears to fit best when only public information needs to be monitored over the communication medium w.r.t. some dynamic properties.

3.4 Dynamic Choreography

Similarly to static choreography, dynamic choreography entails the distribution of monitoring functionality across the distributed system. However, with dynamic choreography distribution occurs *during execution*, hence allowing for the re-distribution of the monitoring framework allowing for the monitoring of dynamic properties. To the best of our knowledge, presently no existing monitoring architecture falls in this category.

To this effect, we propose the study of dynamic choreography through the use of *migrating monitors* i.e. monitors running locally to sub-systems, and physically migrating to other locations when requiring to monitor local behaviour pertinent to the global correctness of the system, as depicted in Figure 4.

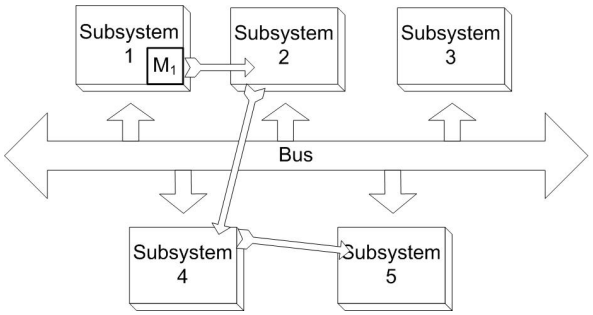

Figure 4: A migrating monitor approach.

Migrating monitors take a *property agnostic* approach, such that each sub-system is instrumented to expose an alphabet of information available to the monitors exclusively at a local level. Monitors are subsequently allowed to migrate to sub-system locations, locally reading information pertinent to the property under consideration. This approach allows for the monitoring of properties learnt at runtime, since new properties can be converted to corresponding migrating monitors, and are subsequently executed *on-the-fly* without the need for system restart. Dynamic choreography is achieved through a conceptually simple operator triggering monitor migration. This operator allows for the runtime framework redistribution, since monitors can be specified to migrate to alternate locations during execution, possibly even locations not known at the start of computation. Migrating monitors are hence capable of handling dynamic configurations, by migrating to new locations once integrated with the global architecture. This, in turn, is a weakness of dynamic choreography approach, since the nodes need to be willing to install monitors known only at runtime. This leads to a question of trust, since the monitor may originate from an untrustworthy party — on the other hand, one can require monitors to be signed by trustworthy partners, or even apply static analysis techniques or require the monitor to appear as proof-carrying code to make the approach practical.

A migrating monitor technique is advantageous in three respects

1. it admits advantages pertaining to choreography based approaches — see Section 3.2. Being a choreography based approach, migrating monitors allow for the preservation of locality while also potentially reducing bandwidth overhead in certain situations.

2. it increases *flexibility*, *i.e.,* the capability of the monitoring framework to adapt to unforeseeable (at compile time) changes during system execution, at little additional complexity. This is achieved by adding the migration primitive. As discussed, changes during system execution can take the form of dynamic system configurations, as well dynamic properties (including properties learnt at runtime and properties which evolve during execution).

3. it achieves elevated *encapsulation*. Migrating monitors operate at a higher level of abstraction, achieving choreography while binding together computational entities and information (monitor state) whose purpose is that of achieving one common goal (monitoring of a property). Conceptually, this is a sharp distinction from static choreography-based approaches, since monitor computation and state operating to verify a particular property are often distributed throughout the system. Note that this does not stop the migrating monitor approach from employing multiple concurrent monitors. Elevated encapsulation also points to migrating monitors being potentially more amenable to *fault tolerance techniques*, since migrating monitors are intuitively reconfigurable on-the-fly. This is a desirable property in the presence of *partial failure* inherent in distributed systems. Hence, one could for example alter a monitor's migratory patterns at runtime once a contributing system entity is deemed to be unavailable. Another by-product of elevated encapsulation is potentially easier *maintenance* (as opposed to maintaining a static choreography based approach), since updating a framework employing migrating monitors would only require update of the migrating monitors, usually far less in number than the amount of updates involved when updating a static choreography based framework (one monitor is assigned to each entity).

Let us consider the monitoring of the second travel agent scenario for the second property that the cost of bookings does not exceed the client's bank balance, using a migrating monitor approach. One could define a migrating monitor, whose state entails (i) the client's online bank information, (ii) hotel and flight booking agency information under consideration, (iii) a counter recording the cost of proposed bookings. Execution of the migrating monitor would start at the travel agent location, collecting information regarding contributing entities during service of the client's request (bank, booking agencies). Using this information, the migrating monitor can then dynamically set location information where the monitor is to migrate to, starting with migrating to contributing booking agencies. At each booking agency, the monitor collects the cost of the particular agency's suggested booking (cost is accumulatively stored using the monitor's counter). Once all booking costs are collected, the monitor finally migrates to the online bank, and using the client's online bank information checks whether the client can afford the suggested bookings.

Note how information locality is preserved while probably resulting in bandwidth overhead reductions (as opposed to orchestrated approaches), since globally monitoring the property has been reduced to local monitoring interspersed with a few monitor migrations. Verification of dynamic properties have also been trivialized through the use of the

migration operator in conjunction with a property agnostic approach. New booking agencies (discovered by the travel agent through service lookup) are immediately monitored by allowing for the monitor to migrate at the new locations.

A migrating monitor approach also admits disadvantages which make its application a sub-optimal choice in certain situations. Firstly, the approach is best suited when the property under consideration admits a migrating monitor characterisation based on substantial local monitoring and few remote migrations. Conversely, properties requiring migrating monitors based on an unreasonable amount of migration may result in substantial bandwidth overhead. The migrating monitor approach is also based on the inherent assumption that multiple (often heterogenous) subsystems *trust* code entities to migrate and run locally, while having access to sensitive local information. Clearly, this may not always be the case, since system administrators may disapprove of external executable entities having privileged access to their systems. Finally, although the property agnostic approach is conceptually advantageous for reasons discussed above, it is a more complex instrumentation approach than that applied in more traditional runtime verification techniques (such as in [11, 4, 13]), which could result in higher probability of failure.

For instance, although verification of both travel agent scenarios for adherence to the first property is also possible using migrating monitor approach, it is not advantageous to do so. Given that the first travel agent scenario admits a static topology it would be wasteful to apply an approach focused on handling dynamic properties, implying that an approach handling static properties would suffice. Moreover, the first property involves public information transferred over the communication medium, implying that information locality is a non issue. Hence, an orchestrated approach would suffice.

In conclusion, migrating monitors offer an alternate approach to distributed monitoring, whose applicability is best when facing strict information confidentiality restrictions in a highly dynamic environment (either in terms of the property being verified, or the underlying system admitting dynamic configurations).

4. CONCLUSIONS AND FUTURE WORK

Runtime monitoring of distributed systems is considerably more complex as opposed to monolithic local system. In this paper, we show how this is heavily influenced by characteristics intrinsic to distributed systems. We also propose what we believe to be a novel migrating monitor approach, advocating the use of location-aware monitors listening to events exclusively at a local level, before migrating to other locations when their behaviour becomes pertinent to the system's overall correctness. We argue that this approach respects information locality and handles dynamic properties (including systems admitting dynamic architectures).

We are currently investigating formal properties of this technique such as (i) monitoring does not effect computation, (ii) local monitoring preserves locality, (ii) local monitoring is equivalent to global monitoring, ignoring location information. Proof of these three framework properties should serve as sanity checks to ascertain applicability of the approach. We are also looking into the extension of LARVA [11] to handle migrating monitors so as to be applied to case studies using Enterprise Service Bus middleware [6].

5. REFERENCES

[1] Charlie Abela, Aaron Calafato, and Gordon J. Pace. Extending wise with contract management. In *WICT 2010*.

[2] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006.

[3] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. pages 44–57. Springer, 2004.

[5] Marco Brambilla, Stefano Ceri, Piero Fraternali, and Ioana Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15:360–409, 2006.

[6] David Chappell. *Enterprise Service Bus*. O'Reilly Media, June 2004.

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

[8] Edsger W. Dijkstra. Notes on Structured Programming. April 1970.

[9] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Interaction-based runtime verification for systems of systems integration. *Computer Science and Engineering Department, University of California, San Diego USA*, July 2008.

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

[11] Gordon Pace, Christian Colombo, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, LNCS 4916. Springer-Verlag, 2008.

[12] Thomas S.Cook, Doron Drusinksy, and Man-Tak Shing. Specification, validation and run-time moniroting of soa based system-of systems temporal behaviors. In *In System of Systems Engineering (SoSE)*. IEEE Computer Society, 2007.

[13] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. *Software Engineering, International Conference on*, 0:418–427, 2004.

[14] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo, and Insup Lee. Dmac: Distributed monitoring and checking. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 184–201. Springer, 2009.