Applying Runtime Verification Techniques to an Enterprise Service Bus

Gabriel Dimech
Dept. of Computer Science
University of Malta
gabriel.dimech.06@um.edu.mt

Christian Colombo
Dept. of Computer Science
University of Malta
christian.colombo@um.edu.mt

Adrian Francalanza
Dept. of Computer Science
University of Malta
adrian.francalanza@um.edu.mt

***Abstract*—An Enterprise Service Bus (ESB) integrates remote components creating a distributed system from a centralised location where certain aspects of the system are dynamic in nature. These characteristics give rise to potential runtime issues arising during the deployment of ESB applications. In this paper we describe why some of these issues may not be addressed at compile time; as current ESB solutions go a long way in providing the right tools to setup integration tests which allow for testing the integration logic, however are unable to guarantee correctness beyond the scope of such tests. Due to the characteristics of ESB applications, we discuss applying Runtime Verification (RV) techniques in three separate approaches over an ESB with the goal of giving a correctness guarantee for problems undetectable at compile-time with the aim of minimising performance impacts inhibited on the ESB system.**

I. Introduction

As software becomes increasingly dependent on external resources such as APIs, data sources and applications to provide more overall value, integrating applications has become increasingly important. Enterprises are now leveraging a range of applications including legacy systems, on-premise applications and cloud services both internal or external to the organisation. As such enterprises evolve, so do their requirements; producing a dynamic system of systems which may not have been designed to communicate with each other, resulting in integration challenges.

A. Background

In order to highlight some issues that may arise when deploying ESB applications, we will take a closer look at the architecture of an ESB and describe an example application.

1) Enterprise Service Bus: An ESB is a tool which integrates systems in a bus-like architecture where the bus acts as a point of reference to the overall distributed or localized system [1]. Within the ESB, components may communicate via a standard protocol. Having a centralised location from which to manage the orchestrated service, allows the user to maintain the system relatively easily when compared to say point-to-point systems where connections are made between components.

2) ESB Example Application: As an example of an ESB application, consider an online flight booking system; were third party airline services and a banking service are orchestrated to provide a flight booking application. A flight booking component may only confirm the booking once the bank component has verified that the payment details and the airline component confirms that the dates specified are permissible. Let us assume that the integration logic allows for communication with just one banking system in order to retrieve banking details, participate in financial transactions and so on. The same application allows for communication with one or more airline booking systems. This will allow the user to get more competitive flight rates and also provide a backup system in the case that one airline web service is down. It is most likely that both the banking system and airline systems are exposed as a service where we are only able to see these systems as a black box. This is depicted in Figure 1 which shows how the flight booking application described above may leverage the banking system and a number of airline services in order to expose a flight booking service to customers. The ESB allows for seamless communication between these components.

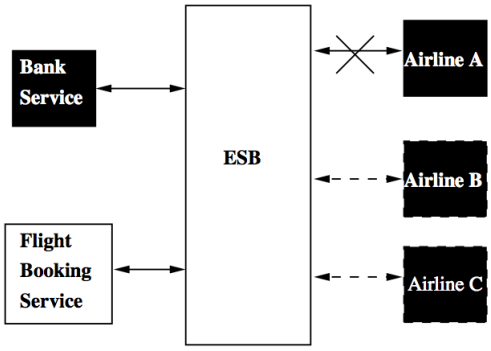

Fig. 1. Example ESB Application

B. Possible Issues affecting Correctness

Having described briefly some characteristics of ESB applications in general as well as the expected behaviour of an example application above, we shall now look at some potential issues that may affect the correct execution of ESB applications. Correctness of an ESB application here relates to the application behaving as intended during execution. For example, this may include that for a given event in the ESB, the application logic correctly transforms messages between protocols, routes via the intended path to and from destinations and in general the overall behaviour of the application is as intended by the ESB application architect.

1) Dynamic Sources and Destinations: Some ESB implementations provide a feature for service selection within

the integration logic [3]. This allows for deciding message destinations at run-time where for example the HTTP path or port number on which to send an HTTP request, is obtained from the message currently being handled. For such a scenario, it is not possible to test all possible message destinations. This is depicted in Figure 1 where a connection to Airline A has been lost at run-time (denoted by a cross over a solid line), therefore we may forward the request to a backup airline service (In Figure 1 connections to backup services are denoted with dotted lines). An ESB application may be invoked via an event received from a messaging queue data source where the message producers may not necessarily be known at compile time, hence it is not possible to test fully all possible message types for such a scenario.

2) Third party Components and Libraries: Testing the integration logic is usually done via mocking third party components during the testing phase. This is very effective for providing correctness of the ESB application logic whose source code we have access to. By mocking the third party components, in this case the banking and airline systems, we are able to create "failure" scenarios and modify the behaviour of the ESB accordingly to handle these issues. The test engineer may create test cases for a number of scenarios so as to verify that the ESB application logic is able to handle the "happy" path/s and most of the "unhappy" paths. Abstracting away implementation and communication details from the application logic, relieves the ESB application developer from many potential mistakes in the application. That said, integrating complex remote systems such as the airline and banking systems, renders the process of writing comprehensive integration tests a laborious and perhaps impossible task. When adding such remote systems, complexity increases exponentially, so too does the risk of errors occurring. This is also known as the state explosion problem in model checking rendering this technique unfeasible for such applications. Also, third party components, such as the airline and bank services, may be updated from time to time. If this happens without the corresponding updates being included in the existing version of the application, this may cause unexpected behaviour in the application.

Besides communicating with third party components as explained above, ESBs include a large number of third party libraries which are required to enable key features such as transformation, consuming/exposing web services, clustering, data source connections and so on. This means that whilst an ESB application is running, there is also the likelihood that one of these third party libraries (sometimes these may be closed source) may fail (for example memory leaks).

Figure 1 depicts the flight booking system example where we do have access to the code inside the flight booking application as well as the integration logic (denoted by white boxes in Figure 1), allowing developers to tweak the functionality if required. The banking and airline components are seen as black boxes (denoted by black boxes in Figure 1) and are only visible to the ESB via a connection (for example via HTTP), therefore we can only assume that these components will function as expected.

3) Complex Integration Logic: A source of error for ESB applications may arise from bugs in the integration logic done by the system architect/engineer which go unnoticed during the testing phase. As an example of such a scenario, consider the flight booking system described earlier: The flight booking application returns confirmation to the user without the airline component having returned confirmation that the dates requested are available. This is not the correct behaviour of the application, even though the user has received confirmation.

4) Distributed Transactions: Typical ESB applications handle a high throughput of sensitive information which may require to take part in some form of transaction. More often than not, such a transaction may require participation of various distributed resources such as databases and message queues. The transaction manager which is in charge of overseeing the success or rollback of a transaction makes use of the two-phase commit protocol for such a scenario. This process includes a 'prepare' phase where participating resources are asked to prepare for committing to the transaction. If not all the participating resources return success for this prepare phase, the transaction is rolled back, otherwise committed. In the event that an error occurs after the participating resources have successfully responded to the prepare phase, then the participating resources risk entering an inconsistent state as the transaction will be committed on some of the participating resources. This is known as the "Two generals problem" whereby consistency of all participating resources is not guaranteed [4].

The possible issues affecting correctness of ESB applications given here is by no means an exhaustive list. Such issues are best identified on a per application basis, as each application has a separate configuration in terms of routing logic, data transformation, data resource connections, custom code within the ESB and so on.

II. Why Runtime Verification?

The scenarios and examples given above, for instance that of dynamic destinations, may be tested during the testing phase against a number of mocked destinations for any issues in the integration logic, however this will not provide correctness guarantees with message destinations decided at runtime. Similarly, when testing integration with third party components as mocked services, testing techniques are unable to provide correctness guarantees in the event that say a remote component returns incorrect data without an error being raised in the ESB logic.

With the main aim of providing correctness guarantees to ESB applications and the above considerations in mind, we propose using testing techniques for scenarios where runtime information is not required and complement this using runtime verification techniques for scenarios where runtime information is required.

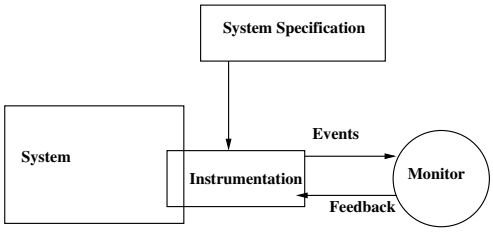

Fig. 2. Runtime Verification Process

Runtime Verification (RV) is a technique used for monitoring and analysis of software system runs [5], [6], [7], allowing us to detect violations in system behaviour requiring runtime information which is not possible using testing techniques. At the same time, it circumvents state explosion problems associated with model checking techniques by compromising the extent of checking, limiting itself to ensuring only that the current system execution is valid. It has proved to be scalable enough to be applied to numerous real-world systems, providing feasible ways how to leverage formal techniques originating in academia. RV promises to be a suitable technique for checking correctness in ESB systems because it can be performed at runtime, while the ESB is executing. This allows it to obtain runtime information such as the current components connected to the bus or the current thread interleaving of components when checking for correctness, instead of having to predict them upfront, which in turn rules out potential execution states that would otherwise need to be considered.

Figure 2 depicts the process of applying runtime verification to a system where a user produces a specification which is used to specify correctness properties and the instrumentation code which will be deployed in parallel with the running target system. The instrumentation code will intercept and forward relevant system events to the monitor which verifies whether the event violates a correctness property.

III. Overheads

One consideration that must be taken when applying RV is that of minimising overheads. A runtime verifier will observe a systems' behaviour to verify whether the correctness specification has been violated. This means that there will be a performance impact on the target system if the runtime verifier resides onboard, and/or a network performance impact if messages are intercepted using proxies. In such a performance critical system such as an ESB, we are tasked with finding the optimal balance in a trade off between providing an adequate correctness guaranteeing system and performance impact.

IV. Three approaches identified

In order to find an optimal solution (in terms of overheads) for applying runtime verification techniques to an ESB application, we propose three separate approaches in terms of the level at which we are instrumenting the code for intercepting ESB events. A runtime verifier will allow a user to specify correctness properties for the target system. This way, the user is specifying which system events are required to verify correctness of the system. The verifier is responsible for intercepting these events and forwarding these to the correctness logic code for verification, the level at which we are intercepting ESB events is the main variable between the three proposed approaches. These approaches shall be evaluated against a real world ESB use case in order to determine which approach produces least performance impact.

Figure 3 shows the three different levels at which we may intercept ESB system events. In an ESB application a user may connect two or more components in a configuration script. In doing this, the script shall call the underlying ESB source code for sending an event. This creates a connection with the second component using the protocol specified by the user. The three approaches discussed here relate to the three levels depicted in Figure 3.

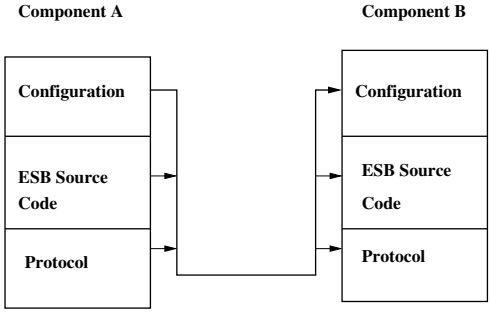

Fig. 3. Three Levels for Intercepting Events

A. ESB Configuration Level

The first approach investigated is a variation of Larva [8], with the difference being that a user will configure correctness properties at a higher level of abstraction (than the source code level). Most ESB solutions (including the open source Mule ESB) provide a configuration layer where the integration logic may be configured from one single point via a script which is more intuitive than for example via source code. This configuration provides a domain specific language which may be grasped relatively quickly by developers as opposed to learning the internals from source code. This approach will allow users to specify correctness properties for the runtime verifier requiring a knowledge of the ESB configuration, rather than ESB source code.

B. ESB Source Code Level

The second approach is to allow the user to configure correctness properties by specifying events intercepted in the source code of the ESB. This approach will require the user to have a deep understanding of the ESB source code, however it also allows the user to intercept most event types identified in the source code. The technology used for intercepting system events at runtime is the AOP implementation AspectJ. This technology provides separation of concerns in terms of source code, allowing the user to invoke external code at certain points within the target system code. This approach has been applied to an ESB use case application using the research tool Larva.

C. Proxy Level

In both the previous two approaches, a performance impact is inhibited over the ESB system directly for collecting system events. In order to lessen this impact we may collect the events via a proxy which intercepts requests and responses to and from the ESB. Once the required system events have been intercepted via proxies, these may be forwarded to the verifier as in the two previously explained approaches. One limitation that may affect the effectiveness of this approach is that ESBs are designed to communicate with multiple protocols, therefore proxies for multiple protocols shall be required for this approach to be effective.

V. Conclusion

With the use of an example ESB application we have shown that testing and model checking techniques are ideal for addressing issues which require only information available at compile time. Due to the nature of ESB applications, there are circumstances where this will not suffice. For instance when the ESB application is expected to communicate with unknown remote components, we propose runtime verification techniques where we are only interested in the current thread of execution and have all runtime information available. We also propose three separate approaches for applying runtime verification techniques with the aim of evaluating which of these approaches inhibits least performance impact on the ESB.

References

[1] David A. Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly, 2004.

[2] Ross Mason. Mediation - separating business logic from messaging, May 2013. http://www.mulesoft.org/documentation/display/34X/Mediation

[3] Anne Thomas Mane. Enterprise service bus: A definition, May 2013. http://i.i.cbsi.com/cnwk.1d/html/itp/burton_ESB.pdf

[4] Yousef J. Al-Houmaily and George Samaras. Two-phase commit. In *Encyclopedia of Database Systems*, pages 3204–3209. 2009.

[5] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *JLAP*, 78(5):293–303, 2009.

[6] Séverine Colin and Leonardo Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555, 2004.

[7] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *RV*, volume 6418 of *LNCS*, 2010.

[8] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE Computer Society, November 2009.