The Journal of Systems and Software 85 (2012) 480–497

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A proposal to detect errors in Enterprise Application Integration solutions

Rafael Z. Frantz[a,*], Rafael Corchuelo[b], Carlos Molina-Jiménez[c]

[a] UNIJUÍ University, Department of Technology, Rua do Comércio, 3000, Ijuí 98700-000, RS, Brazil
[b] Universidad de Sevilla, ETSI Informática, Avda. de la Reina Mercedes, s/n, Sevilla E-41012, Spain
[c] Newcastle University, School of Computing Science, Newcastle Upon Tyne, NE1 7RU, United Kingdom

ARTICLE INFO

Article history:
Received 31 May 2010
Received in revised form 25 October 2011
Accepted 26 October 2011
Available online 25 November 2011

Keywords:
Enterprise Application Integration
Fault-tolerance
Error monitoring
Error detection

ABSTRACT

Enterprise Application Integration (EAI) solutions comprise a set of specific-purpose processes that implement exogenous message workflows. The goal is to keep a number of applications' data in synchrony or to develop new functionality on top of them. Such solutions are prone to errors because they are highly distributed and usually involve applications that were not designed with integration concerns in mind. This has motivated many authors to work on provisioning EAI solutions with fault-tolerance capabilities. In this article we analyse EAI solutions from two orthogonal perspectives: viewpoint (orchestration versus choreography) and execution model (process- versus task-based model). A review of the literature shows that current proposals are bound to a specific viewpoint or execution model or have important limitations. To address the problem, we have devised an error monitor that can be used to provision EAI solutions with fault-tolerance capabilities. Our theoretical analysis proves that the algorithms we use are computationally tractable, and our experimental results prove that they are efficient enough to be used in situations in which the workload is very high.

1. Introduction

The computer infrastructure of a typical today's enterprise can be seen as a software ecosystem that involves several complementary applications purchased from different providers or built at home (Messerschmitt and Szyperski, 2003). A recurrent challenge is to make these applications inter-operate with each other to keep their data synchronised or to create a new piece of functionality (Hohpe and Woolf, 2003). This problem is known as Enterprise Application Integration (EAI).

A typical EAI solution consists of one or more processes that interact with each other and with the existing applications by means of ports that read/write messages from/to communication channels. Roughly speaking, they implement an exogenous workflow in which messages are read from a subset of applications, routed through the processes, which may transform them and write the results to another subset of applications. Ports abstract away from the details of a specific communication mechanism, which may range from an RPC-based protocol over HTTP to a document-based protocol implemented on a database management system. Processes build on tasks to filter, enrich, split, aggregate, route or transform messages, to name a few (Hohpe and Woolf, 2003). It is common that solutions share processes, which makes them overlap or even include others. Fig. 1 shows two solutions that we shall use throughout the article to illustrate our proposal. In this example, there are two overlapping solutions, namely: Solution1, which integrates applications App1 and App2 by means of processes Prc1 and Prc2, and Solution2, which integrates applications App3, App4 and App2 by means of processes Prc3 and Prc2. Note that process Prc2 is shared by both solutions.

EAI solutions can be characterised from several perspectives. In this article, we focus on viewpoint and execution model.

By viewpoint, we refer to whether a solution is specified as an orchestration or as a choreography. It is an orchestration if there is a single process that co-ordinates every exchange of messages (Peltz, 2003). This process plays the role of a centralised point of control that can consequently have an accurate global view of the current state of execution. This is a valuable piece of information that can be used, for instance, to identify the applications and processes that are involved in an error. Contrarily, an EAI solution is a choreography if there is not a centralised point of control, i.e., applications and processes interact in a peer-to-peer fashion; this consequently implies that no process can have an accurate global view of the execution (Peltz, 2003). Currently, WS-BPEL (OASIS, 2007) and WS-CDL (W3C, 2005) are the de facto standards to specify orchestrations and choreographies, respectively.

Regarding the execution model, it is worth mentioning that EAI solutions must be deployed to a run time system that provides an execution engine. Depending on the granularity of execution,

* Corresponding author.
E-mail addresses: rzfrantz@unijui.edu.br (R.Z. Frantz), corchu@us.es (R. Corchuelo), carlos.molina@ncl.ac.uk (C. Molina-Jiménez).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.10.048

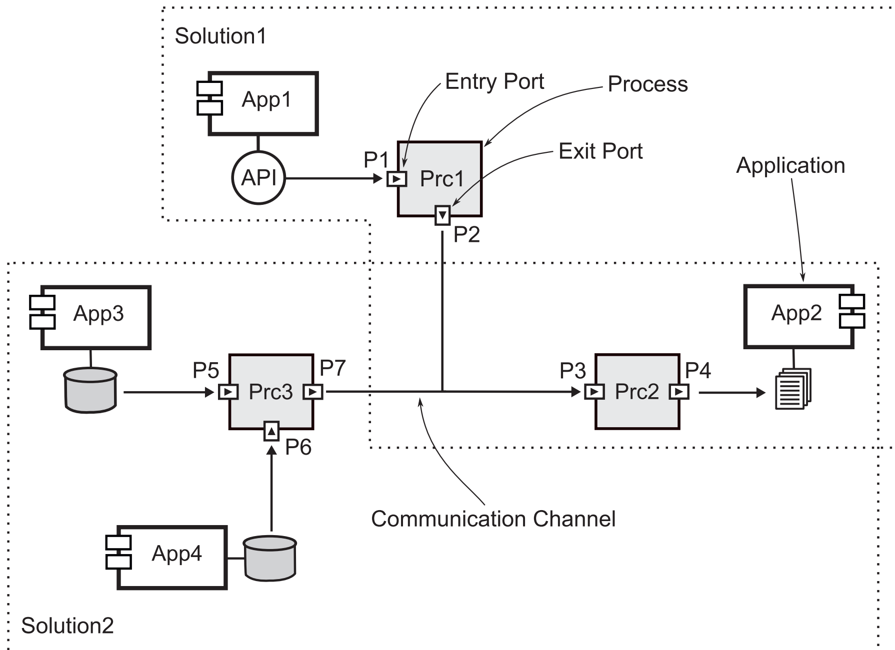

Fig. 1. Sample integration solutions.

we distinguish between the process- and the task-based execution models. In the process-based model, the engine controls process instances as a whole, i.e., there is no means that it can interact with the internal tasks; contrarily, in the task-based model, the engine may control both process instances and their internal tasks. The implication is that the process-based execution model requires a queue to store messages, a mechanism to correlate them, and decide when a new process instance can be started; furthermore, each process instance requires a thread to be allocated exclusively, which may have a negative impact on performance in cases in which a process instance sends a request and has to wait for a long time before it gets the answer (for instance, think of a request that requires the intervention of a person or a configuration that assigns low priorities to request that come from integration solutions, Hagen and Alonso, 2000). Since the task-based execution model deals with the tasks inside a process instance, it can allocate threads more efficiently; in other words, no thread shall be idle as long as there is a task ready to be executed, independently from the process to which this task belongs.

EAI solutions are inherently distributed; they are thus vulnerable to a variety of errors that can have an impact on their normal behaviour. Errors are due to faults, which can be either permanent, e.g., due a software defect, or transient, e.g., due to a resource that is temporarily unavailable. Errors that are not dealt with properly are perceived as failures by end users (Campbell and Randell, 1986; Avizienis et al., 2004). Fault-tolerance proposals aim to help keep systems delivering their functionality in spite of faults. Typically, they can be modelled as a pipeline that goes through the following stages: event reporting, error monitoring, error diagnosing, and error recovering. The event reporting stage deals with reporting whether a port was able to handle a message or not. In the error monitoring stage, events are stored and analysed to find correlations that shall later be checked for validity. When an error is detected, a notification is created and sent to the error diagnosing stage, whose aim is to identify the cause of the error, the messages and the parties involved. The error recovering stage attempts to execute recovery actions to help the system compensate for the existence of faults and the occurrence of errors.

In the literature there are several proposals that deal with fault tolerance (Zeng et al., 2005; Liu et al., 2007, 2006, 2001; Erradi et al., 2006; Chiu et al., 1999; Hagen and Alonso, 2000; Alonso et al., 2000; Chen et al., 2004; Ermagan et al., 2008; Baresi et al., 2008; Li et al., 2009, 2008; Borrego et al., 2010; Yan and Dague, 2007; Yan et al., 2005). We have analysed them from the viewpoint and execution model perspectives. Our conclusion is that most of them aim at the orchestration viewpoint and the process-based execution model, since they focus on Web Services, WS-BPEL, and traditional workflow systems. There are a few exceptions that take the choreography viewpoint and/or the task-based execution model into account, but they have important limitations that make them difficult to use in a general context. The main limitation of Chen et al. (2004) is that processes can have only one input and messages cannot be split or aggregated inside the workflow; these are common tasks in EAI solutions. In proposals (Ermagan et al., 2008; Baresi et al., 2008), although the authors suggest that it can be applied to the choreography viewpoint, no implementation or evaluation is provided, which makes them difficult to assess and apply in practice. Finally, the limitation of proposal (Li et al., 2008) is that it is theoretical and transforming it into a practical tool that deals with distribution problems in a software ecosystem does not seem straightforward.

In this article, we focus on the error monitoring stage. We deal with faults that can happen in the communication mechanism or during the processing of messages, e.g., structural and deadline errors. Our proposal is sketched in Fig. 2. It relies on a so-called Meta-information database that stores meta-information about the solutions being monitored, e.g., which processes and ports are involved in a solution. Note that this database is external to the monitor since it is intended to be shared with other stages of the fault-tolerance pipeline. The monitor itself is composed of two subsystems and two databases, namely: Event Handler, Error Detector, Work Graph and Work Queue. The Event Handler handles events that inform about a port reading or writing a message, either successfully or unsuccessfully. It uses the events to build a graph structure that is stored in the Work Graph database; this graph keeps track of the messages processes exchange and their parent–child relationships. The Error Detector analyses this graph to find and verify

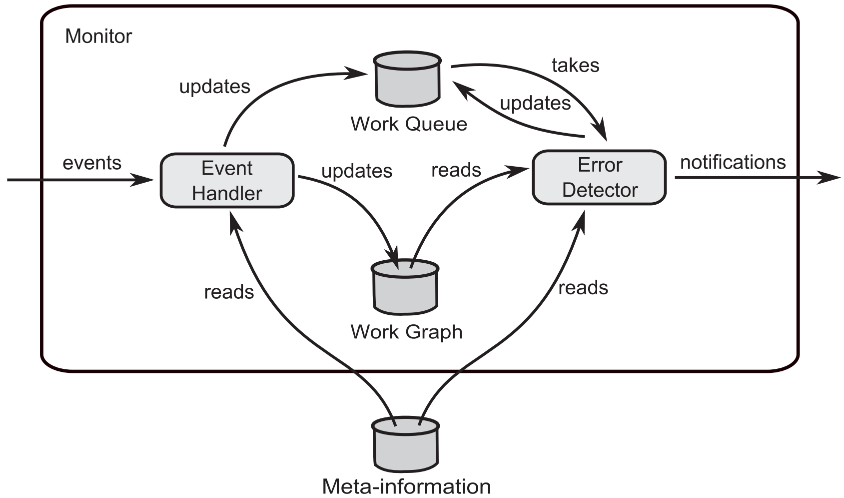

Fig. 2. Abstract view of the monitor.

correlations. The Work Queue database is used as an intermediate buffer that allows the Event Handler and the Error Detector to work in total asynchrony. Every time the Event Handler processes an event, it stores a piece of information in the Work Queue database; this information instructs the Error Detector to analyse the Work Graph database at a specific point in time in order to find the correlation in which a specific message is involved. To verify correlations, the Error Detector builds on both built-in and user-defined rules; the former allows to detect communications or deadline errors; the latter allows to detect structural errors that depend on the semantics of a given process or solution, i.e., correlations that lack messages or have more messages than expected. The only assumption we make is that the clock resolution of the monitor is enough to distinguish between every two messages that are read or written in a row; in other words, we can distinguish between multiple events that involve the same message at the same port. In practice, this is not a shortcoming since current clock resolutions are in the order of nanoseconds, whereas reading or writing to a port usually requires more time.

Our main contribution is that our proposal is not bound to a particular viewpoint and/or execution model; neither imposes it any practical limitations. We have analysed our proposal from a theoretical point of view, and we have proved that the algorithms on which it relies are computationally tractable; furthermore, we have carried out a series of experiments that prove that it performs quite well under heavy workloads. These results improve on a previous proposal (Frantz et al., 2011) in which we devised another error monitoring system whose error detection complexity depended on the size of a history database, which increased monotonically; our current solution does not depend at all on such a database, which consequently makes it is more scalable.

The rest of the article is structured as follows: Section 2, discusses the related work; the Meta-information database shared by all of the stages of the fault-tolerance pipeline is presented in Section 3; Section 4, reports on the Event Handler; Section 5, reports on the Error Detector; the time complexity analysis of our algorithms is presented in Section 6; the experiments are introduced in Section 7; and, finally, we draw our conclusions in Section 8.

2. Related work

The literature distinguishes between static and run-time fault tolerance proposals. As discussed in Wu et al. (2006), static analysis is concerned with the detection of logical errors in the specification of a system; for instance, race condition errors that emerge from overlooking data dependencies. In this proposal, the authors use a directed acyclic graph to specify the synchronisation constraints imposed on the execution of individual tasks of business processes; they then convert the graph into a coloured Petri net and analyse it to detect potential synchronisation errors. In contrast, we assume that the EAI solutions with which we deal are logically consistent; we then focus on errors that emerge during their execution due, for instance, to delayed or missing messages. The work on static analysis is orthogonal to ours; therefore, it falls out of the scope of this article.

We have realised that the distinction between the error detection and the error recovery stages is blurred in many proposals. This is common in cases in which the complexity of the error detection algorithm is trivial or abstracted away from the user's view by means of another mechanism. In our proposal, we make a clear distinction between error detection and error recovery.

In some cases, the presence of an error can be determined from the analysis of a single event, e.g., the widely used try-catch mechanism falls within this category (Goodenough, 1975). These cases fall outside the scope of our research. We are interested in cases in which it is necessary to process traces of events that are related to each other. Traces like these are prone to propagate faulty data. Representative examples include discrete event-based systems, e.g., workflow systems (Hagen and Alonso, 2000), automatic controllers used in manufacturing processes (Li et al., 2008), and computer-based controllers used to automatically operate aircrafts, train traffic crossing and medical devices (Levenson, 1991). Error detection is a challenging problem in this context, in particular, when the number of events notified to the monitor is large, e.g., of the order of hundreds of events per second.

To place our proposal in context, it is worth clarifying that our solution builds on a centralised error monitor that can receive events from several EAI solutions concurrently. We are aware that some authors have pointed out that centralised error detectors do not scale well (Sayed Mouchaweh, 2010). Alternatively, they favour decentralised error detectors which are composed of two or more local error detectors that gather and process their own local information and collaborate to detect global errors. A similar idea to handle scalability is also suggested in Laguna et al. (2009), Khanna et al. (2007, 2009). Our reservations about decentralised error detectors is the complexity involved in the co-ordination and synchronisation of their activities, and the difficulty to deal with open systems in which processes and solutions are not pre-defined, but may be created and destroyed as time goes by; we leave this alternative out of our discussion on the basis that centralised error

Table 1
Summary of the related work.

Proposal	Orchestration	Choreography	Process-based	Task-based
Zeng et al. (2005)	+	–	+	–
Liu et al. (2007)	+	–	+	–
Erradi et al. (2006)	+	–	+	–
Liu et al. (2006)	+	–	+	–
Chiu et al. (1999)	+	–	+	–
Hagen and Alonso (2000)	+	–	+	–
Liu et al. (2001)	+	–	+	–
Alonso et al. (2000)	+	–	+	–
Chen et al. (2004)	+	+[a]	+	+[a]
Ermagan et al. (2008)	+	+[a]	+	–
Baresi et al. (2008)	+	+[a]	+	–
Li et al. (2009)	+	–	+	–
Borrego et al. (2010)	+	–	+	–
Li et al. (2008)	–	+[a]	–	+[a]
Yan and Dague (2007)	+	–	+	–
Yan et al. (2005)	+	–	+	–

[a] With important limitations that are discusses in the text.

detectors seem to satisfy our requirements well even in situations with high workloads, cf. Section 7.

A distinctive feature of our research is that we do not only suggest an error detector, but we also provide a rule-based language to express the expected behaviours of processes and solutions. The rules provide flexibility to our error detection mechanism. Finally, it is worth clarifying that the proposals that provide a rule-based language consider that the messages or events in the log are already correlated, presumably because they aim only at orchestrated EAI solutions. Since our proposal aims to be independent from the viewpoint and the execution model, our monitor accounts for the arrival of uncorrelated messages and then finds their correlations. A salient feature of our proposal is that it emphasises the relevance of evaluating the error detection algorithm, from several dimensions. This is an issue frequently overlooked in the current literature.

There are several proposals in the literature on fault-tolerance aiming to help keep systems delivering their functionality in spite of errors. Our conclusion is that most of them aim at the orchestration viewpoint and the process-based execution model, since they focus on Web Services, WS-BPEL, and traditional workflow systems. The few proposals that cover choreography and the task-based execution model have important limitations, which prevent them from being used in a general context. For example, in Chen et al. (2004) processes can have only one input and messages cannot be split or aggregated inside the workflow; in Ermagan et al. (2008), Baresi et al. (2008), no implementation or evaluation is provided, which makes it difficult to assess and apply in practice; Li et al. (2008) addresses only a single class of errors. Our proposal tackles the problem of endowing EAI solutions with a fault-tolerance mechanism that is independent from the viewpoint and the execution model; furthermore, it does not impose any practical limitations. Our failure semantics includes communication, structural and deadline errors.

In the following, we report on more details about the proposals we have summarised in Table 1.

Fault tolerance in business processes and web service composition has been addressed by several authors, but mainly with a focus on error recovery. For instance, in Zeng et al. (2005) the authors realise that service compositions are vulnerable to failures and produce different exceptions; they then show how Event-Condition-Action rules can be used to handle them at run time. The focus of this proposal is on error recovery. Error detection is mentioned in passing only, presumably, because their context does not require sophisticated error detection algorithms. A similar discussion on exception handling for WS-BPEL can be found in Liu et al. (2007), which is complementary to Zeng et al. (2005). In Erradi et al. (2006), the authors propose a policy-driven middleware solution to handle exceptions in web service compositions. With this article, we share the view that communication operations are the most error-prone. Relevant to us is also the failure semantics, which covers the communication failure types addressed in our proposal. An algorithm for the execution of WS-BPEL processes with relaxed transactions is presented in Liu et al. (2006). In Liu et al. (2007), the authors enhance their approach with a rule-based language for defining error recovery rules. It is worth emphasising that papers (Zeng et al., 2005; Erradi et al., 2006; Liu et al., 2006, 2007) focus on orchestrated applications.

The research conducted by the workflow community on fault-tolerance is also related to our proposal. An abstract model for workflows with fault-tolerance features is discussed in Chiu et al. (1999); this article also provides a good survey on fault-tolerance approaches. An algorithm for handling faults automatically in applications integrated by means of workflow management systems, which is amenable to compensation actions or to the two-phase commit protocol, is suggested in Hagen and Alonso (2000). Our reservation about this idea is that it does not isolate fault-tolerance mechanisms from business logic. In addition, it is suitable only for orchestrated applications that are based on the process-based execution model. Error recovery in workflow applications in which compensation actions are difficult or infeasible to implement is discussed in Liu et al. (2001). The authors assume that there is a centralised workflow engine, which suggests that they focus on orchestrated solutions. In Alonso et al. (2000), the authors discuss error recovery in orchestrated workflow applications that build on the process-based execution model. An architecture to implement fault-tolerance based on ad hoc workflows is discussed in Chen et al. (2004); runtime error detection is central in this proposal. The architecture relies on a web server, an application server, a database server, and a logging system. Like in our proposal, error detection is achieved by means of the analysis of message traces. The reservation we have about this approach is that processes can have only one input and messages cannot be split or aggregated inside the workflow; however, at least in theory, it is suitable for both orchestrated and choreographed processes; it supports both the process- and the task-based execution model.

An architecture for fault-tolerant workflows, based on finite state machines that recognise valid sequence of messages, is discussed in Ermagan et al. (2008). Error recovery actions are triggered when a message is found to be invalid, or the execution time of the state machine goes beyond a pre-defined deadline. This proposal is suitable for both orchestrated and choreographed processes; however it aims at process-based executions. In Baresi et al. (2008),

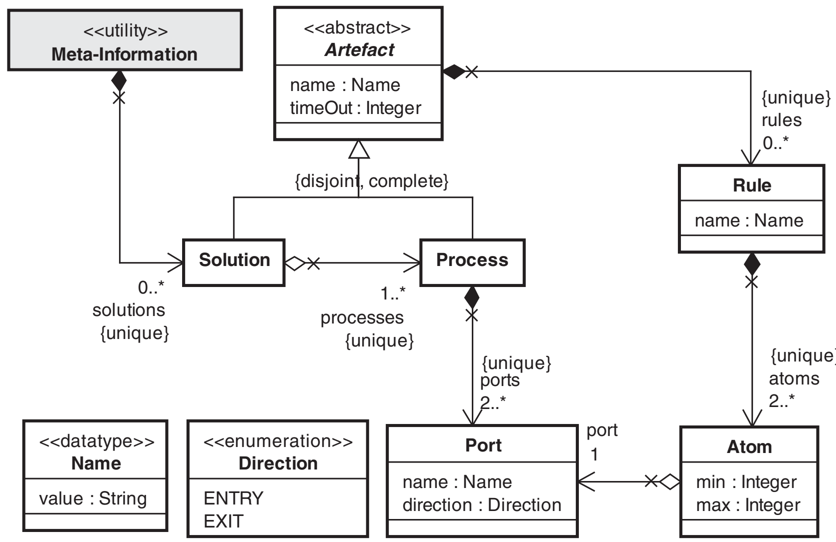

Fig. 3. Model of the Meta-information database.

$$\underbrace{P_1\,[n_1..m_1]\ \&\ P_2\,[n_2..m_2]\ \ \ldots\ \ P_k\,[n_k..m_k]}_{\text{Atoms declaring the input}} \longrightarrow \underbrace{P_{k+1}\,[n_{k+1}..m_{k+1}]\ \&\ P_{k+2}\,[n_{k+2}..m_{k+2}]\ \ \ldots\ \ P_{k+q}\,[n_{k+q}..m_{k+q}]}_{\text{Atoms declaring the output}}$$

Fig. 4. Textual syntax for rules.

the authors discuss some preliminary ideas for building an error monitor that can be used for both orchestrated and choreographed processes. No implementation or evaluation is provided, which makes it difficult to assess and apply in practice.

An approach for runtime detection of errors that emerge from potential corrupted data and faulty web services in WS-BPEL orchestrated processes is discussed in Li et al. (2009). The central idea is to regard WS-BPEL processes as discrete event systems and map them onto coloured Petri nets. The Petri net model is then integrated into additional WS-BPEL processes that are triggered when exceptions are thrown. Although the authors believe that this approach can be extended to handle choreographed processes, they do not discuss how this can be done. In Borrego et al. (2010), the authors study runtime error detection in business processes specified in BPMN and discuss a framework that includes a diagnosis layer that runs in parallel and independently from the main BPMN process. The authors consider that a BPMN process is composed of several activities that might deliver incorrect output, for instance, due to incorrect inputs provided by humans. The correct functionality of each activity is specified by means of compliance rules that are later mapped onto constraint satisfaction problems; errors are detected by means of a constraint solver. The diagnosis layer is conceptually similar to the monitor in our proposal. This proposal seems to focus on functional errors (also called business errors); in contrast, we focus on non-functional errors related to the computer infrastructure. A limitation of this proposal is that, in its current state, it can handle only a single instance of a BPMN process.

Error detection has also been intensively studied by designers of controllers that automate the operation of manufacturing plants, which can also be viewed as discrete event systems. These controllers need to process events that arrive from different sensors. Consequently, to determine whether the plant is operating correctly, the controller needs to perform intensive event correlation. In Li et al. (2008), the authors explain how one can detect errors in manufacturing plants that can be specified and controlled by means of Petri nets. They focus on place faults (for instance, tokens that are not removed after firing a transition) caused by sensor failures and bit flips. To provide the controller with error detection capabilities they embed it into a larger controller that provides additional redundant places, tokens and transitions that are used to specify correctness invariants. Errors are detected by linear parity checks at run time. The similarity between this proposal and ours lies in the use of event correlation for error detection. This proposal aims at choreographed applications. Our criticism is that it addresses only a single class of errors, namely, place errors. Neither is it clear how these ideas can be ported to business processes that are not modelled as Petri nets and involve a large amount of events and process instances executing simultaneously.

A core feature of our proposal is that we assume that all of the events produced by an EAI solution are notified (in other words, observable to the error detector). Several authors have studied error detection in discrete event systems that are considered to be in failure when they do not produce one or more expected events (Sampath et al., 1996). The challenge here is to analyse the observable events to infer what unobservable event or events drove the system into an abnormal state. This kind of systems is outside the scope of our current work. In Yan and Dague (2007), Yan et al. (2005), the authors suggest to re-use the body of knowledge about error detection in industrial discrete event systems. They discuss runtime error detection of orchestrated web services. A salient feature of this proposal is that, similarly to Sampath et al. (1996), the authors assume that failure events are not observable. The granularity of execution in this approach is at the process level. Our

a) P[+] = P[1 .. Integer.MAX_VALUE]
b) P[*] = P[0 .. Integer.MAX_VALUE]
c) P[?] = P[0..1]
d) P[n] = P[n..n]

Fig. 5. Syntactic sugar for rules.

criticism against this proposal is that it focuses only on orchestrated web services and the process-based execution model.

In Frantz et al. (2011), we presented a preliminary proposal to detect errors in the context of EAI solutions. Its complexity to detect errors was $O(b + c\,h)$, where b and c denote variables to which there is an upper bound as long as the structure of the solutions being monitored do not change, and h denotes the average size of a history database that keeps record of the messages that are exchanged in these solutions. Note that h increases monotonically with respect to time, which implies that after a point in time, the complexity of the algorithm is dominated by h, i.e., the algorithm behaves linearly with respect to the size of the history database. The practical implication was that we needed to put an upper bound on h, i.e., we only considered events within a time window. Our current solution does not depend at all on any history databases, which implies that it is far more scalable.

3. The Meta-information database

In this section, we report on the meta-data we use to represent the information our proposal requires about the artefacts it monitors, i.e., solutions and processes. This meta-data are stored in the Meta-information database. We do not provide details on how this database is managed since this is a typical information system with an interface that can be used by administrators to register, unregister, or list information about the solutions being monitored. Instead, we focus on the meta-data it stores, whose model is presented in Fig. 3, and provide a few hints on our implementation.

A solution must be composed of at least one process, and every process must have at least two ports with different directions (Direction::ENTRY or Direction::EXIT). Every artefact, port, or rule has a name that identifies it uniquely. In addition, artefacts have a time out, which denotes the maximum time they can consume to process a set of correlated messages, and a set of rules that helps verify the correlations in which they are involved.

Rules are central to our proposal. Fig. 4 presents the syntax we use to write them textually. They are composed of two groups of atoms that are separated by an arrow. Atoms at the left hand side declare the input of the rule, whereas atoms at the right hand side declare the output of the rule, i.e., if the number of messages at the left hand side atoms occurs, then it is expected the specified number of messages in the atoms at the right hand side. Each atom is of the following form: P[min..max], where P refers to a port name, and min and max are natural numbers that represent the minimum and the maximum number of messages that are allowed at port P in a given correlation (min ≤ max). For the sake of brevity, we use the common syntactic sugar depicted in Fig. 5.

Fig. 6 presents a few rules for the artefacts in the sample solutions we introduced in Fig. 1. For instance, Rule R1 is associated to process Prc1 and involves entry port P1 and exit port P2; it states that a given correlation is valid if there is one message at port P1 and zero or one correlated message at port P2. Similarly, Rule R6 is associated to Solution2; it states that a given correlation is valid if there is one message at port P5, one correlated message at port P6, and one or more correlated messages at port P4.

4. The Event Handler

Fig. 7 depicts the model we have devised for the Event Handler. Roughly speaking, it handles events that inform about a port reading or writing a message, either successfully or unsuccessfully. The events are used to build a graph that is maintained incrementally in the Work Graph database. This graph records information about the messages being exchanged in an EAI solution and their parent–child relationships, i.e., which messages originate from which ones, for example, after executing split and merge operations. In addition, the Event Handler updates the Work Queue database in order to schedule the activation of Error Detector.

Events can be of type Reception, which are notified from ports that read data from an application (either successfully or unsuccessfully) and ports that fail to read data from communication channels, Shipment, which are notified from ports that write information (either successfully or unsuccessfully), and Transfer, which are notified from ports that succeed to read data. Every event has a target binding and zero, one, or more source bindings. We use this term to refer to the data contained 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 no problem was detected, Status::RE to mean that there was a read error, or Status::WE to mean that there was a write error. Recall that the only assumption we make is that the clock resolution of the monitor is enough to distinguish between every two messages that are read or written consecutively. Thus, we assume that no confusion regarding the same message being read from or written to the same port may happen.

The Event Handler is implemented as a single method that handles all types of events. The algorithm for this method is presented in Fig. 8. It gets an event e as input and proceeds as follows: it first finds the process to which the event refers and the solutions to which this process belongs. Then, it computes the earliest time at which the Error Detector should analyse the target binding, which is the instant when the message in the target binding was read or written plus the maximum time out involved; this is a safe deadline that guarantees that every artefact should have enough time to process the corresponding correlation. Note that the time we calculate (*notBefore*) is just a hint to be interpreted as "the Error Detector should not analyse that binding before this time"; obviously, the sooner the binding is analysed after this time has passed, the better, but it is not a real-time requirement. The algorithm then fetches both the Work Graph and the Work Queue instances and adds the target binding to them both; then, it iterates over the source bindings, if any, and adds then to the Work Graph together with an edge to link them to the target binding.

Fig. 9 illustrates a graph that results from executing the previous algorithm on a series of bindings regarding the sample system in Fig. 1. Ellipses denote bindings and arrows denote edges that connect parent bindings to their corresponding child bindings. For instance, n1 is the parent binding of n4, and the latter is the parent of n6, which is in turn the parent of n8. Inside each binding we represent the instant, the port name, the message id, and the status, respectively. A snapshot of the Work Queue is also presented in this figure.

5. The Error Detector

The model of the Error Detector is presented in Fig. 10, and the algorithm to detect errors is presented in Fig. 11.

The Error Detector executes a never-ending loop in which it fetches entries from the Work Queue, finds the correlations in which the corresponding bindings are involved, and then verifies them to find possible errors. Recall that each binding in the Work Queue is scheduled to be processed not before a given time; this implies that this algorithm may need to block for some time when the earliest binding is scheduled to be analysed after the current moment. After a correlation is verified, its bindings are removed from the Work Queue, since analysing them again would not result in new correlations.

In the following subsections, we discuss the sub-algorithms on which the Error Detector relies. In Section 5.1, we present our algorithm to find the correlation in which a binding is involved; we

Prc1

R1 = P1[1] ⟶ P2[?]

Prc2

R2 = P3[1] ⟶ P4[+]

Prc3

R3 = P5[1] & P6[1] ⟶ P7[2..4]

R4 = P5[1] & P6[0] ⟶ P7[1]

Solution1

R5 = P1[1] ⟶ P4[*]

Solution2

R6 = P5[1] & P6[1] ⟶ P4[+]

Fig. 6. Sample rules.

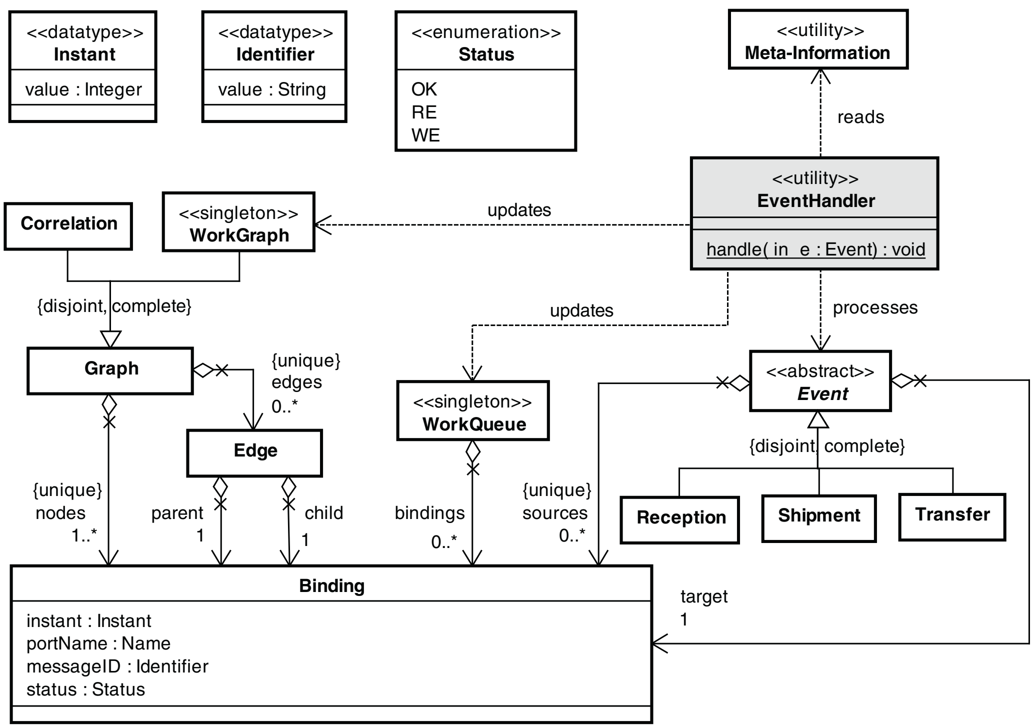

Fig. 7. Model of the Event Handler.

```
to handle(in e: Event) do
  p = find the process to which a port called e.target.portName belongs
      in the Meta-information database
  s = find all solutions to which p belongs
      in the Meta-information database
  notBefore = e.target.instant + maximum time out of artefacts in s ∪ {p}
  g = WorkGraph.getInstance()
  q = WorkQueue.getInstance()
  add e.target to g.nodes
  add e.target with priority notBefore to q
  for each binding b in e.sources do
    r = new Edge(parent = b, child = e.target)
    add r to g.edges
  end for
end
```

Fig. 8. Algorithm to handle events.

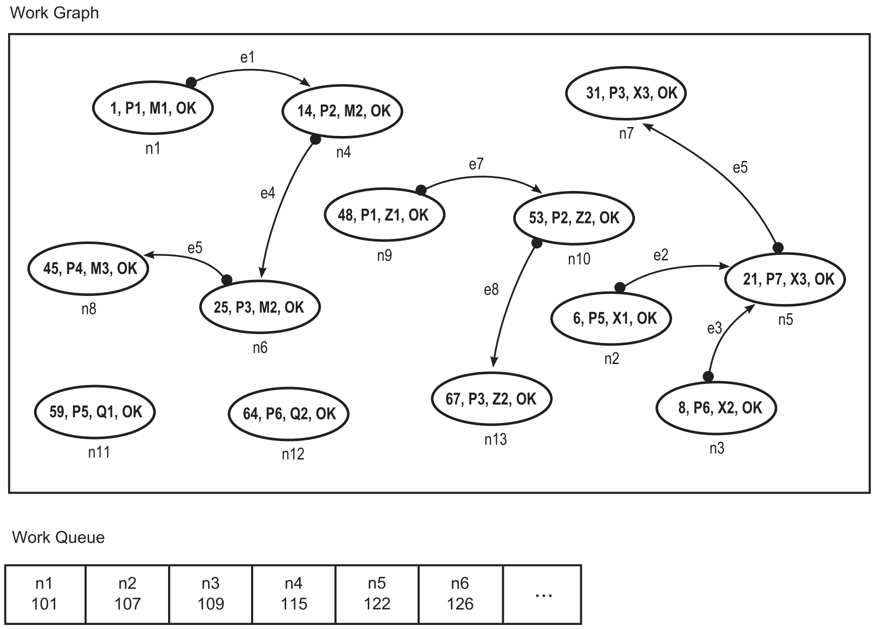

Fig. 9. Sample work graph.

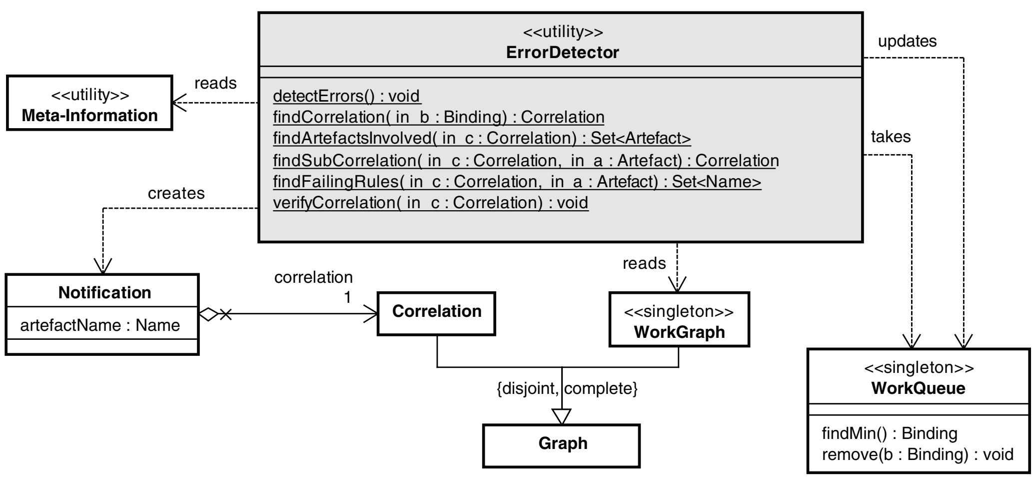

Fig. 10. Model of Error Detector.

```
to detectErrors() do
  q = WorkQueue.getInstance()
  repeat
    b = fetch minimum of q (waiting if necessary)
    c = findCorrelation(b)
    verifyCorrelation(c)
    for each binding b in c.nodes do
      remove b from q
    end for
  end repeat
end
```

Fig. 11. Algorithm to detect errors.

then report on a number of ancillary sub-algorithms, namely: an algorithm to find the artefacts that are involved in a given correlation, cf. Section 5.2, an algorithm to find the sub-correlation that corresponds to a given artefact, cf. Section 5.3; and an algorithm to find the subset of sub-rules according to which a correlation is invalid, cf. Section 5.4; the previous algorithms are used to implement the algorithm to verify a correlation, which we present in Section 5.5.

5.1. *Finding correlations*

A correlation is represented as a graph that has a single connected component that represents a subset of messages that are correlated to each other (Hopcroft and Tarjan, 1973), cf. Fig. 10.

The Error Detector provides a method called findCorrelation whose algorithm is presented in Fig. 12. It gets a binding as input and calculates the correlation in which it is involved. The main loop navigates from the binding that is passed as a parameter to all of the bindings that are reachable from it, either directly or transitively, and to all of the bindings from which it can be reached, either directly or transitively. In other words, it implements a breadth-first search to calculate the expansion of a node in a graph (Gross and Yellen, 2003).

```
to findCorrelation(in b: Binding): Correlation do
  result = new Correlation(nodes = ∅, edges = ∅)
  g = WorkGraph.getInstance()
  q = ∅
  add b to q
  whilst q <> ∅ do
    d = take a binding from q
    add d to result.nodes
    cs = (children of d in g) \ result.nodes
    add cs to q
    for each c in cs do
      h = new Edge(parent = d, child = c)
      add h to result.edges
    end for
    ps = (parents of d in g) \ result.nodes
    add ps to q
    for each p in ps do
      h = new Edge(parent = p, child = d)
      add h to result.edges
    end for
  end whilst
  return result
end
```

Fig. 12. Algorithm to find correlations.

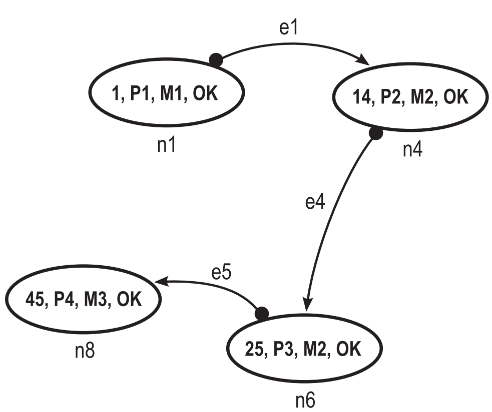

Fig. 13. Sample correlation.

For instance, if algorithm findCorrelation is invoked on bindings n1, n4, n6, or n8 in Fig. 9, it then would return the correlation in Fig. 13.

5.2. Finding the artefacts involved in a correlation

A correlation may involve several artefacts. This situation is very common since typical EAI solutions involve several processes, some of which may be shared. This implies that an event that is reported from a port may result in a binding that actually involves several artefacts.

The Error Detector provides a method called findArtefactsInvolved whose algorithm is presented in Fig. 14. It gets a correlation as input and returns a set of artefacts. The main loop of this method iterates over the set of bindings in the correlation that is passed as a parameter. In each iteration, it firsts finds the processes that own the ports from which the corresponding events were reported, and the solutions in which they are involved. The loop simply adds all of these artefacts to the result, and then returns the whole collection.

For instance, if we invoke method findArtefactsInvolved on the correlation in Fig. 13, the following artefacts involved would be returned: Solution1, Solution2, Prc1, and Prc2, cf. Fig. 16.

5.3. Finding sub-correlations

By sub-correlation, we refer to a subset of a correlation in which the ports involved belong to a given artefact. Note that there is not any structural differences between correlations and sub-correlations: they both are represented as graphs. In the sequel, we write (sub)correlation wherever we wish to emphasise that there is a single artefact from which all of the events represented in a correlation were reported.

The Error Detector provides a method called findSubCorrelation whose algorithm is presented in Fig. 15. It gets a correlation and an artefact as input and returns a (sub)correlation. The algorithm first finds which ports belong to the artefact, and then finds all of the bindings in the correlation whose ports are in the previous set; to create the resulting sub-correlation we just need to find the whole collection of edges that connect the previous bindings.

For instance, Fig. 16 shows all of the sub-correlations we can find in the correlation in Fig. 13. The dotted boxes indicate the boundary of each sub-correlation.

5.4. Finding failing rules

The Error Detector provides a method called findFailingRules that takes a (sub)correlation and an artefact as input and returns a set of names that denote the rules associated with the artefact that need to be satisfied to declare the correlation valid.

The algorithm to the findFailingRules method is presented in Fig. 17. It iterates through the collection of rules that are associated with a given artefact, and then through their atoms. The algorithm basically counts the number of bindings in the given (sub)correlation that correspond to events that were reported from the ports to which each atom refers; if this count is not within the limits that the atom specifies, then the corresponding rule does not validate the correlation and can thus be added to the result of the algorithm.

Figs. 18 and 19 illustrate the two situations in which a correlation is considered invalid due to a failing rule. The left side of the figures represents the (sub)correlation being analysed, whereas the right side represents the times at which events were reported; now represents the instant at which the analysis is performed, and deadline the latest time at which a correlation is expected to be produced (see more on this below); not before represents the time not before which the initial binding in a correlation can be analysed. Consider, for instance, rule R1 for artefact Solution1, which was introduced in Fig. 6; it states that zero or one correlated bindings are expected at port P2 for each binding at port P1. Note that in the (sub)correlation in Fig. 18 there are two correlated bindings at port P2, namely n4 and n20, which causes rule R1 to fail due to excess of bindings. Contrarily, in Fig. 19 the (sub)correlation contains less bindings than expected.

5.5. Verifying correlations

Verifying (sub)correlations is one of the central tasks of the Error Detector. A (sub)correlation can be either valid or invalid. It is valid if all of its bindings have status Status::OK, at least one rule does not fail to validate it, and all of the messages to which it refers where read or written within a given deadline; otherwise,

```
to findArtefactsInvolved(in c: Correlation): Set(Artefact) do
  result = ∅
  for each binding b in c.nodes do
    p = find the process to which a port called b.portName belongs in
        the Meta-information database
    s = find the solutions to which a process p belongs in
        the Meta-information database
    add s ∪ {p} to result
  end for
  return result
end
```

Fig. 14. Algorithm to find the artefacts involved in a correlation.

```
to findSubCorrelation(in c: Correlation, in a: Artefact): Correlation do
  p = find all ports of artefact a in the Meta-information database
  b = find all bindings b in c.nodes such that binding b.portName
      belongs to p.portName in the Meta-information database
  e = find all edges x such that {x.source, x.target} ⊆ b
  return new Correlation(nodes = b, edges = e)
end
```

Fig. 15. Algorithm to find sub-correlations.

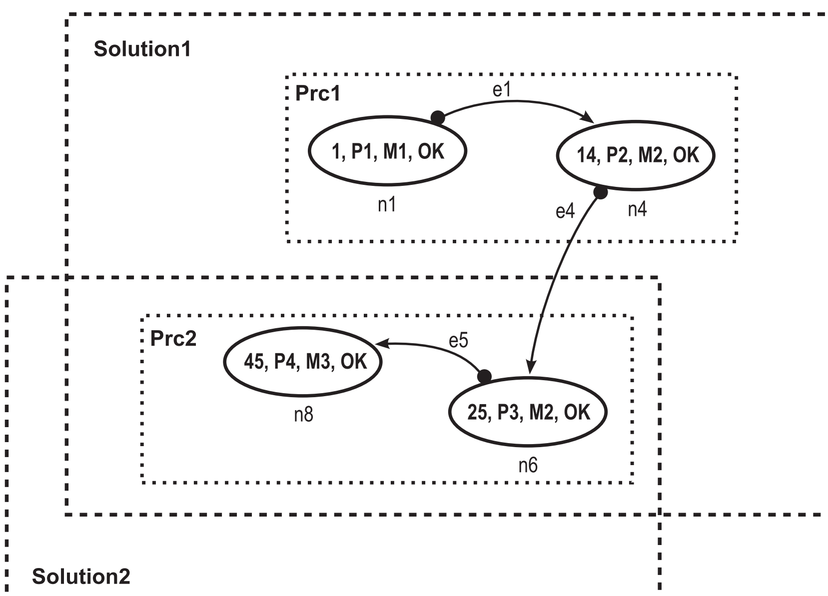

Fig. 16. Artefacts involved in a correlation.

it is invalid. The deadline refers to the time when the first message in a correlation was read or written plus the time out of the corresponding artefact. Recall that each artefact is associated with a time out that represents the maximum time it is expected to produce a correlation, cf. Fig. 3.

The Error Detector provides a method called verifyCorrelation to perform this task. The algorithm to this method is presented in Fig. 20. It takes a correlation as input and if the current correlation is invalid, then it produces a notification. The algorithm first calculates the artefacts involved in the correlation and iterates through them to find their corresponding (sub)correlations; each (sub)correlation is checked for validity according to the definition in the previous paragraph. (Sub)correlations that are found invalid are transformed into notifications that are sent to the following fault-tolerance stage so that they can be diagnosed.

6. Complexity analysis

In this section, we analyse our proposal and characterise its complexity. It can deal with an arbitrary number of processes and solutions, but we assume that there is a sensible upper bound; this

```
to findFailingRules(in c: Correlation, in a: Artefact): Set(Name) do
  result = ∅
  for each rule r in a.rules do
    for each atom t in r.atoms do
      n = count bindings b in c.nodes such that b.portName == t.port.name
      if n <= t.min or n >= t.max then
        add r.name to result
      end if
    end for
  end for
  return result
end
```

Fig. 17. Algorithm to find failing rules.

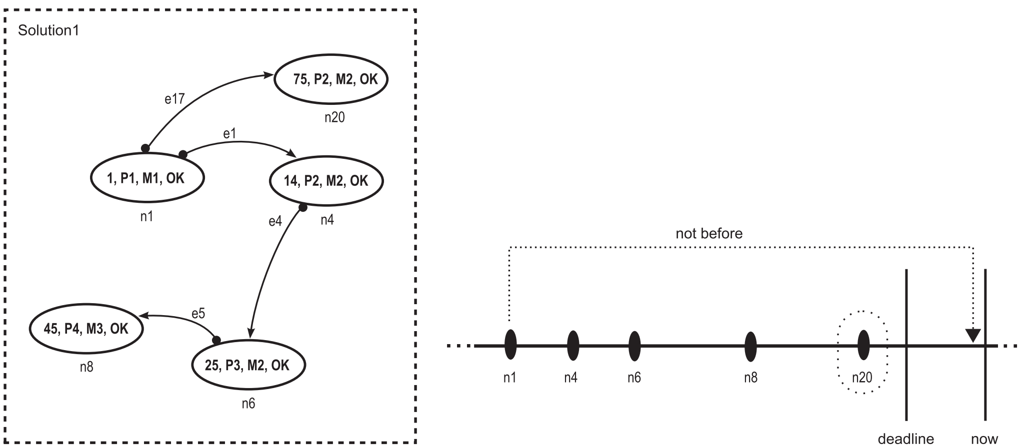

Fig. 18. A correlation that does not satisfy a rule due to excess of bindings.

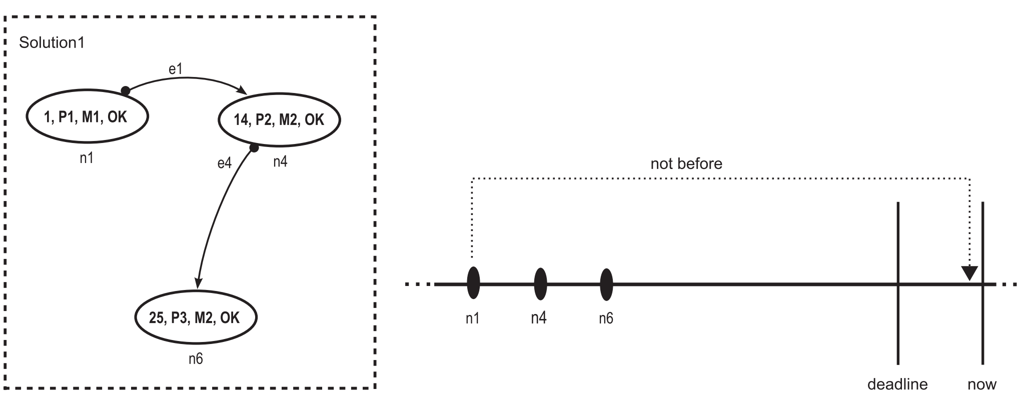

Fig. 19. Sub-correlation which causes a rule to fail due to lack of bindings.

does not amount to loss of generality since the number of artefacts of which a company's software ecosystem is composed must be necessarily finite.

Table 2 summarises the notation we use in this section. Our analysis proves that our proposal is computationally tractable since handling events runs in $O(1)$ time and detecting errors runs in $O(\log n)$ time for a given ecosystem. This makes the proposal appealing from a theoretical point of view since it is logarithmic on the size of the Work Queue database, which is expected not to be monotonically increasing or decreasing, but to grow and shrink as time progresses. Our experiments support this conjecture, cf. Section 7.

Table 2
Notation used in our complexity analysis.

Notation	Meaning
s	Maximum number of solutions to which a process can belong
u	Maximum number of source bindings in an event
b	Maximum number of bindings in a correlation
c	Maximum number of child bindings of a given binding
p	Maximum number of parent bindings of a given binding
r	Maximum number of rules associated with an artefact
t	Maximum number of atoms in a rule
a	Maximum number of artefacts involved in a correlation
n	Number of entries in the Work Queue

```
to verifyCorrelation(in c: Correlation) do
  artefacts = findArtefactsInvolved(c)
  for each artefact a in artefacts do
    s = findSubCorrelation(c, a)
    status = for all binding b in s, b.status == Status :: OK
    earliestInstant = minimum of s.nodes.instant
    latestInstant = maximum of s.nodes.instant
    deadline = earliestInstant + a.timeOut
    isValid = (latestInstant <= deadline) and
              (status == true) and
              findFailingRules(s, a) <> a.rules
    if not isValid then
      n = new Notification(artefactName = a.name, correlation = c)
      send n to the next fault-tolerance stage
    end if
  end for
end
```

Fig. 20. Algorithm to verify correlations.

6.1. On the implementation

The Meta-information database is a simple set of data. None of our queries involve joining information from other databases. In our prototype we use a hash function to index the entries of the Meta-information database, and we have implemented maps from port names onto processes, processes onto solutions, and so on. The space required by this design is proportional to the number of artefacts, ports, and rules. As a conclusion, it is possible to retrieve information from the Meta-information database in $O(1)$ time.

The Work Queue database relies on a Brodal's priority queue (Brodal, 1996), which allows to insert entries or retrieve the next to be analysed in $O(1)$ time, whereas removing an entry takes $O(\log n)$ time, where n denotes the size of the queue.

6.2. Handling events

We first report on the complexity of the algorithm to handle events, and prove that it is computationally tractable because it runs in constant time for a given ecosystem.

Theorem 1. *Algorithm* handle *in Fig. 8 terminates in $O(s+u)$ time.*

Proof. Handling an event involves finding a process using a port name is and the solutions to which this process belongs. Finding this information can be accomplished in $O(1)$ time in our implementation (lines 2 and 4). The computation of the maximum time out can be accomplished in $O(s)$ time (line 6). Getting the Work Graph and the Work Queue instances can be accomplished in $O(1)$ time (lines 7 and 8). Lines 9 and 10 add the target binding to the Work Graph and to the Work Queue databases, respectively, which can also be accomplished in $O(1)$ time. The loop in lines 11–14 iterates u times at most. Lines 12 and 13 can be implemented in $O(1)$ time since they just create an object and add it to a set. As a conclusion, the algorithm handle terminates in $O(s+u)$ time. □

Corollary 1. *In a given ecosystem, there must be an upper bound to $s+u$ because the number of solutions in a company's ecosystem is finite, which in turn implies that there is an upper bound to the number of source bindings in an event. As a conclusion algorithm* handle *terminates in $O(1)$ time in a given ecosystem.*

6.3. Detecting errors

We now analyse the complexity of the algorithm to detect errors. Note that this algorithm does not terminate, since it is a never ending loop. In this case, the complexity refers to the complexity of an iteration of this loop.

Theorem 2. *Every iteration of Algorithm* detectErrors *in Fig. 11 terminates in $O(b(1+c+p+a+\log n)+a\,r\,t)$ time.*

Proof. In each iteration, the algorithm first fetches an entry from the Work Queue database, which is accomplished in $O(1)$ time in our implementation, cf. Section 3 (line 4). According to Theorems 3 and 4 below, lines 5 and 6 run in $O(b(c+p))$ and $O(b+a(b+r\,t))$ time, respectively. The loop in lines 7–9 iterates b times at most; in each iteration, line 8 removes a binding from the Work Queue database, which is accomplished in $O(\log n)$ time. As a conclusion, each iteration of Algorithm detectErrors terminates in $O(1+b(c+p)+b+a(b+r\,t)+b\log n)=O(b(1+c+p+a+\log n)+a\,r\,t))$ time. □

Corollary 2. *In a given ecosystem, b, c, p, a, r, and t are constants because there is an upper limit to the number of artefacts a company runs. As a conclusion, each iteration of Algorithm* detectErrors *terminates in $O(\log n)$ time in a given ecosystem.*

Next, we analyse the complexity of Algorithm findCorrelation.

Theorem 3. *Algorithm* findCorrelation *in Fig. 12 terminates in $O(b(c+p))$ time.*

Proof. The loop in lines 6–21 iterates b times at most. In each iteration, the algorithm executes two inner loops. The first one, iterates c times at most and the operations inside terminate in $O(1)$ time since they just involve creating objects and adding them to a set. Similarly, the second one iterates p times at most and the operations inside also terminate in $O(1)$ time. As a conclusion, Algorithm findCorrelation terminates in $O(b(c+p))$ time. □

Now, we analyse the complexity of Algorithm verifyCorrelation.

Theorem 4. *Algorithm* verifyCorrelation *in Fig. 20 terminates in $O(b+a(b+r\,t))$ time.*

Proof. The algorithm first calculates the number of artefacts involved in a given correlation, which terminates in $O(b)$ time according to Theorem 5 below (line 2). It then executes a loop that iterates a maximum of a times (lines 3–16). In each iteration, it first needs to find a number of sub-correlations, which terminates in $O(b)$ time according to Theorem 6 (line 4); then, it calculates the status, the earliest instant, and the latest instant, which requires iterating a maximum of b times (lines 5–7); calculating the deadline can be accomplished in $O(1)$ time (line 8), but determining if a sub-correlation is valid involves executing algorithm findFailingRules, which runs in $O(r\,t)$ time according to Theorem 7 below. Lines 12–15 run in $O(1)$ time since they just require creating an object and sending it to another stage. As a conclusion, Algorithm verifyCorrelation terminates in $O(b + a(b + r\,t))$ time. □

We analyse the complexity of the three sub-algorithms on which verifyCorrelation relies in the following theorems.

Theorem 5. *Algorithm* findArtefactsInvolved *in Fig. 14 terminates in $O(b)$ time.*

Proof. The loop at lines 3–9 iterates b times at most. Lines 4–8 can be implemented in $O(1)$ time, cf. Section 3. As a conclusion, finding the artefacts involved in a given correlation terminates in $O(b)$ time. □

Theorem 6. *Algorithm* findSubCorrelation *in Fig. 15 terminates in $O(b)$ time.*

Proof. The algorithm first finds the ports that belong to a given artefact, which can be accomplished in $O(1)$ time (line 2). It then finds the bindings in a correlation whose port belongs to the previous set, which requires iterating through the bindings in the correlation (line 3); this requires $O(b)$ time in the worst case. Finally, the algorithm finds the edges that connect the previous bindings, which also requires $O(b)$ time (line 5), and creates an object in $O(1)$ time (line 6). As a conclusion, Algorithm findSubCorrelation terminates in $O(1 + 2\,b) = O(b)$ time. □

Theorem 7. *Algorithm* findFailingRules *in Fig. 17 terminates in $O(r\,t)$ time.*

Proof. The loop at lines 3–10 iterates a maximum of r times, and the inner loop at lines 4–9 iterates t times at most. As a conclusion, Algorithm findFailingRules runs in $O(r\,t)$ time. □

7. Experiments

We have conducted a series of experiments to evaluate our proposal in the laboratory. We implemented them on top of a discrete event simulation layer that allowed us to run the experiments in simulated time. We ran the experiments on a machine equipped with a four-core Intel Xeon processor running at 3.00 GHz, 16 GB of RAM, Windows Server 2008 64-bit, and the 1.6.0 version of the Java runtime Environment. In the following sections, we first provide additional details on the patterns with which we have experimented, and then on the experimentation parameters; later, we draw our conclusions about the experimental results.

7.1. Experimentation patterns

We set up six well-known patterns that lay at the core of most real-world EAI solutions, namely: pipeline, dispatcher, merger, request-reply, splitter, and aggregator (Hohpe and Woolf, 2003). In the sequel, we use term producer to refer to the process or application that produces the messages that are fed into a pattern; similarly, we use term consumer to refer to the process or application that consumes the messages that the pattern produces. Furthermore, we use variable n to denote the total number of processes involved in each pattern.

In the pipeline pattern, messages flow from a producer to a consumer in sequence: the messages a process produces are consumed by the next process in the pipeline, cf. Fig. 21. There is a single producer and a single consumer, i.e., changes to n have an impact on the number of intermediate processes only. In other words, $2n$ events are reported to the monitor per message the producer feeds into the pattern. The number of events depends on the number of ports a message goes through in the pattern, each port notifies an event.

In the dispatcher pattern, there is a process that routes the messages it produces to only a specific consumer, cf. Fig. 22. Note that changes to n have an effect on the number of consumers, not on the number of producers, which is one. That is, 4 events are reported to the monitor per message the producer feeds into the pattern.

The goal of the merger pattern is to gather messages from several producers and route them to a unique consumer, cf. Fig. 23. Changes to n have an impact on the number of producers, not on the number of consumers, which is one. Note that 4 events are reported to the monitor per message a producer feeds into the pattern.

In the request-reply pattern, there are a number of processes that require a service from another process, cf. Fig. 24. Changes to n have an effect on the number of client processes that send requests to the single server process. Every message fed into the pattern results in 6 events that are reported to the monitor.

The splitter pattern has a process that splits the messages it receives from a producer into two or more messages, each of which carries a piece of the original message to a different consumer, cf. Fig. 25. Note that changes to n have an effect on the number of consumers, not on the number of producers, which is one. That is, $2(n-1)+n$ events are reported to the monitor per message the producer feeds into the pattern.

In the aggregator pattern, there is a process that aggregates messages from different producers into a single message, which is made available to a unique consumer, cf. Fig. 26. Changes to n have an impact on the number of producers, not on the number of consumers, which is one. Note that $2(n-1)+n$ events are reported to the monitor per message delivered by the pattern.

7.2. Experimentation parameters and variables

Each experiment consisted of running an instance of a pattern with a fixed number of processes (n) and a fixed mean message production rate (t); we varied n in the range 3–15 processes, and t in the range 100–1000 ms, with increments of 100 ms. In total, we ran 130 experiments for each pattern to draw our conclusions.

We sampled the production rate from a negative exponential with parameter t. Similarly, we sampled both the time to transmit messages and the time each process took to produce a correlation from a negative exponential with parameter 250 ms; the time out of every artefact was set to 5 min. We carried out additional experiments with other values for these parameters and found that they did not have an impact on the conclusions. Each experiment was run for a duration of 24 h.

In each experiment, we measured the following variables: the time to handle an event (*THE*), the size of the Work Queue (*QS*), the time each binding spent in the Work Queue after it was scheduled to be analysed (*TSQ*), and the time the detectError algorithm took to perform each iteration of its main loop (*TDE*). (In the sequel, we report on the averaged values of these variables after discarding less than 0.5% outliers using the well-known Chevischev inequality.)

7.3. Experimentation results

Figs. 27–32 present the results we have gathered regarding variables *THE*, *QS*, *TSQ*, and *TDE* for each pattern.

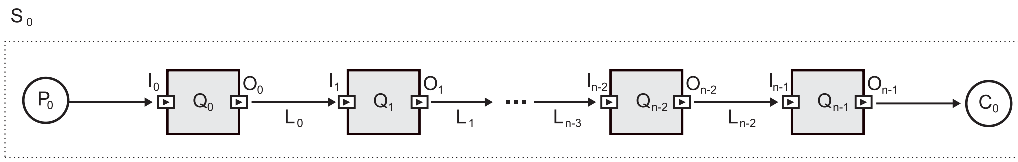

Fig. 21. Pipeline pattern.

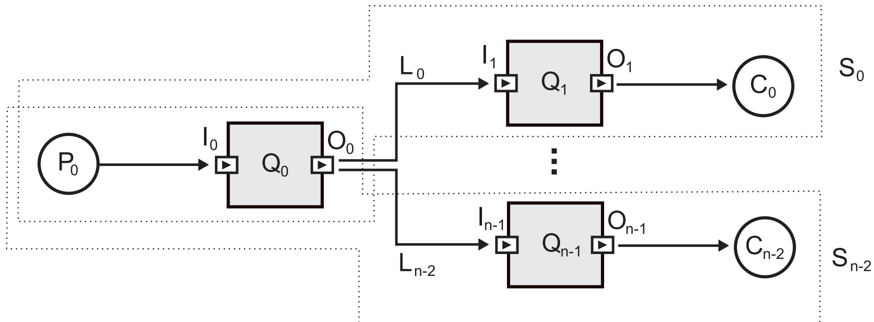

Fig. 22. Dispatcher pattern.

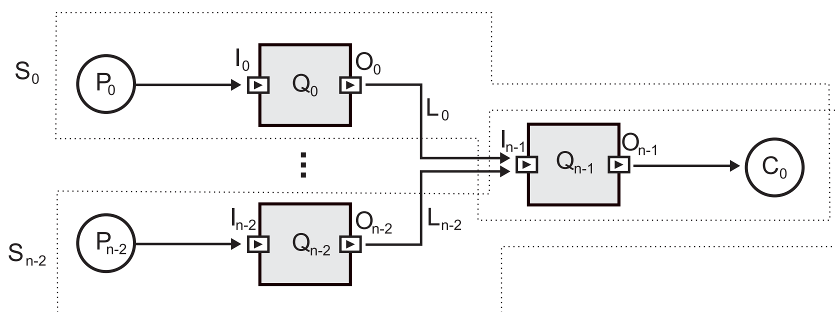

Fig. 23. Merger pattern.

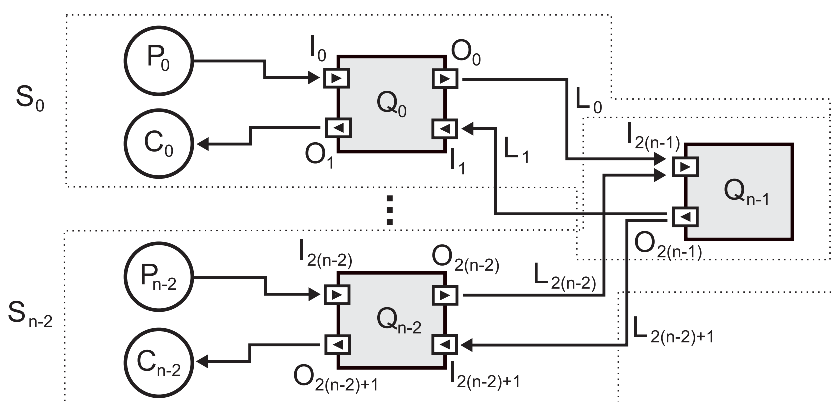

Fig. 24. Request-reply pattern.

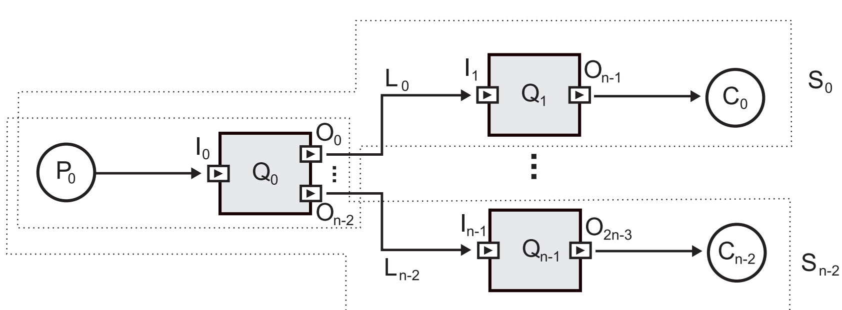

Fig. 25. Splitter pattern.

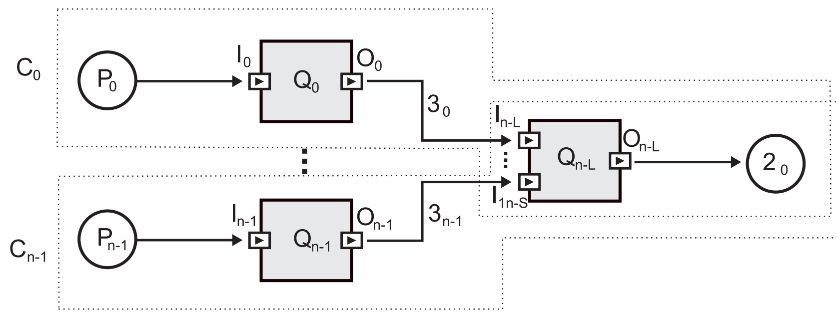

Fig. 26. Aggregator pattern.

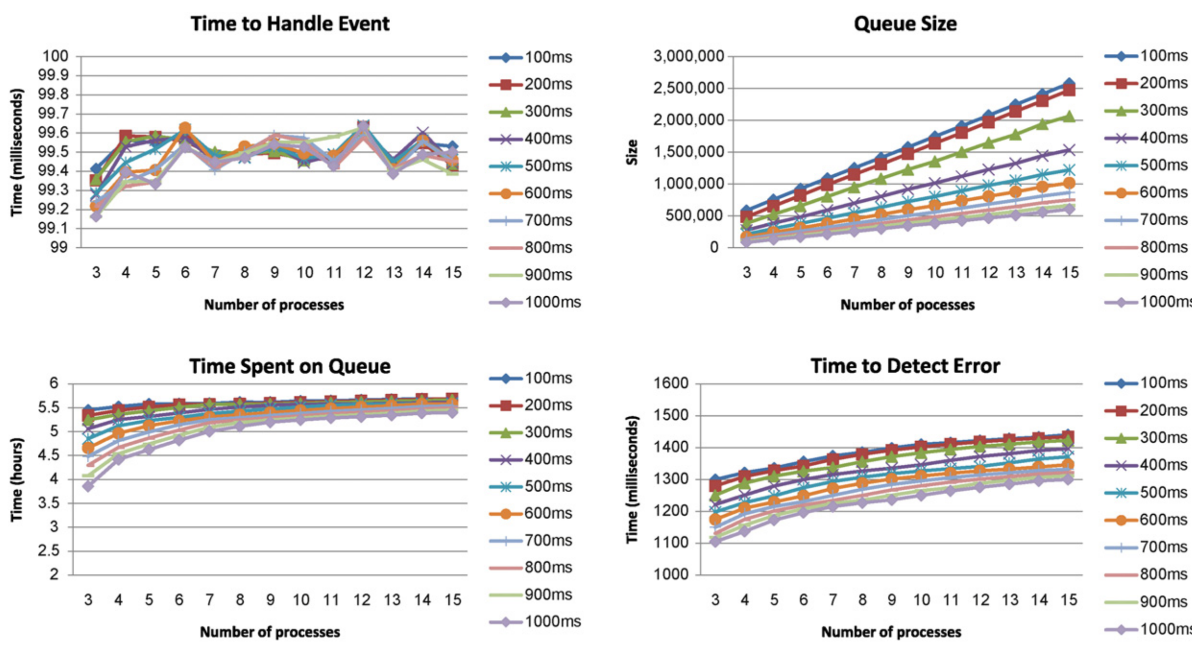

Fig. 27. Results of the experiments for the pipeline pattern.

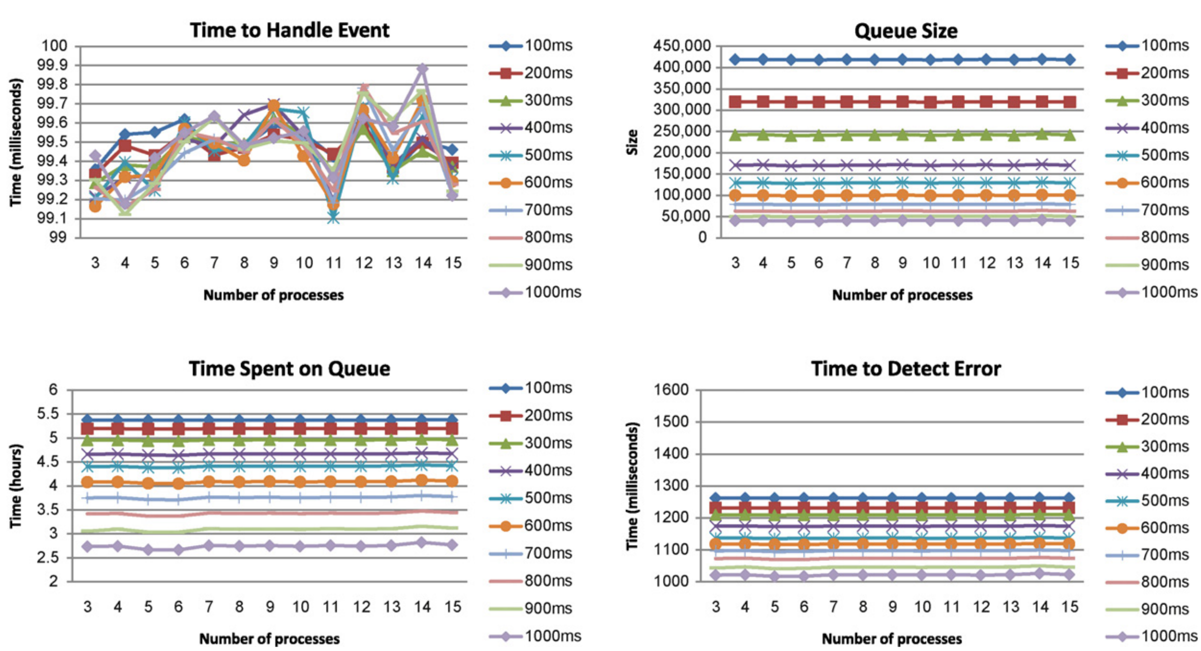

Fig. 28. Results of the experiments for the dispatcher pattern.

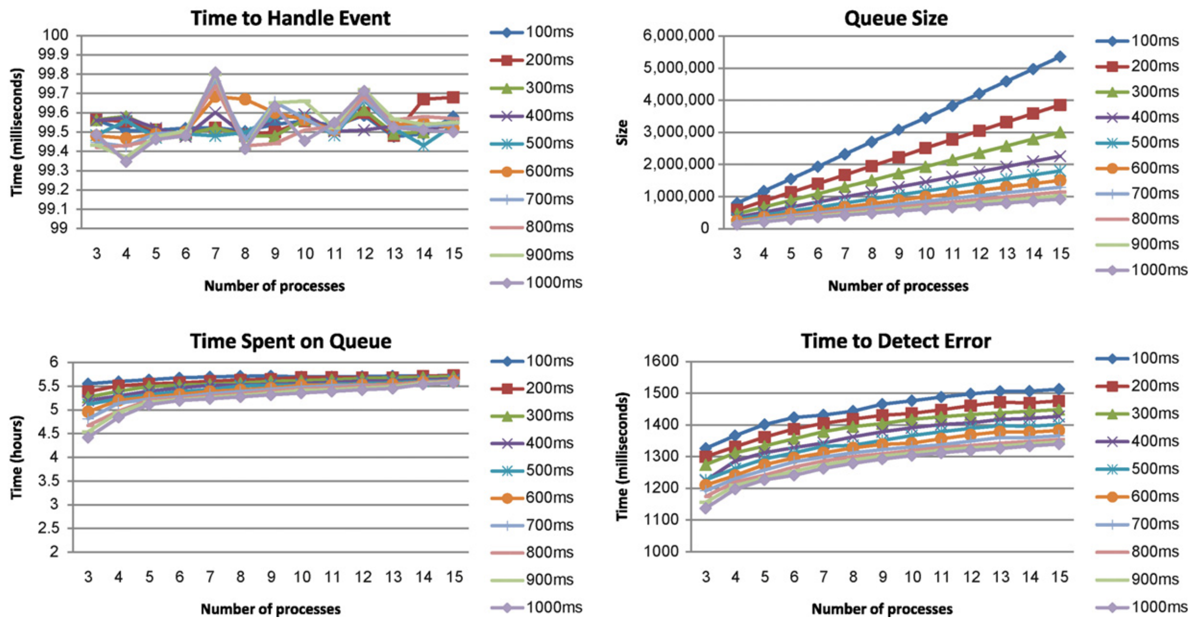

Fig. 29. Results of the experiments for the merger pattern.

The time to handle events (*THE*) remains low in all of the experiments, and the changes to *n* or *t* do not seem to have an impact on it. According to Theorem 1, the time to handle an event depends on the maximum number of solutions to which a process can belong and on the maximum number of source bindings in an event, which are fixed constants for a given ecosystem. In Corollary 1, we argued that there is an upper bound to these figures, and we concluded that the time to handle events might be considered $O(1)$ in practice. The experiments corroborate this idea, since *THE* seems to be totally independent from *n* or *t* in practice.

The size of the Work Queue (*QS*) is important because it has an impact on the memory footprint (the larger the queue, the more memory is consumed) and on the time required to complete an iteration of Algorithm detectErrors (recall that this algorithm removes all of the bindings that are correlated with the binding being analysed from the Work Queue, which requires $O(|QS|)$ time). It depends on the number of events reported in each experiment. In most cases, it behaves linearly with respect to the number of processes with a slope that depends on the mean message production rate (it decreases as this parameter increases). The only exception is the dispatcher pattern, in which *QS* seems to be a constant that depends on the mean message production rate only. The reason for this behaviour is that changes to *n* do not have an impact on the number of events that are reported. Note that every new process in the pipeline pattern contributes with 2 additional events, new processes in the merger pattern contribute with 4 additional events, and new processes in the request-reply pattern contribute with 6 additional events. Contrarily, adding a new process to the dispatcher pattern does not contribute with new events. The reason is that adding a new consumer implies that there is a new process

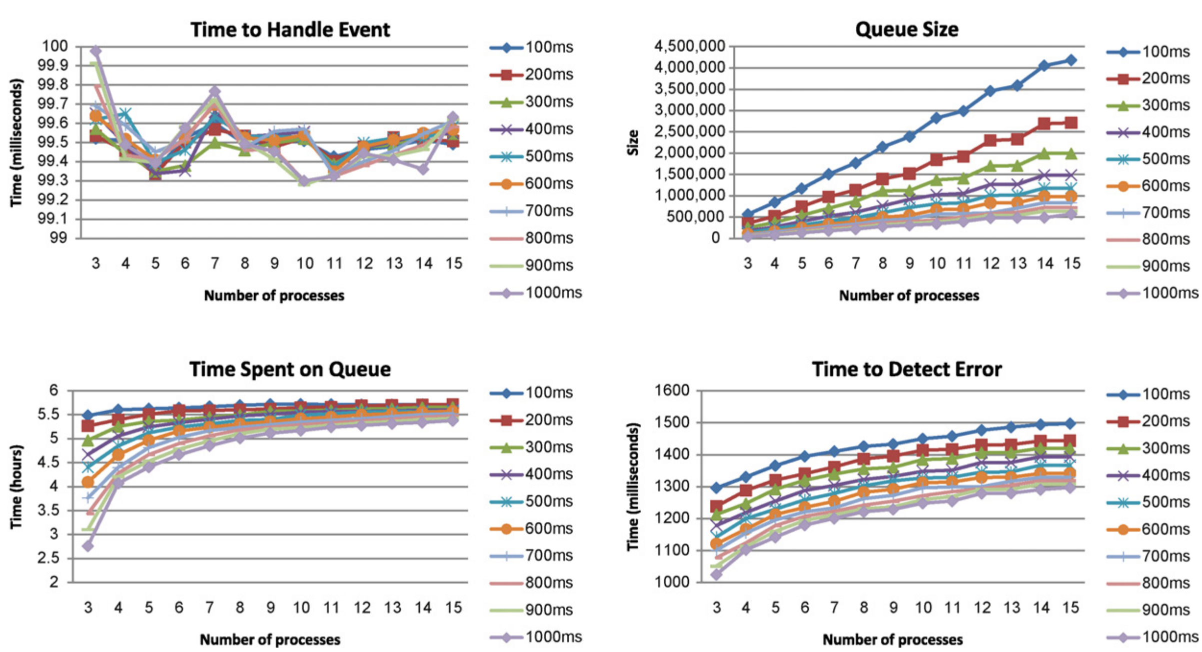

Fig. 30. Results of the experiments for the request-reply pattern.

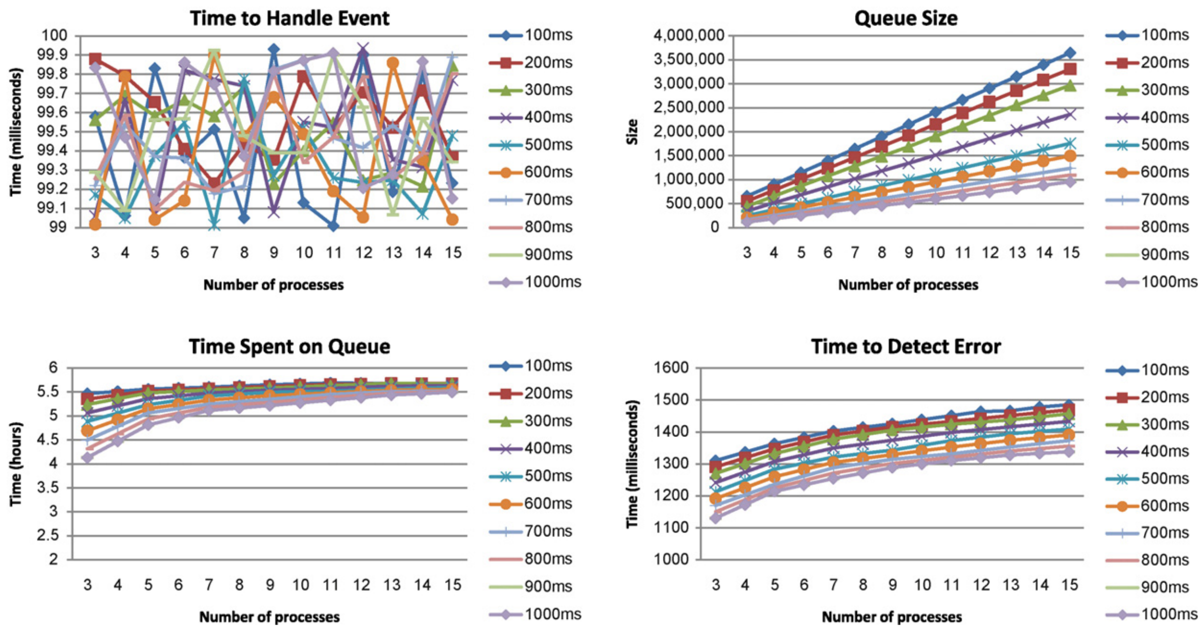

Fig. 31. Results of the experiments for the splitter pattern.

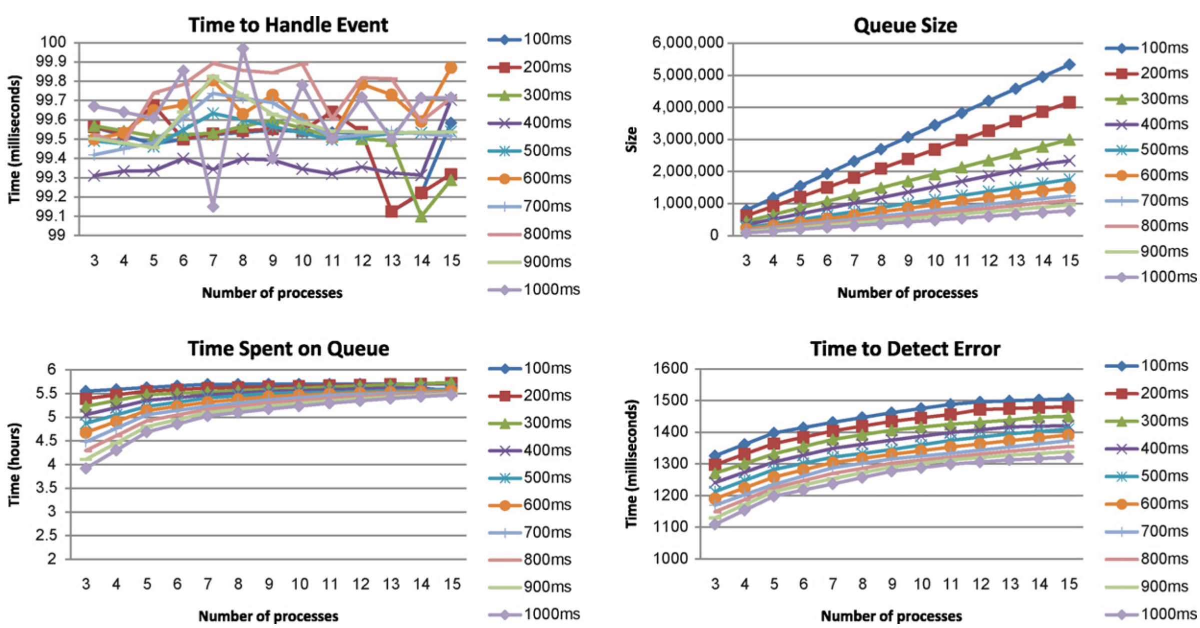

Fig. 32. Results of the experiments for the aggregator pattern.

that competes for the messages the producer feeds into the pattern; in other words, the events are reported from different sources, but the total number of events remains constant.

Variable *TSQ* is the most relevant to draw our conclusions. Recall that this variable measures the time a binding spends in the Work Queue after it is scheduled to be analysed by the Error Detector. The length of the time spent in the queue does not affect the correct functionality of the monitor. However, generally speaking, the less time, the better since this implies that errors shall be detected, diagnosed, and recovered sooner. Our experiments prove that *TSQ* seems to behave logarithmically in the number of processes in all cases, except for the dispatcher pattern, in which it behaves constantly. This implies that a change in the number of processes does not usually have a significant impact on the time bindings spend in the Work Queue. Neither does the mean message production rate seem to have a negative impact on this variable.

Regarding the time to detect errors (*TDE*), we proved that it behaves logarithmically in the size of the Work Queue, cf. Theorem 2. This theoretical result is promising as long as the size of the Work Queue does not increase monotonically, as we conjectured in Section 6. Our results support this conjecture since variable *TDE* ranges in the order of seconds in all of our experiments, even in loaded scenarios with 15 processes and a message production rate of 100 ms.

8. Conclusions

In this article, we have addressed the problem of detecting errors in EAI solutions. We have reported on a monitor that receives monitoring events with information about the messages that are read or written at each port. These events are used to build a graph that keeps record of the messages exchanged within the EAI solutions being monitored. We have designed and implemented the algorithms to analyse this graph and detect potential errors introduced when messages are read and written by ports. Our failure semantics includes communication, structural and deadline errors.

In our analysis of the related work we have found out that the majority of the proposals in the literature focus on EAI solutions that are specified as orchestrations and build on the process-based execution model. There are a few exceptions that take choreographies or the task-based execution model into account, but they have important limitations that hinder their applicability in practice.

We have analysed our proposal from a theoretical point of view and our conclusion is that it is computationally tractable. We have also carried out a series of experiments that proof our proposal can be used in settings in which the workload is high.

Acknowledgements

We would like to seize this opportunity to thank our reviewers and editors for their suggestions on how to improve our results and how to present them in this article.

The first author conducted part of this work at the Newcastle University, UK as visiting member of staff. His work is partially funded by the Evangelischer Entwicklungsdienst e.V. (EED). The first and the second authors are partially funded by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, and TIN2010-09988-E). The third author is partially funded by UK EPSRC Platform Grant Number EP/D037743/1.

References

Alonso, G., Hagen, C., Divyakant, D., El Abbadi, A., Mohan, C., 2000. Enhancing the fault tolerance of workflow management systems. IEEE Concurrency 8 (3), 74–81.
Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1 (1), 11–33.
Baresi, L., Guinea, S., Kazhamiakin, R., Pistore, M., 2008. An integrated approach for the run-time monitoring of BPEL orchestrations. In: Towards a Service-Based Internet, vol. 5377 of LNCS, Springer, pp. 1–12.
Borrego, D., Gasca, R.M., Gómez-López, M.T., Parody, L., 2010. Contract-based diagnosis for business process instances using business compliance rules. In: 21th International Workshop on Principles of Diagnosis, pp. 169–176.
Brodal, G.S., 1996. Worst-case efficient priority queues. In: 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58.
Campbell, R.H., Randell, B., 1986. Error recovery in asynchronous systems. IEEE Transactions on Software Engineering 12 (8), 811–826.
Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E., 2004. Path-based failure and evolution management. In: International Symposium on Networked Systems Design and Implementation, 23.
Chiu, D., Li, Q., Karlapalem, K., 1999. A meta modeling approach to workflow management systems supporting exception handling. Information Systems 24 (2), 159–184.
Ermagan, V., Kruger, I., Menarini, M., 2008. A fault tolerance approach for enterprise applications. In: IEEE International Conference Services Computing, vol. 2, pp. 63–72.
Erradi, A., Maheshwari, P., Tosic, V., 2006. Recovery policies for enhancing web services reliability. In: International Conference on Web Services, pp. 189–196.
Frantz, R.Z., Corchuelo, R., Molina-Jiménez, C., 2011. Error-detection in enterprise application integration solutions. In: Conference on Enterprise Information Systems, pp. 170–179.
Goodenough, J.B., 1975. Exception handling: issues and proposed notation. Communications of the ACM 18 (12), 683–696.
Gross, J.L., Yellen, J., 2003. Handbook of Graph Theory. CRC Press.
Hagen, C., Alonso, G., 2000. Exception handling in workflow management systems. IEEE Transactions on Software Engineering 26 (10), 943–958.
Hohpe, G., Woolf, B., 2003. Enterprise Integration Patterns – Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.
Hopcroft, J.E., Tarjan, R.E., 1973. Efficient algorithms for graph manipulation. Communications of the ACM 16 (6), 372–378.
Khanna, G., Laguna, I., Arshad, F.A., Bagchi, S., 2007. Stateful detection in high throughput distributed systems. In: 26th IEEE International Symposium on Reliable Distributed Systems, IEEE Computer Society, pp. 275–287.
Khanna, G., Varadharajan, P., Bagchi, S., 2009. Automated online monitoring of distributed applications through external monitors. IEEE Transactions on Dependable and Secure Computing 3 (2), 115–129.
Laguna, I., Arshad, F.A., Grothe, D.M., Bagchi, S., 2009. How to keep your head above water while detecting errors. In: 10th International Middleware Conference, vol. 5896, Springer, pp. 205–225.
Levenson, N.G., 1991. Software safety in embedded computer systems. Communications of the ACM 34 (2), 34–46.
Li, L., Hadjicostis, C.N., Sreenivas, R.S., 2008. Designs of bisimilar Petri Net controllers with fault tolerance capabilities. IEEE Transactions on System Man Cybernetics Part A 38 (1), 207–217.
Li, Y., Melliti, T., Dague, P., 2009. A colored Petri nets model for diagnosing data faults of BPEL services. In: 20th International Workshop on Principles of Diagnosis, pp. 267–274.
Liu, A., Huang, L., Li, Q., Xiao, M., 2006. Fault-tolerant orchestration of transactionsctional web services. In: International Conference on Web Information Systems Engineering, pp. 90–101.
Liu, A., Li, Q., Huang, L., Xiao, M., 2007. A declarative approach to enhancing the reliability of BPEL processes. In: IEEE International Conference on Web Services, pp. 272–279.
Liu, C., Orlowska, M.E., Lin, X., Zhou, X., 2001. Improving backward recovery in workflow systems. In: 7th International Conference on Database System Advances and Applications, p. 276.
Messerschmitt, D., Szyperski, C., 2003. Software EcoSystem: Understanding an Indispensable Technology and Industry. MIT Press.
Sayed Mouchaweh, M., 2010. Decentralized fault detection and isolation of manufacturing systems. In: 21th International Workshop on Principles of Diagnosis.
OASIS. Web Services Business Process Execution Language Version 2.0 Specification, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
Peltz, C., 2003. Web services orchestration and choreography. Computer 36 (10), 46–52.
Sampath, M., Sengupta, R., Lafortune, S., 1996. Failure diagnosis using discrete-event models. IEEE Transactions on Control System Technology 4 (2), 105–124.
W3C. Web Services Choreography Description Language Version 1.0 Specification, 2005. http://www.w3.org/TR/ws-cdl-10/.
Wu, Q., Pu, C., Sahai, A., 2006. DAG synchronization contraint language for business process. In: IEEE International Conference on Commerce and Enterprise Computing, IEEE Computer Society.
Yan, Y., Dague, P., 2007. Modeling and diagnosing orchestrated web service processes. In: IEEE International Conference on Web Services, IEEE Computer Society, pp. 51–59.
Yan, Y., Pencole, Y., Cordier, M.-O., Grastien, A., 2005. Monitoring web service networks in a model-based approach. In: Third European Conference on Web Services, IEEE Computer Society.
Zeng, L., Lei, H., Jeng, J.-J., Chung, J.-Y., Benatallah, B., 2005. Policy-driven exception-management for composite web services. In: IEEE International Conference on E-Commerce Technology, pp. 355–363.

Rafael Z. Frantz received his graduation in Informatics at UNIJUI University, Brazil in 2002. Currently, he is developing his PhD at the University of Seville, Spain. His main research interests include enterprise application integration and domain-specific languages.

Rafael Corchuelo is a Reader in Computer Science who is with the Department of Computer Languages and Systems of the University of Sevilla, Spain. He received his PhD degree from this University, and he leads its Research Group on Distributed Systems since 1997; his current research interests focus on the integration of web data islands; previously, he worked on multiparty interaction and fairness issues.

Carlos Molina-Jiménez was awarded a PhD degree in Computing Science form Newcastle University, UK. His research interests lie in distributed systems and are currently focused on middleware for supporting contract-regulated business applications, policy enforcement and formal validation of business processes.