Towards a Reactive Semantic Execution Environment

Srdjan Komazec and Federico Michele Facca

Semantic Technology Institute (STI) - Innsbruck
ICT Technologiepark, Technikerstrasse 21a, 6020 Innsbruck, Austria
firstname.lastname@sti2.at

Abstract. Managing complex and distributed software systems built on top of the service-oriented paradigm has never been more challenging. While Semantic Web Service technologies offer a promising set of languages and tools as a foundation to resolve the heterogeneity and scalability issues, they are still failing to provide an autonomic execution environment. In this paper we present an approach based on Semantic Web Services to enable the monitoring and self-management of a Semantic Execution Environment (SEE), a brokerage system for Semantic Web Services. Our approach is founded on the event-triggered reactivity paradigm in order to facilitate environment control, thus contributing to its autonomicity, robustness and flexibility.

1 Introduction

Current trends in software development show more and more a growth of loosely coupled and widely distributed heterogeneous systems based on the Web. In the recent past, research efforts mainly concentrated on solving the problem of integration of such systems by leveraging semantic technologies. For example, the work related to the Web Service Modeling Ontology (WSMO)[1] framework originated from this problem. We think for the same reasons that semantic technologies can facilitate the integration of loosely coupled and heavily distributed systems, they can also efficiently and effectively support their monitoring and reactivity. Oberle[2] paved the road by studying the benefit of semantic technologies to support development and management of the middleware-based applications. Our research does not simply aim at improving management and monitoring of distributed software systems through semantics, we aim at increasing the level of automation of such systems inline with the Semantic Web Service approach. Thus we think it is possible to relieve humans from performing certain maintenance operations for such systems by making them self-manageable and adaptable through the adoption of a formal description of the system and the actions that can be performed on it to govern its overall behavior dynamically.

Naturally, the approach presented can be applied to any distributed software system. In this paper we focus on a Semantic Execution Environment (SEE), a complex distributed system that enacts the Semantically Enabled Service-oriented Architecture lifecycle. In this way we are self-reflective: We are testing

R. Meersman, P. Herrero, and T. Dillon (Eds.): OTM 2009 Workshops, LNCS 5872, pp. 877–887, 2009.

our approach, based on Semantic Web Services, to manage themselves. Moreover, this simplifies the implementation work needed for lifting events from the syntactic level to the semantic one, given that SEE is already semantically enabled and current efforts toward its description based on semantic technologies are ongoing. The approach we present adopts the WSMO framework for the description of SEE components, related events and their correlation (in term of ontologies) while the reactive behavior enabling self-management of the systems is described in terms of Abstract State Machines (ASM).

The paper structure is the following: Section 2 presents SEE and its limitations; in Section 3 we describe our approach detailing its formal grounding; Section 4 applies the approach to the SEE; in Section 5 we provide a discussion of related research efforts. Finally, Section 6 summarizes the work presented and depicts future work directions.

2 The Semantic Execution Environment

Reactivity can improve behavior in many ways such as performance, adaptation, stability and resilience to failures. In this paper we apply our research on semantically-enabled monitoring and reactivity to a family of complex service brokerage platforms based on semantic technologies called Semantic Execution Environment (SEE)[3]. SEE is foreseen as the next generation of service-oriented computing based on machine processable semantics. Its aim is the automation of all the steps of the (semantic) web service's lifecyle: i.e., discovery, composition, ranking and selection, data and process mediation, choreography, orchestration, and invocation. For each step of the lifecyle, SEE adopts a dedicated broker service; brokers are orchestrated together in the workflows to enable complex scenarios where the different steps of the lifecyle are covered.

Even though many research efforts have driven the SEE initiative and its standardization, little work has been done so far in the area of SEE monitoring and self-management, e.g. Vaculin et al.[4]. The observation is that the temporal and highly dynamic knowledge related to the SEE execution (and execution of external Web Services) can enable a high level of self-management. In particular, we foresee the following areas in which SEE can be improved by the application of our approach.

Closing the Knowledge Control Loop. Monitoring of SEE broker services represents an indispensable step towards creating a more adaptable environment. Rich and valuable knowledge is produced during the environment execution that can be used for various purposes. For example, service invocation time and availability can be collected to affect the service ranking. Historical data can also be used to identify performance bottlenecks in the environment, restore the systems state at some point in time or even predict future behavior of the environment.

Fault Management. Failure detection of the broker and external services can be fixed by a number of strategies such as discovering and employing other components/services capable of fulfilling the user goal or reattempting to accomplish

the task with the same service. Special attention must be given to the capability of the failed service to persistently change the state. In such a case, a compensation mechanism must be provided to suppress the effects of the previously unsuccessful invocation.

Self-managed Environment. While SEE is contributing to the dynamic usage of Web Services, its behavior is rather fixed. It is not capable of coping with unexpected situations (e.g. failover of collaborating peers or unavailability of resources such as ontologies and Semantic Web Service descriptions). Furthermore, the overall system functionality is predefined and fixed which hinders any possibility to react to unusual circumstances, i.e. to dynamically change the process when confronted with new conditions. This situation requires support of a rule-based event-triggered reactivity in order to enable a reliable and adaptable environment. For example, in order to enable a Web-scale behaviour of SEE, it could be interesting to introduce some heuristic in its behaviour: when the discovery process takes too long, SEE stops the process and considers only the services discovered till that moment.

3 Monitoring and Self-management of Semantic Execution Environment

The architecture of our solution represents an implementation of the *closed control loop* paradigm. Systems based on this paradigm observe its parameters and autonomously respond with appropriate actions, as presented in Figure 1. All the SEE brokers are instrumented, thus capable of emitting events. The events are registered in the storage governed by the ontologies. Decisions regarding the actions taken upon sensing/deriving knowledge about the system are delegated to the *Reactivity Engine*. The engine detects (complex) situations of interest by consulting the registered events, analyzing them in some broader context (e.g. consulting additional knowledge) and selects appropriate actions, which will be enacted over the system.

Ontologies. To enable the formalization of the monitored system and its events, and to support the resolution of heterogeneity conflicts, we rely on a stack of ontologies comprising the knowledge base of runtime system behavior. Implementation of the ontologies is flexible enough to give an opportunity to adjust them to a specific domain of application. We provide two upper level ontologies:

- *Time ontology* - which enables reasoning about time points and intervals,
- *Events ontology* - which classifies basic events that can be observed in the domain of interest.

The foundational concepts of *Time* and *Events* ontologies share the same ground principles as presented in the work of Pedrinacci et al.[5]. The *Time* ontology defines basic time-related notions such as **TimeInstance** and **TimeInterval**

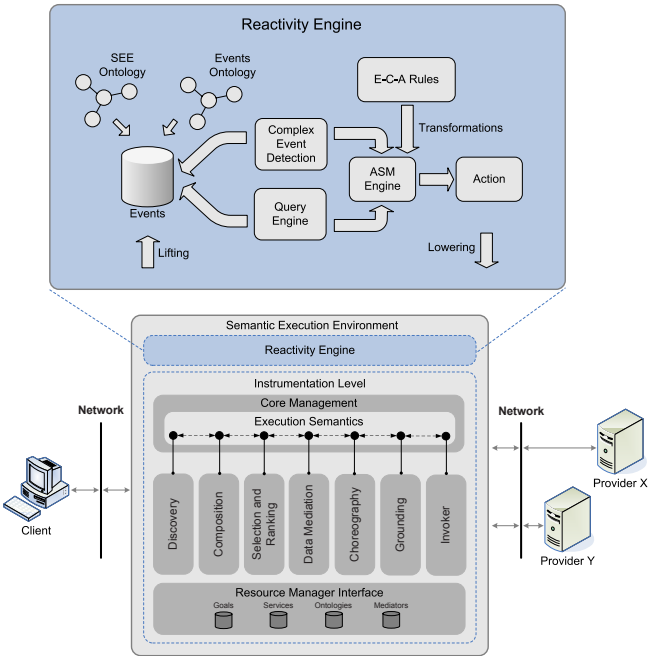

Fig. 1. Architecture of the presented approach

as well as set of axioms (e.g. **before**, **after**, etc.) which enable reasoning about time notions compliant to Allen's algebra[6].

Events ontology relies on *Time* ontology and defines **MonitoringEvent** and **MonitoringActivity** as the ground concept. MonitoringEvent represents a point in time when an event of interest happened and is related to the domain component responsible for its generation (the **generatedBy** property). Since we are envisioning applications of the framework in a distributed environment (where clock synchronization represents an issue), the concept defines two distinctive time points (**occursAt** which stands for the component local time when the event occurred, and **receivedAt** which is the time point of event reception by a monitoring subsystem). MonitoringActivity represents a time interval bounded by two MonitoringEvent occurrences.

Modeling a Component and Its Behavior. We adopted WSMO for modeling service's (i.e. broker's) capability and behavior. The usage of a SWS framework suits our needs because it gives formal characterization of a service's functionality which can be easily related to the events represented in the ontological manner and to the reactivity rules.

Modeling Global Reactivity. The global system behavior can be modeled by relying on various techniques, such as Finite State Machines, Petri Nets, and UML State Chart diagrams. The assertions stated above are main motivating

factors to select Abstract State Machines (ASM)[7] as the used approach. Reasons to use ASMs compared to the the other approaches stem from the set of properties that distinguish them and makes them more suitable, such as arbitrary levels of abstraction, formality, minimality, state-based, and maximality.

An approach used to describe ASMs is borrowed from Scicluna et al.[8]. An ASM consists of a finite set of transition rules which are executed in parallel. The rules operate on top of a finite collection of static and dynamic functions. Values of the former cannot change during a run of a machine, while values of the latter can be changed by environment (agents interacting with the machine) or the machine itself. Our approach regards reactive behavior as a pair $\langle signature, rule \rangle_{react}$ where

- *signature* is a state signature which precisely defines the structure of the state in the context of the Knowledge Base (KB), and
- *rule* is a set of guarded transition rules.

A state signature is a pair $\langle KB, mode \rangle_{ss}$ where *KB* is a knowledge base that defines concepts comprising the state and *mode* is a set of *modes* which take a form of $\langle type, concept \rangle_{mode}$ where

- *type* can be one of the symbols *static*, *in*, *out*, *shared* or *controlled*, as described by [8], and
- *concept* is a concept defined in *KB*.

A guarded transition *rule* can take the form of

- an *if-then* rule $\langle cond, rule \rangle_{if}$,
- a *forall* rule $\langle variables, cond, rule \rangle_{forall}$,
- a *choose* rule $\langle variables, cond, rule \rangle_{choose}$,
- a *piped* rule $\langle rule \rangle$,
- an *add* rule $\langle \alpha \rangle_{add}$,
- an *update* rule $\langle \alpha \rangle_{update}$, and
- a *delete* rule $\langle \alpha \rangle_{delete}$.

where *cond* is a condition such that the set of free variables in it corresponds with *variables* which is a set of variable identifiers; *rule* is a nonempty set of guarded transition rules and α is a fact (i.e. variable free instance of knowledge).

The ASM description is additionally extended with rules that enable invocation and discovery of Web Services. The *invokeWS* rule $\langle wsIRI, fact \rangle_{invokeWS}$ invokes the Web service identified by *wsIRI* and uses *fact* as input data. The *discoverWS* rule $\langle goalIRI, wsIRI \rangle_{discoverWS}$ executes discovery based on the goal identified by *goalIRI* and binds to the variable *wsIRI* result of the discovery process.

4 First Prototype Implementation

Although the research presented in this paper is not yet mature we have realized a first prototype to validate our claims. The prototype is based on the case study introduced in Section 2.

Extension of the Event Ontology. The Event Ontology briefly discussed in Section 3 has been extended with the definitions of events and activities related to the SEE domain. This ontology relies on the Reference Ontology for Semantically-enabled Service Oriented Architecture[9], which formalizes notions of interest such as Service Description, Goal Description, etc. For example, the *InvokeWebServiceActivity* from the extended ontology describes either successful or unsuccessful invocation of an external Web Service (the distinction is made by the usage of appropriate *MonitoringEvent* type, *InvokeWebServiceEnded* or *InvokeWebServiceFailed*).

Instrumentation of the SEE Environment. Event acquisition is performed by proper instrumentation of the observed brokers during which suitable lifting mechanisms are used, i.e. transformation of collected low-level data to the semantically represented counterparts according to the extended ontology. Instrumentation has been conducted by application of Aspect Oriented Programming which provides a non-intrusive technique for weaving the crosscutting behavior (like security, monitoring, logging, etc.) into the code. The developed aspects are merely collecting data needed to create the appropriate *MonitoringEvent* and *MonitoringActivity* instances and storing these instances into the repository that is used by the Reactivity engine.

4.1 Examples of Self-management Behavior

Closing the Knowledge Control Loop. Many service-related properties such as Web Service availability, response times and number of failed invocations over the total number of invocations can affect the service rating during the ranking and selection step. Let's assume that there exists a Quality of Service ontology used by the Ranking broker which holds an instance of the Ranking concept for each external service registered in the system. The Ranking concept has a couple of associated attributes like *hasWebService*, which identifies a Web Service, *hasRankingValue*, which defines current ranking value for the Web Service and *hasAvgDuration*, which is the average duration of time experienced during past Web Service invocations. Let's further assume that, for the sake of simplicity, ranking is defined as

$$rank(ws)_{new} = rank(ws)_{old} - \frac{t_{end} - t_{start} - T_{avg}}{T_{avg}}$$

where *ws* represents the Web Service identifier, t_{start} and t_{end} represent starting and ending time for the Web Service invocation and T_{avg} represents the

```
reactivity RankingAdaptation

  stateSignature
    importsOntology{
      „" http://www.sti−innsbruck.at/ontologies/Ranking#",
      „" http://www.sti−innsbruck.at/ontologies/EventsOntology#"
    }

    in events#
    out ra#Ranking

  transitionRules
    forall {?event, ?ranking}
      with
        ?event [
          startTime hasValue ?startTime,
          endTime hasValue ?endTime,
          hasWebService hasValue ?wsID
        ] memberOf events#InovokeWebServiceActivity
        and
        ?ranking [
          hasWebService hasValue ?wsID,
          hasAvgDuration hasValue ?tAvg,
          hasRankingValue hasValue ?rankOld
        ] memberOf ra#Ranking
    do
      update ?ranking [
        hasWebService hasValue ?wsID,
        hasRankingValue hasValue ?rankOld − (?endTime − ?startTime − tAvg) / tAvg,
        ]memberOf ra#Ranking
  endForall
```

Listing 1. Updating ranking value after Web Service invocation

average duration of the Web Service invocation. It is worth noting that each time the Web Service is successfully invoked an instance of the *InovokeWebService-Activity* is generated which has, as associated attributes, *startTime*, *endTime*, and *hasWebService*, thus identifying the invocation of the particular Web Service. An example of reactive behavior which updates the Web Service ranking value upon detection of the *InovokeWebServiceActivity* instance is presented in Listing 1.

Fault Management and Self-managed Environment. A service invocation failure situation can be resolved in a different ways depending upon the adopted strategy. In the case of idempotent method invocation the operation can be repeated again in order to exclude issues bound to underlying network failures. Successive invocation failures can further be regarded as a service-related issue, in which case the system can decide whether to enact the second best solution to fulfill the user goal. This behavior is represented in Listing 2.

The first *forall* rule identifies one sole occurrence of the *InvokeWebService-Activity* which has failed (i.e. closing activity event is of the type *InvokeWeb-ServiceFailed*). The response of the system is to initiate another invocation of the same service with the same input data. The second *forall* rule detects two occurrences of failed *InvokeWebServiceActivity* related to the same session and Web Service invocation. In this case the system exercises another discovery and invokes the newly discovered Web Service. Initial experiments show that the application of the proposed approach to SEE improves its functionality by reducing the number of system failures and by allowing for example, QoS ranking of services.

5 Related Work

Reactive behavior is drawing attention to many areas of computer science. The early work of Harel et al.[10] clarified the founding notions of reactive systems by relying on the statecharts method (grounded in Finite State Automata, FSA). By that time it was clear that a reactivity modeling approach needs to support state-based computational processes, but FSA was abandoned very soon since it does not support modularity neither hierarchical structures and suffers from exponential growth of state and transition numbering. Nowadays, reactivity represents a core pillar of IBM's Autonomic Computing initiative[11], which advocates system self-management by introducing a monitor-analyze-plan-execute control loop based on knowledge, thus relieving humans from of the responsibility of directly managing systems. Veanes et al.[12] presented an approach to on-the-fly testing of reactive systems based on Abstract State Machines. Besides testing, our approach envisions support for complete reactive behavior development (i.e. modeling, implementation, testing and execution).

Active Databases largely contributed to the body of knowledge around reactive systems. The work of Chakravarthy et al.[13] introduces a language called Snoop on top of which they define different detection contexts and event detection architecture (based on even-detection trees). Gehani et al.[14] gave yet another formalization of complex event expressions and associated operators as well as an incremental detection technique based on FSA. Our work has been directly inspired by the paradigm since Web Service Modeling eXecution environment (WSMX) instances coupled with the reactive engine can be seen as managed elements in a collaborative environment tailored in the P2P fashion. Schmidt et al.[15] introduced a survey on current trends in the field of event-driven reactivity where they examined a variety of approaches in the area of Complex Event Processing (CEP) and Event-Condition-Action (ECA) rules. According to their categorization, our solution resides in the area of logic-based approaches, where the most prominent work is represented by Paschke[16].

Monitoring represents the first step in establishing the reactive behavior of a system. Wang et al.[17] enumerate a number of events that can be produced by SOAs where the observability of the specific event type depends on the possibility of positioning appropriate probes in the monitored object. However, the research community showed only limited interest in Semantic Web Service monitoring. Vaculin et al.[4] defined an event taxonomy and a solution for detection of complex events in the context of the OWL-S choreographies. Nevertheless, the existing approaches are not considering any application of the service capability and behavior models (i.e. service choreography and orchestration) during the monitoring functions, neither monitoring of the overall SEE environment. Oberle[2] presented an ontology-based approach to support development and administration of the middleware-based applications. Although currently absent, formal characterization of the Semantic Execution Environments can contribute to the annotation and processing of the complex events produced by them.

```
reactivity ServiceInvocationFailureRecovery

  stateSignature
    importsOntology {
      _"http://www.sti-innsbruck.at/ontologies/TimeOntology#TimeOntology",
      _"http://www.sti-innsbruck.at/ontologies/EventsOntology#EventsOntology",
      _"http://www.semantic-soa.org/ReferenceOntology" }

    in domain#GoalDescription

    transitionRules
      forall {?x}
        with
          ?x memberOf InvokeWebServiceActivity and
          ?x[hasInputInstances hasValue ?inputInstance] and
          ?x[hasWebService hasValue ?ws] and
          ?x[endTime hasValue ?xEndTime] and
          ?xEndTime memberOf InvokeWebServiceFailed and
          naf ?x[after hasValue ?z] and
          naf ?x[before hasValue ?m]
        do
          invokeWS(?ws, ?inputInstance)
      endForall

      forall {?x, ?y}
        with
          ?x memberOf InvokeWebServiceActivity and
          ?y memberOf InvokeWebServiceActivity and
          ?x[hasWebService hasValue ?ws] and
          ?x[hasInputInstances hasValue ?inputInstance] and
          ?y[hasWebService hasValue ?ws] and
          ?x != ?y and
          ?x[hasSessionID hasValue ?sId] and
          ?y[hasSessionID hasValue ?sId] and
          ?x[endTime hasValue ?xEndTime] and
          ?y[endTime hasValue ?yEndTime] and
          ?xEndTime memberOf InvokeWebServiceFailed and
          ?yEndTime memberOf InvokeWebServiceFailed and
          ?x[before hasValue ?y] and
          naf ?x[after hasValue ?z] and
          naf ?y[before hasValue ?m]
        do
          discoverWS(?goal, ?wsNew)
          invokeWS(?wsNew, ?inputInstance)
      endForall
```

Listing 2. Implementation of the reactive behavior used to recover from service invocation failures

6 Conclusions and Future Work

In this paper we presented an approach towards semantically enabled monitoring and self-management of Semantic Execution Environments. Our solution is based on a set of ontologies (namely *Time*, *Event* and *Domain*) which provides a unified space for registering and reasoning about run-time service broker behavior. The components are represented as *Semantic Web Services* which enables comprehensive and formal description of their capabilities and behavior. Representation of the reactive behavior is assigned to Abstract State Machines, which represent yet another formal state-based approach to model a system behavior in a concise and expressive manner. Usage of the approach has been exemplified in a case study related to the Semantic Execution Environment. The study showed an elegant yet powerful way to define and execute reactive behavior in cases of service invocation failure, which indicates the value of the presented approach.

In the future, we will concentrate on integrating a fast Complex Event Processing (CEP) engine to raise the overall system performance by lifting the load of complex event detection from the ASM engine (i.e. instead of using a reasoner). Additionally, for the use case, we plan to enrich WSMX brokers with a

management interface that supports timed events in order to be able to generate periodic events on behalf of the managed component. This will allow periodic progress checking of the complex computations (e.g. discovery running over a large set of services) and timed notifications (e.g. taking results of the discovery before completion of the process over the entire set of registered services).

Acknowledgments. The work published in this paper is funded by the E.C. through the COIN IP (Project No. 216256). The authors wish to acknowledge the Commission for their support.

References

1. Fensel, D., Lausen, H., Pollers, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J. (eds.): Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer, Heidelberg (2007)
2. Oberle, D. (ed.): Semantic Management of Middleware. Springer Science + Business and Media, Heidelberg (2005)
3. Fensel, D., Kerrigan, M., Zaremba, M. (eds.): Implementing Semantic Web Services: The SESA Framework. Springer, Heidelberg (2008)
4. Vaculin, R., Sycara, K.: Specifying and Monitoring Composite Events for Semantic Web Services. In: Proceedings of the 5th IEEE European Conference on Web Services (November 2007)
5. Pedrinaci, C., Domingue, J., Alves de Medeiros, A.K.: A Core Ontology for Business Process Analysis. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 49–64. Springer, Heidelberg (2008)
6. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM 26(11), 832–843 (1983)
7. Borger, E., Stark, R. (eds.): Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
8. Scicluna, J., Polleres, A., Roman, D.: Ontology-based Choreography and Orchestration of WSMO Services. Technical Report D14v0.2, Semantic Technology Institute (STI) Innsbruck (2006)
9. Norton, B., Kerrigan, M., Mocan, A., Carenini, A., Cimpian, E., Haines, M., Scicluna, J., Zaremba, M.: Reference Ontology for Semantic Service Oriented Architectures. Technical report, OASIS Semantic Execution Environment TC (2008)
10. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and models of concurrent systems, pp. 477–498. Springer, New York (1985)
11. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
12. Veanes, M., Campbell, C., Schulte, W., Kohli, P.: On-The-Fly Testing of Reactive Systems. Technical Report MSR-TR-2005-05, Microsoft Research (2005)
13. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite Events for Active Databases: Semantics, Contexts and Detection. In: VLDB 1994: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 606–617. Morgan Kaufmann Publishers Inc., San Francisco (1994)
14. Gehani, N.H., Jagadish, H.V., Shmueli, O.: Composite event specification in active databases: Model and implementation. In: Proceedings of the 18th VLDB Conference, pp. 327–338 (1992)

15. Schmidt, K.U., Anicic, D., Sthmer, R.: Event-driven Reactivity: A Survey and Requirements Analysis. In: Proceedings of the 3rd international Workshop on Semantic Business Process Management at 5th European Semantic Web Conference (2008)
16. Paschke, A.: ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language. In: Proceedings of Int. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML 2006), Athens, Georgia, USA (November 2006)
17. Wang, Q., Liu, Y., Li, M., Mei, H.: An Online Monitoring Approach for Web services. In: COMPSAC 2007: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Washington, DC, USA, pp. 335–342. IEEE Computer Society, Los Alamitos (2007)