Error-Detection in Enterprise Application Integration Solutions

Rafael Z. Frantz[1], Rafael Corchuelo[2], and Carlos Molina-Jiménez[3]

[1] Dep. de Tecnologia, UNIJUÍ University
Rua do Comércio 3000, Ijuí 98700-000, RS, Brazil
rzfrantz@unijui.edu.br

[2] Dep. de Lenguajes y Sistemas Informáticos, Universidad de Sevilla
Avda. Reina Mercedes, s/n, Sevilla 41012, Spain
corchu@us.es

[3] School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom
carlos.molina@ncl.ac.uk

Abstract. Enterprise Application Integration (EAI) is a field of Software Engineering. Its focus is on helping software engineers integrate existing applications at a sensible costs, so that they can easily implement and evolve business processes. EAI solutions are distributed in nature, which makes them inherently prone to failures. In this paper, we report on a proposal to address error detection in EAI solutions. The main contribution is that it can deal with both choreographies and orchestrations and that it is independent from the execution model used.

Keywords: Enterprise Application Integration; Error Monitoring; Error Detection; Dependability and Resilience.

1 Introduction

Companies are relying heavily on computer-based applications to run their businesses processes. Such processes must evolve and adapt as companies evolve and adapt to varying contextual conditions. Common problems include that the applications were not designed to facilitate integrating them with others, i.e., they do not provide a business level API, and that they were implemented using a variety of technologies that do not inter-operate easily [13]. The goal of Enterprise Application Integration (EAI) is to help reduce the costs of EAI solutions to facilitate the implementation and evolution of business processes.

Figure §1 sketches two sample EAI solutions that involve four applications and three integration processes. Note that a solution is only a logical means to organise a set of processes: different solutions can share the same processes, and a solution can contain another solution. The processes interact with the applications using the facilities they provide, e.g., an API in the best case, a user interface, a file, a database or other kinds of resources. They help implement message-based workflows to keep a number of applications' data in synchrony or

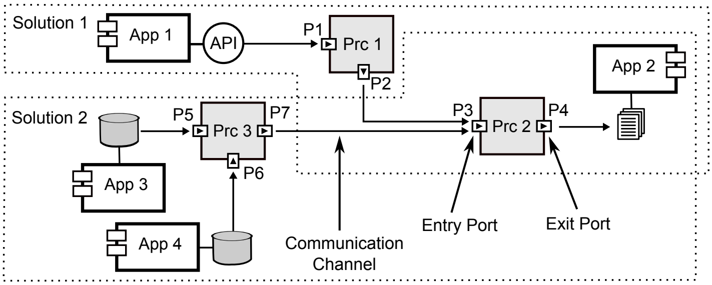

Fig. 1. Sample EAI solutions

to build new functionality on top of them. Processes use ports to communicate with each other or with applications over communication channels. Ports encapsulate reading from or writing to a resource, which helps abstract away from the details of the communication mechanism, which may range from an RPC-based protocol over HTTP to a document-based protocol implemented on a database.

The Service Oriented Architecture initiative has gained importance within the field of EAI, since it provides appropriate technologies to wrap applications and to implement message workflows. Centralised workflows, aka orchestrations, rely on a single process that helps co-ordinate a workflow of messages amongst a number of other processes and applications; contrarily, decentralised workflows, aka choreographies, do not rely on such a central co-ordinator. The tools used to implement workflows include conventional systems [8], others based on BPEL [15], and others like BizTalk [5], or Camel [10].

EAI solutions are distributed in nature, since they involve several applications and processes that may easily fail to communicate with each other [8], which argues for real-world EAI solutions to be fault-tolerant. There seems to be a general consensus that the provisioning fault-tolerance includes the following stages: event reporting, error monitoring, error diagnosing, and error recovery. Event reporting happens when processes report that they have read or written a message by means of a port; the goal of error monitoring is to analyse traces of events to find invalid correlations, i.e., anomalous sets of messages that have been processed together; such correlations must later be diagnosed to find the cause of the anomalies, and appropriate actions to recover from the error must be taken in the error recovery stage.

Orchestration workflows rely on an external mechanism that analyses inbound messages, correlates them, and starts a new instance of the orchestration whenever a correlation is found. The typical execution model is referred to as process-based since a thread must be allocated to run a process on a given correlation; contrarily, the task-based execution model relies on a pool of threads that are allocated to the tasks. Simply put, in the process-based model threads remain allocated to a process even if that process is waiting for the answer to a request to another process; contrarily, in the task-based model, no thread shall be idle as long as a task in a process is ready for execution.

In this paper, we report on a proposal to build an error monitor for EAI solutions. The key contribution is that it works with both orchestrations and choreographies, and that it is independent from the execution model used. In Section §2, we report on other proposals in the literature; in Section §3, we present an overview of our proposal; in Section §4, we delve into our proposal to detect errors; finally, we present our conclusions in Section §5.

2 Related Work

Error detection is relatively easy in orchestration systems because either correlations are found prior to starting an orchestration process and everything happens within the boundaries of this process. Contrarily, in choreographies, a correlation may typically involve several processes that run in total asynchrony, and there is not a single point of control; furthermore, EAI solutions may overlap since it is common that processes are reused across several business processes. This makes it more difficult to endow choreographies with fault-tolerance capabilities.

The research on fault tolerance that has been conducted by the workflow community is closely related to our work. Chiu and others [4] presented an abstract model for workflows with embedded fault-tolerance capabilities; it set the foundations for other proposals in this field. Hagen and Alonso [8] presented a proposal that builds on the two-phase commit protocol, and it is suitable for orchestrations in which the execution model is process-based. Alonso and others [1] provided additional details on the minimum requirements to deal with fault tolerance in orchestrated systems. Liu and others [12] discussed how to deal with fault tolerance in settings in which recovery actions are difficult or infeasible to implement; the authors also assume the existence of a centralised workflow engine, i.e., they also focus on orchestrations. Li and others [11] reported on a theoretical solution that is based on using Petri nets; they see processes as if they were controllers, and report on detecting some classes of errors by means of linear parity checks; the key is that they focus on systems in which a fault can involve an arbitrarily large number of correlated messages, which are consumed and produced by distributed processes, but are assume that they are choreographed by a central processor. An architecture for fault-tolerant workflows, based on finite state machines that recognise valid sequences of messages was discussed in [6]; this proposal is suitable for both orchestrated and choreographed processes; however it is aimed at process-based executions.

The study of fault tolerance in the context of choreographies has been paid less attention in the literature. Chen and others [3] presented a proposal that deviates from the previous ones in that their results can be applied to both orchestrations and choreographies. They assume that the system under consideration is organised into three logical layers (front-end, application server, and database server), plus an orthogonal layer (the logging system). Since they can deal with choreographies, they need to analyse message traces to detect errors. They assume that each message has a unique identifier that allows to trace it throughout the execution flow; unfortunately, they cannot deal with EAI solutions in which messages are split or aggregated, since this would require to find correlations amongst messages, which is not supported at all. Due to this limitation, it can easily deal with both process- and task-based execution models. Yan and Dague [16] suggested to re-use the body of knowledge about error detection in industrial discrete event systems, in error detection in web services applications; they discussed runtime error detection of orchestrated web services; a salient feature of this proposal is that, similarly to [14], the authors assume that failure events are not observable; the granularity of execution in this approach is at process level. Baresi and others [2] discussed some preliminary ideas for building an error monitor that can be used for both orchestrated and choreographed processes. No implementation or evaluation was provided.

Our analysis of the literature reveals most authors in the EAI field focus on orchestrations and the process-based execution model; choreographies and the task-based execution model have been paid little attention so far. Another conclusion is that the distinction amongst the stages required to provision fault tolerance is often blurred. The reason is that many proposals focus on error recovery since error detection or error diagnosing is quite a trivial task. In many proposals, the presence of an error can be derived from a single event. For instance, the conventional try-catch mechanism involves the notification of a single event to be caught by the exception mechanisms [7]. However, there is a large class of applications in which the presence of an error can only be deduced from the analysis of traces of events that are related to each other, e.g., by order, parent-child relationships, propagation, or causations. Error detection in these cases is a challenging problem, in particular, when the number of events is large.

3 Overview of Our Proposal

Our proposal builds on a monitor to which each port must report events, and a set of rules that help determine if correlations are valid or not. A monitor is composed of three modules called Registrar, Event Handler, and Error Detector, three databases called Descriptions Database, Graphs Database, and History Database, and a queue called Graphs Queue. Figure §2 presents the abstract model for them.

The Registrar module is responsible for maintaining the Descriptions Database up to date. This database provides the other modules a description of the solutions, processes, ports, and rules the monitor handles. (Note that we use term 'artefact' to refer to both solutions and processes.)

The Event Handler uses the events reported by ports to update the Graphs Database and the Graphs Queue. An event can be of type Reception, which happens at ports that read data from an application (either successfully or unsuccessfully) and other ports that fail to read data at all, Shipment, which occurs when a port writes information (either successfully or unsuccessfully), and Transfer, which happens when a port succeeds to read data that was written previously by another port. Every event has a target binding and zero, one, or more source bindings. We use this term to refer to the data involved in an event, namely: the instant when the event happened, the name of the port, the identifier of the message read or written, and a status, which can be either Status::OK to mean that

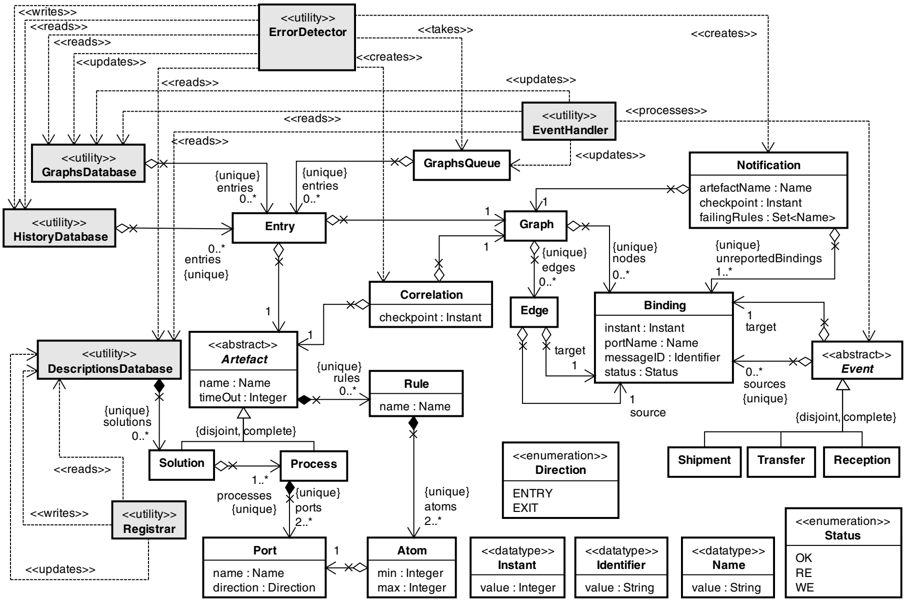

Fig. 2. Model for Registrar, Event Handler and Error Detector modules

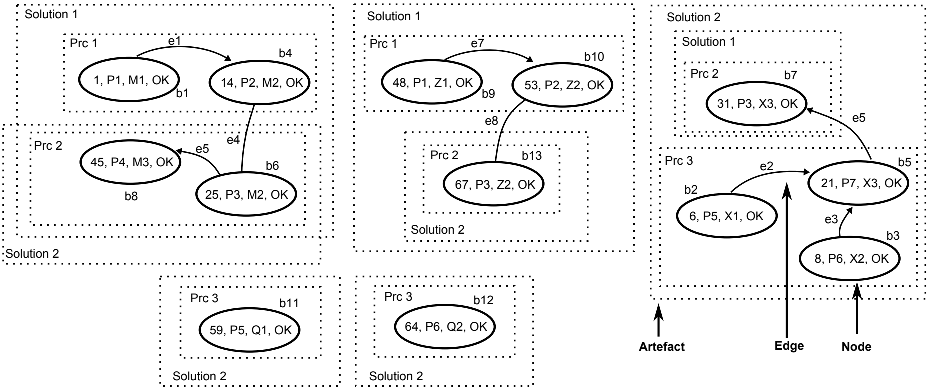

Fig. 3. Sample Graphs Database

no problem was detected, Status::RE to mean that there was a reading failure, or Status::WE to mean that there was a writing failure.

The Graphs Database stores an entry per artefact in the Descriptions Database; such entries contain a graph that the Event Handler builds incrementally, as it receives events. Figure §3 shows a sample Graphs Database for the EAI solution in Figure §1. For instance, let us focus on bindings b_6 and b_4: the former is involved in process Prc$_2$ and both solutions, and it denotes that port P$_3$ dealt with message M_2 at instant 25, and that the result was successful; the later is involved in process Prc_1 and $\mathsf{Solution}_1$ only, and it indicates that port P_2 dealt with message M_2 at instant 14, and that the result was successful; besides, the edge between them both indicates that binding b_6 originates from binding b_4.

The Graphs Queue is used to refer to the entries in the Graphs Database that have changed since the database was analysed for the last time. This helps minimise the work performed by the Error Detector, whose abstract model is presented in Figure §2. Note that it is relatively easy to find correlations in a graph like the one in Figure §3 since this task amounts to finding the connected components of the graph [9]. Contrarily, verifying them depends completely on the semantics of the EAI solutions involved. This is why we assign each artefact an upper bound to the total amount of time it is expected to take to produce a valid correlation, i.e., a time out, and a set of rules of the following form, cf. Figure §2:

$$\mathsf{P}_1[m_1..n_1],\ \ldots,\ \mathsf{P}_p[m_p..n_p] \rightarrow \mathsf{Q}_1[r_1..s_1],\ \ldots,\ \mathsf{Q}_q[r_q..s_q],$$

where P_i and Q_j are port names and m_i, n_i, r_i, s_i denote the minimum and maximum number of messages a correlation allows in each port so that it can be considered valid. For instance, a rule like $\mathsf{P}_5[1..1]$, $\mathsf{P}_6[1..1] \rightarrow \mathsf{P}_4[1..10]$ regarding $\mathsf{Solution}_2$ in Figure §1 means that it is a requirement for a correlation to be considered valid that it has one message at port P_5, one message at port P_6 and then 1–10 messages at port P_4.

The correlations found by the Error Detector module are removed from the Graphs Database and stored in the History Database. This helps us complete them if new messages are reported later.

4 Detecting Errors

Due to space limitations, we do not provide additional details on the Registrar or the Event Handler modules. Instead, we focus on the Error Detector, which is the central module. Its algorithm is as follows:

```
to detectErrors() do
    repeat
        $s$ = findCorrelations()
        for each correlation $c$ in $s$ do
            verifyCorrelation($c$)
        end for
    end repeat
end
```

It runs continuously; in each iteration, it first finds a set of correlations and then verifies them sequentially. In the following subsections, we delve into the algorithms to find correlations and to verify them.

4.1 Finding Correlations

The algorithm to find correlations is as follows:

```
 1: to findCorrelations(): Set(Correlation) do
 2:     take entry $f$ from the Graphs Queue
 3:     $checkpoint$ = getTime()
 4:     $s$ = find connected components of $f.graph$
 5:     $result = \emptyset$
 6:     for each graph $g$ in $s$ do
 7:         $c$ = new Correlation(artefact = $f.artefact$, graph = $g$, checkpoint = $checkpoint$)
 8:         add $c$ to $result$
 9:     end for
10: end
```

This algorithm starts by taking an entry f from the Graphs Queue at line §2; if there is not an entry available, then we assume that the algorithm blocks here until an entry is available. Note that the core of the algorithm is line §4, in which we find the connected components of the graphs that corresponds to the entry we have taken from the Graphs Queue.

4.2 Verifying Correlations

A correlation can be diagnosed as on-going, valid or invalid. A correlation is on-going if its deadline has not expired yet. Bear in mind that correlations are analysed within the context of an artefact, which must have an associated time out and set of rules. The deadline for a correlation is defined as the time of its earliest binding plus this time out. This provides a time frame within which all of the messages involved in the correlation are expected to be reported. A correlation is valid if all of the messages it involves were read or written by the expected deadline, there was no reading or writing failure, and all of the rules involved are passed; otherwise, it is considered invalid and a notification must be generated so that it can be diagnosed and appropriate recovery actions can be executed later. The algorithm to verify a correlation is as follows:

```
to verifyCorrelation(in $c$: Correlation) do
    findCompletion($c$, out $completedGraph$, out $unnotifiedBindings$)
    $status$ = every binding $b$ in $completedGraph.nodes$ has $status$ OK?
    $earliestInstant$ = minimum of $completedGraph.nodes.instant$
    $latestInstant$ = maximum of $completedGraph.nodes.instant$
    $deadline = earliestInstant + c.artefact.timeOut$
    $notPassedRules$ = checkRules($completedGraph$, $c.artefact$)
    $isValid$ = $deadline <= c.checkpoint$ and $latestInstant <= deadline$ and
                $status ==$ true and $notPassedRules == \emptyset$
    $isInvalid$ = ($deadline < latestInstant$) or
                ($deadline <= c.checkpoint$ and (not $status$ or $notPassedRules \neq \emptyset$))
    if $isValid$ then
        $f$ = find the entry for $c.artefact$ in the Graphs Database
        remove $c.graph$ from $f.graph$
        $g$ = new Entry(artefact = $c.artefact$, graph = $c.graph$, isValid = $true$)
        add $g$ to History database
    elsif $isInvalid$ then
        $f$ = find the entry for $c.artefact$ in the Graphs Database
        remove $c.graph$ from $f.graph$
        $g$ = new Entry(artefact = $c.artefact$, graph = $completedGraph$, isValid = $false$)

        add $g$ to History database
        $n$ = new Notification( artefactName = $c.artefact.name$, graph = $completedGraph$,
                                unnotifiedBindings = $unnotifiedBindings$, checkpoint = $c.checkpoint$,
                                notPassedRules = $notPassedRules$)
        send $n$ to the notification port of the monitor
    elsif
        - Nothing to do, since $c$ is an on-going correlation
    end if
end
```

The algorithm gets a Correlation c as input; the first thing it has to do is to complete it with the help of the History Database. Note that correlations that are not on-going are removed from the Graphs Database; due to the asynchronous nature of EAI solutions, that implies that a after a correlation is verified, additional correlated messages may be reported. This is also the reason why before verifying a correlation, it must be completed using the History Database. Algorithm findCompletion, which is explained later, performs this tasks; given a correlation c, it returns a graph that includes $c.graph$ and additional nodes and edges found in the History Database, as well the subset of bindings in the completed correlation that have not been notified, yet.

4.3 Completing Correlations

The algorithm to complete a correlation is as follows:

```
to findCompletion(in $c$: Correlation, out $completedGraph$: Graph,
                      out $unnotifiedBindings$: Set(Binding) ) do
    $completedGraph$ = new Graph(nodes = shallow copy of $c.graph.nodes$,
                                 edges = shallow copy of $c.graph.edges$)
    $unnotifiedBindings$ = shallow copy of $c.graph.nodes$
    $s$ = find all of the entries for $c.artefact$ in the History database
    for each entry $f$ in $s$ do
        $intersection = c.graph.nodes \cap f.graph.nodes$
        if $intersection \neq \emptyset$ then
            merge $f.graph$ into $completedGraph$
            if not $f.isValid$ then
                remove $intersection$ from $unnotifiedBindings$
            end if
        end if
    end for
end
```

This algorithm takes a correlation c as input and returns a graph that is a completed version of $c.graph$ and a set of bindings that have not been notified so far. It first creates an initial completed graph at line §3 from a shallow copy of the nodes and the edges of the graph of correlation c. A shallow copy is made because otherwise line §10 would modify the original graph in correlation c. The loop at lines §7–§15 discovers if there are common bindings between correlation c and an entry f. If common bindings are found they are merged into the resulting completed graph, and bindings that were detected to be already in the graph of entry f are removed from the set of unnotified bindings, leaving only new bindings that were not reported yet. Note that this is done only if graph f represents an invalid graph; otherwise all bindings are new.

4.4 Checking Rules

The algorithm to check rules is as follows:

```
to checkRules(in $g$: Graph, in $t$: Artefact): Set(Name) do
    $result = \emptyset$
    for each rule $r$ in $t.rules$ do
        for each atom $a$ in $r.atoms$ do
            $n$ = count bindings $b$ in $g.nodes$ such that $b.portName == a.port.name$
            if $n < a.min$ or $n > a.max$ then
                add $r.name$ to $result$
            end if
        end for
    end for
end
```

This algorithm takes a graph that represents a correlation and an artefact as input; it returns the subset of rules associated with the artefact that the correlation does not pass. The algorithm is simple since we just need to count the number of bindings that involve the port referenced in the atom; if this figure is not within the margins that the atom establishes, then it is added to the result of the algorithm since that rule is not passed.

5 Conclusions

In this paper, we have presented a proposal to detect errors in the context of EAI solutions. It is novel in that it is not bound to orchestrations or choreographies, neither to a process- nor a task-based execution model; it is totally independent. We have analysed the time complexity of our algorithm and we have proved that it is $O(b + c\,h)$, where b denotes the average number of bindings that have been reported by means of events since the last checkpoint, c denotes the average number of correlations found at each checkpoint, and h the average number of entries for an artefact in the History Database. Note that b and c are expected to vary within some margins as long as the Descriptions Database does not change; contrarily, h increases monotonically as time goes by. This implies that after a point in time, this complexity is dominated by h, i.e., the algorithm behaves linearly in the average number of entries per artefact in the History Database. Recall that the only purpose of this database is to complete correlations that are found in the Graphs Database, just in case a message is processed by a port after the deadline for the corresponding correlation expires. In practice, it makes sense to remove old information from the database periodically, say a week; this puts an upper bound to the size of the History Database, which, in turn, puts an upper bound to the total time the algorithm may take to detect errors.

References

[1] Alonso, G., Hagen, C., Divyakant, D., Abbadi, A.E., Mohan, C.: Enhancing the fault tolerance of workflow management systems. IEEE Concurrency 8(3), 74–81 (2000)

[2] Baresi, L., Guinea, S., Kazhamiakin, R., Pistore, M.: An Integrated Approach for the Run-Time Monitoring of BPEL Orchestrations. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 1–12. Springer, Heidelberg (2008)

[3] Chen, M., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E.: Path-based faliure and evolution management. In: Int'l Symp. Netw. Syst. Des. and Impl., p. 23 (2004)

[4] Chiu, D., Li, Q., Karlapalem, K.: A meta modeling approach to workflow management systems supporting exception handling. Inf. Syst. 24(2), 159–184 (1999)

[5] Dunphy, G., Metwally, A.: Pro BizTalk 2006. Apress (2006)

[6] Ermagan, V., Kruger, I., Menarini, M.: A fault tolerance approach for enterprise applications. In: IEEE Int'l Conf. Serv. Comput., vol. 2, pp. 63–72 (2008)

[7] Goodenough, J.: Exception handling: Issues and proposed notation. Communications of the ACM 18(12), 683–696 (1975)

[8] Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Trans. Softw. Eng. 26(10), 943–958 (2000)

[9] Hopcroft, J.E., Tarjan, R.E.: Efficient algorithms for graph manipulation. Communications ofthe ACM 16(6), 372–378 (1973)

[10] Ibsen, C., Anstey, J.: Camel in Action. Manning Publications (2010)

[11] Li, L., Hadjicostis, C., Sreenivas, R.: Designs of bisimilar petri net controllers with fault tolerance capabilities. IEEE Trans. Syst. Man Cybern. Part A: Syst. Humans 38(1), 207–217 (2008)

[12] Liu, C., Orlowska, M., Lin, X., Zhou, X.: Improving backward recovery in workflow systems. In: Int'l Conf. Database Syst. Adv. Appl., p. 276 (2001)

[13] Messerschmitt, D., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press, Cambridge (2003)

[14] Sampath, M., Sengupta, R., Lafortune, S.: Failure diagnosis using discrete-event models. IEEE Trans. on Control Syst. Technol. 4(2), 105–124 (1996)

[15] Wright, M., Reynolds, A.: Oracle SOA Suite Developer's Guide. Packt Publishing (2009)

[16] Yan, Y., Dague, P.: Modeling and diagnosing orchestrated web service processes. In: IEEE Int'l Conf. on Web Serv., pp. 51–59. IEEE Computer Society, Los Alamitos (2007)