Run-time monitoring for model-based testing of distributed systems

Vladimir Fedotov[1]
Institute for System Programming
Moscow, Russia
Email: vfl@ispras.ru

***Abstract*—Modern enterprise systems are highly distributed and heterogeneous. Apart from the latest attempts on leveraging distributed systems with SOA and SOA-like enterprise integration systems, testing still represents a major challenge.**

This paper discusses practical approach for managing the testing process for distributed systems based on transparent test environment, run-time monitoring of interactions within this environment and interaction model generation.

We also outline an approach for test case generation based on the interaction model and test coverage metric based on the coverage of interaction tree.

I. Introduction

Integration technologies are rapidly advancing since early 90-s, driven by consistently faster networking and better data storage. Being mostly a business domain, integration technologies constantly evolve, leaving in their wake various vendor-locked platforms as a legacy. This legacy forms a heterogeneous environment that is common for any enterprise company big enough.

Enterprise environment is often divided between several technology domains, each formed around a certain kind of solution like an integration broker, application server or service bus. Therefore environment as a whole is highly distributed. Bringing this environment together is a daily struggle for an enterprise IT.

The web-service stack of protocols is the latest attempt to deal with this issue. Centered on Web-service Definition Language (WSDL) web-service protocols provide standardized interface for integration components. Combined with stateless design it enables the strongest feature of web-services – compositions.

Composition is a web-service that acts as a client for several other web-services. Compositions can be stacked over each other to implement different tasks over existing functionality of the system, creating highly distributed environment. Enterprise systems often follow the pyramid pattern (Fig.1), wrapping enterprise applications with web-service interfaces and stacking several layers of compositions on top of them.

Interface complexity, the number of operations and operation parameters, grows from the top to the bottom of the pyramid. Low-level services may wrap an entire API of the underlying systems, requiring an expert knowledge of the business domain for client development. Top-level services implement very specific business processes and provide only basic interface that requires little knowledge of the system internals and can be exposed to the third-party developers.

Integration complexity, the number of outgoing requests for each incoming request, as defined in [1], grows from the bottom to the top of the pyramid. Low-level services are tightly coupled to the applications they wrap thus having zero integration complexity. Top-level services have multiple dependencies on the services below them that have dependencies of their own.

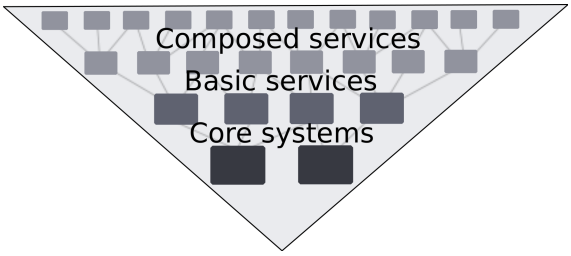

Fig. 1. Pyramid pattern in enterprise application integration

In this paper we would like to discuss the testing process for the systems described above. While web-services by themselves bring nothing new to the classic V-model testing process, compositions testing present an actual challenge, shifting focus from functional to the integration testing. Developing a methodology adjusted for testing of web-service compositions is a goal of the research described in this paper.

II. Motivation

Distributed heterogeneous nature of the enterprise systems presents several major issues that need to be dealt with in order to get meaningful consistent testing process. Other issues described below are the consequences of data-flow centered logic of integration components that makes them tightly coupled to data that is stored externally.

Typical enterprise system consists from several different technology platforms, that implement various, often proprietary, protocols. Applications supporting these protocols form a domain around the platform, which means that test environment is fragmented according to these domains as there are no connections between tests from different domains. Main

[1]This work is partially supported by RFBR 11-07-00084a, 11-07-12075-ofi-m grants.

consequence of environment fragmentation is lack of end-to-end tests that hampers a system testing stage of the process.

While individual components contain only integration logic i.e. transformation between different message formats, actual business logic resides in data stored in the core systems such as billing or resource management. Constraints that exist in this data describe what is possible and what is not in the system. Therefore testing of business logic requires knowledge about these constraints and a way to mine data corresponds to a certain set of constraints. Both of these tasks are extremely difficult as the data structures in the core systems may be incredibly complex. In practice constraint discovery and data mining often done in an informal way: by brainstorming the database structure or consulting with business experts.

The price of informality is that there are no guarantees of completeness for discovered constraint set. It may be too strict, so certain types of input data is not represented in test cases. Or it may be too loose, so certain test cases will fail for no apparent reason. The main consequence of this is that there is no appropriate way to measure test coverage. As test cases are data-driven, they should be executed with the same requests but different data, so typical coverage metrics, such as amount of covered web-service operations, become inadequate to actual test subject – business process implemented by the component composition.

III. Outline

Approach proposed in this paper can be logically divided into two parts: firstly we try to bring the test environment under control by developing a transparent test framework that offers us an ability to observe and control interactions within the test environment; secondly, we develop a technique for modeling these interactions, that gives us a way to formally reason about what is happening in the system.

Test framework described above is based on existing application integration solutions. We are extending an existing open-source enterprise service bus, that offers us various protocol adapters, message routing and transformation capabilities. By having centralized, possibly federated, test environment we deal with test environment fragmentation. Supporting the environment also becomes easier, as protocol adapters can be developed independently and do not affect other parts of the environment. Adapters are connected to the Normalized Message Router that transforms various message formats into canonical one, thus making end-to-end testing easier.

Another important feature of our framework is an interaction monitoring. It provides us with a bridge between real system and a model. As it seems impossible to derive interaction model from component specifications, we see run-time monitoring as a best possible alternative. Run-time monitoring at the unit-testing stage of process helps us to create behavior models for independent components. At the later stages interaction monitoring helps us to determine the outcome of the test executions and coverage reached.

The goal of model generation is to provide binding between data-driven test cases and composition behavior to deal with data constraints discovery issue. Model is built bottom-up, starting from request-reply pairs for individual component and growing to composition of models of several components. It is presumed that model generation starts at the unit testing stage, where data mock-ups are used. At this stage, model describes what types of behavior are possible for an individual component, later we bind them to an actual data, representing an equivalence class.

When interaction model for component composition is ready, we generate test scenarios that represent a certain interaction path within a system. Test stimulus is represented by a message that would be sent into composition entry point, while reaction is a set of all consequent interactions that happened within a system. Stimuli that have the same structure, but contain different data may produce an entirely different interaction pattern. The power of the approach is in testing various interaction patterns that may be hidden behind the entry point.

Essential feature of our approach is test coverage metric. Data complexity in an enterprise system often prevents instantiation of certain classes of data as we are unable to satisfy constraints that exist in the system. Well-managed testing process always aims for a perfect balance between risks and man-hours, so we need a tool to evaluate an amount of work required for test data instantiation and compare it to risks coming from not testing on this data. Our test coverage is measured as coverage of possible interaction patterns in the composition. This coverage metric includes not only coverage of reachable request-reply pairs, but also coverage of reachable data classes discovered on earlier stages of process.

IV. Approach

A. Transparent environment

Basis of our approach is the observable environment. While evaluating properties of the components under test we presume that interactions between these components are observable.

In practice it might be difficult to achieve such level of transparency. For example, there is no easy way, known to us, to observe database interactions. Also interactions over proprietary binary protocols, common for message queues, are observable, but not decodable. Still, most interactions are done over HTTP, with messages formatted in XML, so they are easily observable and readable.

Second part is to bring all the observations together in a single framework. Surely a bunch of HTTP sniffers here and there would not do much for our goal. As a solution we suggest the existing open source ESB platforms like ServiceMix, Synapse and others.

First of all, ESBs already have a lot of adapters for different protocols, thus widening our reach for many different platforms. Second, the concept of an ESB presumes existence of single point of observation for everything that happens inside a system. Third, messages passed into an ESB are normalized to canonical form, so we are relieved from a burden of handling different message formats.

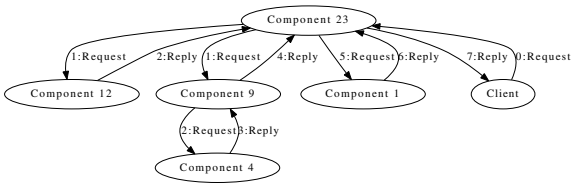

Fig. 2. Interaction graph

We have not yet reviewed all available ESB platforms, Apache Synapse looks most promising due to ease of its configuration by XML-based configuration language, but protocol support is somewhat limited in comparison to other platforms. We will also look into possibility of developing our own solution on top of the existing ones.

The role of the framework described above is to provide a new entity of the system - an observer. Observer should be able to log, analyze, transform and re-route messages passing through him. As we actually see interactions within the environment only if they are visible to observer, modeling the environment actually means modeling the observer's state. We see an observer's state as a queue of incoming messages. In a certain period of time message queue gets processed which basically means that messages get sent to their destinations. Queue-based processing gets us a handy abstraction of time for our model. Instead of dealing with continuous time, we have discrete time represented by a message polling interval of the observer that can be imagined like a turn in a turn-based computer game. For example, messages are considered concurrent if they were retrieved on the same turn and sequential if message B followed message A exactly on the next turn.

B. Interactions model

The first and easiest step of our approach is creation of a connection graph of the system, like the one shown on the Fig.2, that basically represents connections between components. This graph is created by monitoring of the message-flow on the observer and requires almost no processing other than message headers where message destination resides.

The second step is the creation of the interaction trees shown on the Fig.3 that maps the whole interaction between components to a single message that started it. To create an interaction tree we should implement basic rules for message correlation:

1) Messages are concurrent if they were sent on the same turn
2) Messages are sequential if the second message was sent on exactly next turn and destination of the first message matches to source of the second message
3) If there is no messages in the observer's queue, the interaction has ended

The third step is to discover relations between actual request data and component behavior. This step is necessary for a meaningful model as our components are data-driven and their behavior may depend not only on the type of incoming data, but also on its semantics. By developing this technique

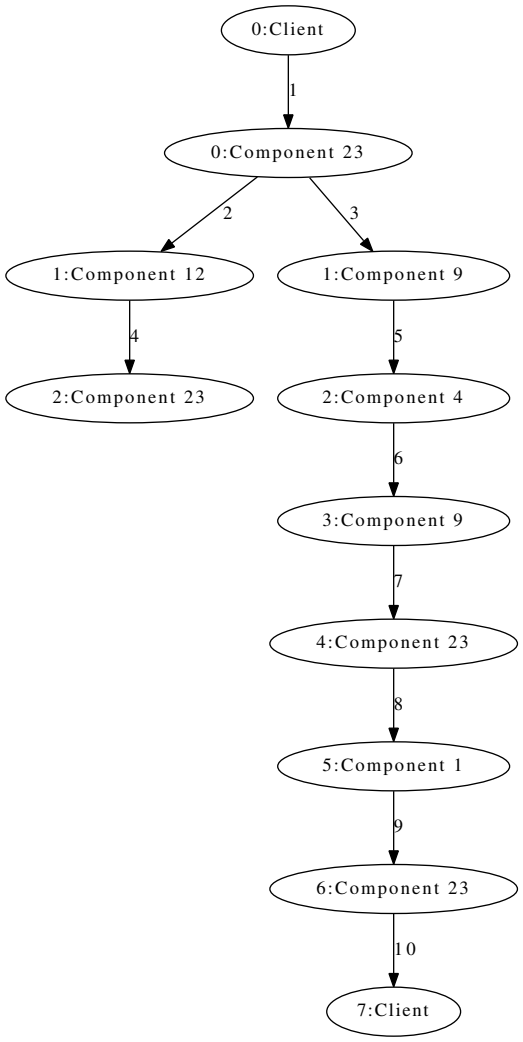

Fig. 3. Interaction tree

further we hope to implement a set of behavioral heuristics for discovering common dependencies in the input data, such as message ordering and equivalence classes.

C. Test generation

Final step of our approach is a test generation based on the interaction model enriched with the discovered classes of input. To test a certain composition we need to derive the minimal set of test stimuli from a model that is able to cover all reachable branches of the interaction tree.

This task is achievable in case we would be able to discover and correlate inputs properly, the problem here is that these tests would be abstract i.e. they would contain a description of classes of data instead of a real data. Real data gathered by observer may become unusable in case of update operations, as the updated data no longer represents its original class.

Currently we see no possible general approach for discovery of concrete data instances, so we presume it is done manually. As it is certainly possible that some classes of data would not have concrete instances (because we were unable to find them,

not because they do not exist), it is vital to develop a coverage metric for these test sets.

D. Model semantics

Finally we would like to discuss a topic somewhat unrelated to the practical application of our approach, but essential for its further development – formal semantics of our model. Currently we are considering several different semantics. First one is widely used LTS semantics [2] [3] [4]. LTS model represents a system as a set of vertices - states and a set of labels - actions that perform a transition from one state to the other.

As we discussed earlier, we consider components as a stateless entities, so it may be unclear how to model them using LTS semantics. We propose a slightly different approach for using the same semantics. As we are modeling a data-flow in the system, we represent state as a message in an observer's queue and transition as an action performed by a component that results in transition to a new state i.e. getting new message in a queue. This approach is expressive enough, but there are certain difficulties in its practical application.

First of all, as we discussed earlier, our inputs have state of their own that should be included in the model. In reality it is some sort of the global state represented by a set of attributes stored in a database, but we think that modeling a global state is not exactly a good idea, because it brings unnecessary complexity to a model without any real value.

We deal with that issue by splitting our state, a message in reality, in two parts: implicit and explicit. Explicit part of the state is an actual message received, while implicit part is an associated context that is hidden from us somewhere in the system. By doing so we get rid of non-determinism we showed earlier, where two same messages may produce different reactions. Here it means that only explicit parts of the messages were equal, and implicit parts were actually different, so we have two different states and there is no indeterminism in these cases. Declaring the two states different is done solely by looking on the results of the same actions, so we do not need to actually compare the implicit parts.

Another option we are considering is somewhat less known actor semantics [5] [6]. Actors model was introduced by Hewitt in 1973 [7] and was supposed to model concurrent systems as a set of related entities – actors. Actors communicate via reliable messaging and have a state of their own. For every incoming message actor can create more actors, send more messages or change its own state.

Most recent and successful implementation of an actor model is done in the Scala language [8], which served for us as an inspiration to look into actor semantics. Scala provides an actor framework for implementation of concurrent systems in a clear and concise way that differs a lot from a traditional locking mechanism.

Model-checking for actors modeled in Rebeca language [9] was successfully implemented in Modere [10] model-checking engine. For now these are the only works in formal verification for actors, but while recent development of an actor model itself was not very active, we see its expressiveness in our domain as a big advantage. Components, especially web services, can be naturally described as actors. Message passing as a way of communication also fits naturally in our approach. Another big advantage is model scalability – a way of composing smaller models into larger ones described in [11].

V. Related work

Testing and verification of the distributed systems is a very popular field of academic research, mostly due to recent peak of SOA-related technologies. The formal specification language - WSDL and the composition description language - BPEL provided by web services attract attention as they seem to be very prominent tools for implementation of various model-based testing techniques.

The ideas that are closely related to ours and, more importantly, already implemented in the Plastic Validation Framework [12] were expressed in the works of Bertolino et al. [13] [14] [15]. The concept of the model-based generation of the test environment, expressed in [14] is very close to our approach. Unfortunate downside of the proposed methods is their limitation to the domain of web services. There also no clear description of the data binding mechanism as web services are described as stateful entities.

Another important work related to our topic, done by Sharygina and Kröning and included in [16], discusses the application of the model-checking techniques in the domain of web-services. Being a preliminary work it only discusses the model checking for certain properties such as deadlocks, but can be easily extended for more practical cases.

Castagna et al. [17] developed the theory of contracts for Web services. These contracts are used as behavioral descriptions of Web services and offer concise and formal way for reasoning about their properties.

Ferrara [18] developed the process algebra approach for reasoning about BPEL services. This approach maps formal abstracts to concrete web-service implementations done in BPEL4WS language.

Textor et al. [19] proposed formal workflow model for SOA monitoring. This model is used for Quality-of-Service monitoring for enterprise applications. Described approach was successfully implemented for self management of the actual enterprise system.

VI. Conclusion

In this paper we have introduced a model-based approach for testing distributed systems. Our approach has strong emphasis on practical application and is based on the run-time monitoring of the system. To enable such monitoring in real enterprise systems we develop transparent test environment that acts as an observer for interactions between components of the distributed system. We use existing open-source ESB as a technology platform for the test environment.

We have outlined an approach for generation of a model of distributed system that is based on observing behavior patterns

of the individual components. This model is composable and can be used throughout all stages of the testing process, from unit-testing up to end-to-end acceptance testing. The main feature of the model is that it allows binding component behavior to the semantics of the input data. Described model is used for generation of tests that cover possible interaction paths and input data classes.

We have also discussed suitable formal semantics for a described model. LTS semantics is common for model-based techniques, but cannot be applied in a straightforward manner because of the stateless design of the system components. Instead of modeling the component state, we model the state of the observer that is represented by a message or multiple messages for concurrent interactions. Another suitable semantics is the actor model, introduced by Hewitt and implemented in Erlang and Scala programming languages. Despite being unpopular for means of formal verification, we see the actors semantics superior to others mostly because of scalability of the models defined with actors.

Presented work is still very much in progress. Description of the model and test generation techniques is preliminary and would be improved in future. We also plan to put more efforts in researching the actors semantics.

REFERENCES

[1] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[2] J. Tretmans, "Model Based Testing with Labelled Transition Systems."

[3] V. Kuliamin, "Component architecture of model-based testing environment," *Programming and Computer Software*, vol. 36, no. 5, pp. 289–305, 2010. [Online]. Available: http://dx.doi.org/10.1134/S036176881005004X

[4] I. Burdonov and A. Kosachev, "Conformance testing based on a state relation," *Proceedings of the Institute for System Programming of RAS*, vol. 18, pp. 183–220, 2010.

[5] C. Hewitt, "Actor Model of Computation: Scalable Robust Information Systems," pp. 1–29, 2011.

[6] S. Smith, I. A. Mason, and C. Talcott, "Towards a Theory of Actor Computation."

[7] C. Hewitt and P. Bishop, "A universal modular actor formalism for artificial intelligence," *Joint conference on Artificial intelligence*, pp. 235–245, 1973.

[8] M. Odersky and L. Spoon, "Programming in Scala," 2008.

[9] M. Sirjani and M. M. Jaghoori, "Ten Years of Analyzing Actors : Rebeca Experience."

[10] A. Movaghar, "Modere : The Model-checking Engine of Rebeca Mohammad Mahdi Jaghoori," pp. 1810–1815.

[11] A. Gul, "A Foundation for Actor Computation," no. July 1993, 1993.

[12] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "The PLASTIC Framework and Tools for Testing Service-Oriented Applications," pp. 106–139.

[13] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "WS-TAXI: A WSDL-based Testing Tool for Web Services," *International Conference on Software Testing Verification and Validation*, pp. 326–335, Apr. 2009.

[14] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-Based Generation of Testbeds for Web Services."

[15] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening SOA Testing," pp. 161–170, 2009.

[16] L. Baresi, *Test and Analysis of Web Services*, 2007.

[17] G. Castagna, N. Gesbert, and L. Padovani, "A theory of contracts for Web services," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 5, pp. 1–61, Jun. 2009.

[18] A. Ferrara, L. Sapienza, and V. Salaria, "Web Services : a Process Algebra Approach," pp. 242–251.

[19] A. Textor, M. Schmid, J. Schaefer, and R. Kroeger, "SOA Monitoring Based on a Formal Workflow Model with Constraints," *Applied Sciences*, pp. 47–53, 2009.