Investigating Instrumentation Techniques for ESB Runtime Verification*

Christian Colombo[1], Gabriel Dimech[2], and Adrian Francalanza[1]

[1] Department of Computer Science, University of Malta
[2] Ricston Ltd., Malta
christian.colombo@um.edu.mt, gabriel.dimech@ricston.com,
adrian.francalanza@um.edu.mt

Abstract. Enterprise Service Buses (ESBs) are *highly-dynamic* component platforms that are hard to test for correctness because their connected components may not necessarily be present prior to deployment. Runtime Verification (RV) is a potential solution towards ascertaining correctness of an ESB, by checking the ESB's execution *at runtime*, and detecting any deviations from the expected behaviour. A crucial aspect impinging upon the feasibility of this verification approach is the *runtime overheads* introduced, which may have adverse effects on the execution of the ESB system being monitored. In turn, one factor that bears a major effect on such overheads is the *instrumentation mechanism* adopted by the RV setup. In this paper we identify three likely (but substantially different) ESB instrumentation mechanisms, detail their implementation over a widely-used ESB platform, assess them qualitatively, and empirically evaluate the runtime overheads introduced by these mechanisms.

1 Introduction

Enterprise Service Buses (ESBs) [4] are software platforms used to streamline the communication across various components within an enterprise, ranging from legacy systems, locally housed databases, third-party off-the-shelf applications, to cloud services. They abstract away from complications associated with differing communication protocols/data formats and component distributions, by providing a message-oriented middleware that handles the necessary format translations and message routing. This enables the enterprise administrator to organise these components into a Service-Oriented Architecture (SOA) [17, 15] where one can focus on the enterprise application logic.

Despite their merits, component systems running on ESBs are still hard to build correctly. ESBs are inherently *dynamic* so as to handle the changing needs of an enterprise over an uninterrupted period of operation. For instance, new components may be added at runtime, others may be disconnected or replaced, or even duplicated so as to improve aspects such as throughput and fault-tolerance. This dynamicity has a ripple effect on the internal workings of the *resp.* middleware, whereby the destination of messages may need to be determined at runtime. Moreover, ESB communication is

* The research work disclosed in this publication is partially funded by the *Master it!* Scholarship Scheme (Malta).

intrinsically *asynchronous* so as to make the architecture scalable; this means that one has also to contend with message reordering, which introduces another layer of unpredictability. These aspects substantially diminish the effectiveness of commonly used pre-deployment techniques for ascertaining correctness, *e.g.*, testing and static analysis.

Runtime Verification (RV) [11] is a technique that allows a system to be verified *post-deployment*: using software entities called monitors, it analyses system runtime events and checks whether they adhere to (or violate) a predefined correctness specification. Since checks are performed at runtime, RV may potentially use information that is not necessarily known before execution commences, such as the components currently connected to the ESB, the *resp.* execution interleaving of these (concurrent) components, and the order of messages received (together with their *resp.* payloads). This allows the analysis to be more precise and tractable, thereby circumventing the limitations of the pre-deployment verification techniques discussed above.

A determining criteria for whether such a verification technique is feasible in practice is the level of*runtime overheads* induced by the RV setup, which should be kept below some acceptable threshold. More specifically, the runtime checks carried out by the monitors, together with the additional machinery required to extract and report the system events of interest, has a computational burden on the execution of the system being analysed in terms of the additional computational resources required; this typically results in performance degradation of the system itself, once it starts competing with the RV setup over scarce resources.

Instrumentation — the mechanism by which the monitors are hooked on to the system so as to extract events and analyse them — is a fundamental ingredient of any RV setup, affecting aspects such as the observability of system events and the maintainability of the setup. Importantly, instrumentation can usually be introduced in a variety of ways, each carrying varying effects on the level of overheads induced. The choice of an appropriate instrumentation strategy for RV does not, however, depend solely on the induced overheads. In this paper, we focus on the following instrumentation criteria:

Efficiency [16] Instrumenting software naturally introduces an overhead in two respects: the amount of extra resources used, and the extent to which this impacts the user experience, *e.g.*, longer response time. Although resource consumption, by itself, may not lead directly to service degradation (namely due to the availability of excess resources), it is generally still interesting to measure resource consumption for the eventuality of an increase of load on the system. *In the context of RV, instrumentation efficiency affects directly runtime overheads of the technique.*

Level of abstraction [12, 13] Choosing the right level of abstraction dictates how understandable and easy it is to express the instrumentation points of interest. In general, lower abstraction levels give better access to the internals of the system, at the expense of usability and maintainability, since it requires an understanding of how the system works at that level of abstraction. *Since RV relies on the user to specify points of interest, selecting a level of abstraction familiar to the user affects the usability of the RV framework.*

Expressiveness [13] (flexibility in [12]) The instrumentation (join)points available — sub-classified into *structural* (*e.g.*, a method or a class variable) and *temporal* (*e.g.*, before the method is called or after the method returns) — affect what can be ob-

served. In the case of layered architectures such as ESBs, this is (in part) linked to the abstraction level chosen, since the points of interest available at a lower level may have no corresponding point of interest at a higher level. *Expressiveness may inhibit the possibility of certain checks being carried out at runtime.*

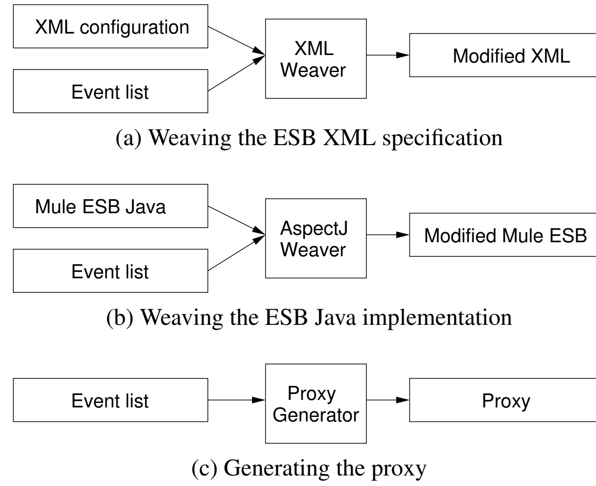
Coupling (combining/extending flexibility [16] and portability [12]) The level of coupling refers to the bond between the weaving mechanism and the system being weaved. This has implications on maintainability (*e.g.*, whether the weaving mechanism can be changed without affecting the system, or whether the system requires recompilation upon instrumentation modification) and reusability (*e.g.*, whether the same weaving approach can be applied to other systems). *In dynamic RV settings, this affects whether correctness criteria may be feasibly altered at runtime.*

Finding the right balance across these criteria is not an easy task as some of them are in direct conflict. In the context of this work targetting RV setups for ESBs, we assume that efficiency is given a higher priority than the other criteria and it is down to the RV specifier to leverage a balance between the other criteria so as to attain adequate levels of overhead. In Section 2 we present three RV instrumentation alternatives for ESBs and evaluate their *resp.* advantages *wrt.* the software instrumentation criteria above. Through a series of empirical investigations over a typical ESB case study, in Section 3 we analyse the *resp.* runtime overheads introduced by each of the instrumentation methods identified in Section 2, and compare gains and losses *wrt.* the other instrumentation criteria. Our findings may thus be used as a guiding principle by an RV instrumentor when leveraging a feasible RV setup over any ESB setting.

2 Design

There are various ESB solutions used in industry [2, 18, 20, 9] all sharing similar architectures and core concepts [4]. Our study focusses on one of the open source solutions, namely Mule ESB, with a considerable market share (*i.e.*, 16% [8]). Mule ESB is organised over three layers: (1) a domain-specific language (DSL) layer (*configuration layer*), which allows users to connect remote components via (XML) configuration scripts [7]; (2) a *Java source code layer* which is compiled from the XML specification, and thirdly, (3) a *layer of protocols* (e.g., HTTP) over which components communicate. These layers present various options for choosing an instrumentation strategy. In this study, we consider one strategy per layer, and evaluate them against the backdrop of the criteria presented in the Introduction.

Configuration layer strategy. This is the level of abstraction a typical Mule user is accustomed to, since the events exposed are the high level events expressed within the application specification. These are then instrumented as shown in Figure 1(a) by a custom XML weaver, generating an ESB configuration which, once compiled, intercepts relevant events and relays them to a dedicated verifier component (see Figure 2(left)).
This strategy induces a high level of coupling: modifying the application configuration (which is expected to happen regularly, *e.g.*, update the components connected or adding new features to the ESB) requires re-weaving, *i.e.*, the process shown in Figure 1(a) would have to be repeated.



(a) Weaving the ESB XML specification

(b) Weaving the ESB Java implementation

(c) Generating the proxy

Fig. 1: The three approaches of generating weavers

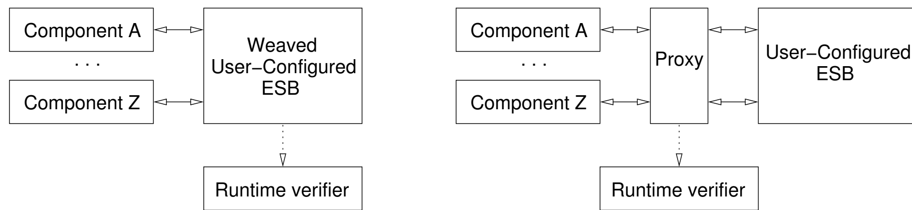


Fig. 2: Event interception through XML and Java weaving (left), proxy (right)

Source layer strategy. The Mule ESB is implemented in Java and this strategy exposes all the internal workings of the ESB. As a result, it provides the highest level of expressivity amongst the three approaches. The flip-side of this is that the RV specifier requires knowledge of how the ESB is implemented at source level, something a regular ESB user may not have. An important consideration is that this instrumentation strategy is not affected by changes in the application configuration (*i.e.*, the instrumentation process depicted in Figure 1(b) does not involve XML configurations); rather, instrumentation is affected by software updates to the ESB implementation. However, ESB software updates occur less frequently than application reconfigurations.

Protocol layer strategy. This strategy intercepts system events as communication messages on the bus, and thus sits at a higher abstraction level than the other approaches[3]. Instrumentation can even be implemented as a proxy (see Figure 1(c)), leaving the application configuration or the ESB implementation unaffected, thus requiring the lowest level of coupling. The price paid for this autonomy is expressivity: limiting intercep-

[3] As the communication mechanism is itself layered, there are a number of levels of abstraction possible. However, we choose to work at the most abstract level possible, *i.e.*, Mule messages.

	Abstraction	Expressivity	Low coupling
Configuration Layer	∼	∼	↓
Source Layer	↓	↑	∼
Protocol Layer	↑	↓	↑

Table 1: Comparing strategies: good (↑), bad (↓), and in-between (∼).

tions to communication messages means that internal states/events may not be visible from this abstraction level. From the specifier's perspective, identifying events of interest is similar to the configuration layer strategy where one specifies inbound/outbound endpoints whose messages should be reported to the monitor. The only difference is that internal component events are not available from this external perspective. To avoid programming the proxy manually, we chose to automatically generate a proxy from the specified events (as shown in Figure 1(c)) which is able to intercept and relay relevant events at runtime (see Figure 2(right)).

Table 1 summarises the characteristics of the three strategies thus far. Each strategy has its strengths and weaknesses, reflecting the trade-offs discussed above. Note that we do not give a verdict on the efficiency aspect of the *resp.* instrumentation techniques at this stage; this is investigated in more depth in the next section.

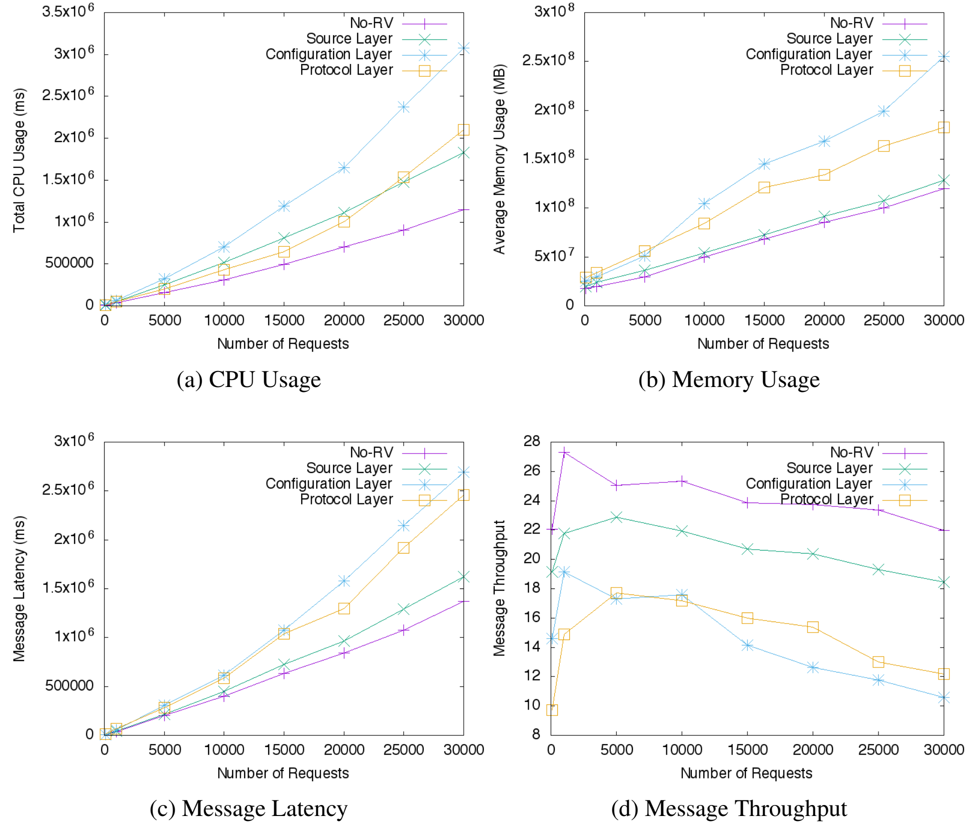
3 Performance Evaluation

The instrumentation strategies of Section 2 exhibit different characteristics that affect the type of properties monitored and their *resp.* ease of monitoring. For instance, certain properties cannot be monitored at certain levels of abstraction, whereas a verification technology that is *not* Java-based would disadvantage instrumentation strategies that are close to the Mule source implementation. In what follows, we normalise these differences (*e.g.*, by limiting our experiments to properties referring to events expressible in *any* instrumentation strategy) and focus on various quantitative measures for assessing the overheads introduced by each strategy.

3.1 Case Study

We employ a third-party, medium-sized ESB application[4] that was not purposely built with our experiments in mind, but for which various loads could be applied. The application listens for changes made in the 'Opportunities' table within a cloud-based service (Salesforce Customer Relationship Management (CRM) [6]). Each time this table is updated, the ESB is notified and triggers a request to retrieve the full details of the update via a web service request. Subsequently, the ESB transforms the received data into canonical format and routes the message based on its content: If the opportunity is 'Won', the message is routed to an external Business Process Management (BPM) application, Activity Workflow Engine, which will allow the account manager to approve

[4] `https://github.com/jdeoliveira/esb-bpm-example`



(a) CPU Usage

(b) Memory Usage

(c) Message Latency

(d) Message Throughput

Fig. 3: ESB performance metrics

the opportunity. If the opportunity is 'Lost', then the BPM process is not invoked but persisted either in database or file system depending on the postal address.

The expected (correct) behaviour described above was formalised in terms of a finite state machine and we used RV to ensure that the ESB behaviour complies to the specification, *e.g.*, ensuring that a 'Won' opportunity is followed by a corresponding valid message. To this end, first, we used message timestamps to determine the order in which messages were sent out. Secondly, we used message contents to identify the kind of message and verify that it is handled accordingly.

3.2 Results

Runtime overheads introduced by RV tools are typically measured in terms of the additional CPU and memory used by the *resp.* monitored system; see [1]. CPU and memory usage statistics for varying request loads are reported in the top row of Figure 3, for both the base-line (unmonitored system) and the *resp.* instrumentation strategies presented

in Section 2. Whilst CPU and memory usage trends give insight into the performance of the system, we also calculated overheads in terms of message latency and message throughput as these are more indicative of the service deterioration experienced by the end users in asynchronous, message-based component systems. The results are reported in the bottom row of Figure 3.

Every measure attests that ESB Source Layer instrumentation yields the lowest overheads. Whereas, in the case of CPU usage, these overheads seem marginally better than the Protocol Layer instrumentation, a major discrepancy can be observed between the Source Layer instrumentation and the other strategies in the case of memory consumption; in fact, the memory consumed by the former instrumentation is remarkably close to that of the baseline. These results are also confirmed by the bottom graphs of Figure 3: for both message latency and throughput there is a pronounced discrepancy between Source Layer instrumentation overheads and the overheads introduced by other instrumentation strategies. We also note, however, that in every case the Protocol Layer instrumentation performs better than the Configuration Layer instrumentation.

Although Source Layer instrumentation yields the lowest overheads, it is by no means a silver bullet. As summarised in Table 1, adequate installation requires sufficient knowledge of the source-level ESB implementation; this level of expertise cannot usually be expected from ESB administrators, who are mainly concerned with operations at the business logic tier and thus mainly operate in terms of XML configuration scripts. Source Layer instrumentation may also pose maintenance problems when new Mule implementation updates are installed, which may affect the definition of the correctness specifications being monitored for that rely on state variables and methods from the previous implementation; in Table 1 this is summarised as medium coupling. In cases where the specifier does not possess adequate knowledge of the ESB implementation internals, and the properties to be monitored for can be suitably expressed, Protocol Layer instrumentation may constitute a good compromise amongst the various strategies. Although the overheads introduced are higher, its low coupling also means that the system instrumented with the *resp.* RV setup would be easier to maintain.

4 Conclusion

We have identified and studied three potential instrumentation strategies that may be adopted when setting up an RV framework over ESBs. Our contributions are:

- We provide a proof-of-concept implementation for each instrumentation method over Mule [7], an industry-strength ESB distribution.
- We evaluate each instrumentation method *wrt.* a series of criteria typically applied to evaluate software instrumentation (see Table 1).
- We asses the overheads introduced by each method, in terms of system performance, Figure 3.

Related Work: We are aware of two main bodies of work which apply runtime verification to ESBs. Psiuk *et al.* [17] propose an RV framework for ESB systems implemented using the JBI specification (*e.g.*, ServiceMix and OpenESB) using AspectJ as instrumentation, while Kruger *et al.* [10] apply runtime verification to a Mule ESB using

Spring AOP. In both cases, the focus was not performance or the choice of the instrumentation approach but rather the design of the architecture, from the specification of the properties, to monitor synthesis, to instrumentation, and effective monitoring.

Future Work: Due to the inherent nature of ESBs, the instrumentation used in our study is *asynchronous* [14, 19, 3], where the execution of the individual ESB components generating the events is independent to that of the monitor. Although asynchronous monitoring yields lower overheads than its synchronous counterpart [3], it may result in late detections. It is worth investigating the applicability of hybrid techniques such as [5, 3] over ESBs so as to attain timely detections. Independently to this, it is worthwhile verifying whether the results obtained in our study can be replicated over (*i*) other ESB implementations other than Mule (*ii*) other ESB case-studies.

References

1. 1st international competition of software for runtime verification. `http://rv2014.imag.fr/monitoring-competition` (2014)
2. Barnett, M., Schulte, W.: Spying on Components: A runtime Verification Technique. In: SAVCBS. pp. 7–13. OOPSLA (2001)
3. Cassar, I., Francalanza, A.: On Synchronous and Asynchronous Monitor Instrumentation for Actor-based Systems. In: FOCLASA. EPTCS, vol. 175, pp. 54–68 (2014)
4. Chappell, D.A.: Enterprise Service Bus: Theory in Practice. O'Reilly Media (2004)
5. Colombo, C., Pace, G.J.: Fast-forward runtime monitoring - an industrial case study. In: RV. LNCS, vol. 7687, pp. 214–228. Springer (2012)
6. Cusumano, M.: Cloud computing and SaaS as new computing platforms. Communications of the ACM 53(4), 27–29 (2010)
7. Dossot, David D'Emic, J., Romero, V.: Mule in Action. Manning Publications Co. (2014)
8. Gopal, J., more: Guide To Enterprise Integration. http://www.dzone.com/research/guide-to-enterprise-integration (2014)
9. Ibsen, C., Anstey, J.: Camel in Action. Manning Publications Co. (2010)
10. Krüger, I.H., Meisinger, M., Menarini, M.: Interaction-Based Runtime Verification for Systems of Systems Integration. J. Log. Comput. 20(3), 725–742 (2010)
11. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. JLAP 78(5), 293–303 (2009)
12. Mahrenholz, D., Spinczyk, O., Schroder-Preikschat, W.: Program instrumentation for debugging and monitoring with AspectC++. In: ISORC. pp. 249–256 (2002)
13. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: Disl: a domain-specific language for bytecode instrumentation. In: AOSD. pp. 239–250. ACM (2012)
14. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. STTT (2011)
15. Papazoglou, M., van den Heuvel, W.J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. VLDB 16(3), 389–415 (2007)
16. Popovici, A., Alonso, G., Gross, T.: Just-in-time aspects: Efficient dynamic weaving for java. In: AOSD. pp. 100–109. ACM (2003)
17. Psiuk, M., Bujok, T., Zielinski, K.: Enterprise Service Bus Monitoring Framework for SOA Systems. IEEE Transactions on Services Computing 5(3), 450–466 (2012)
18. Rademakers, T., Dirksen, J.: Open-Source ESBs in Action. Manning Publications Co. (2008)
19. Roşu, G., Havelund, K.: Rewriting-Based Techniques for Runtime Verification. ASE 12(2), 151–197 (2005)
20. Siriwardena, P.: Enterprise Integration with WSO2 ESB. Packt Publishing Ltd (2013)