This article was presented and published at:
XXV Ibero-American Conference on Software Engineering (CIbSE)
https://www.youtube.com/watch?v=iv_MiB0wJ2E
Check conference's web site at
https://cibse2022.frc.utn.edu.ar/
Year 2022

On the Need to Use Smart Contracts in Enterprise Application Integration

Fernando Parahyba[1], Eldair F. Dornelles[1], Fabricia Roos-Frantz[1], Rafael Z. Frantz[1], Carlos Molina-Jiménez[2], Antonia M. Reina Quintero[3], Jose Bocanegra[4], Sandro Sawicki[1]

[1]Unijuí University, Brazil

[2]Department of Computer Science and Technology, University of Cambridge, UK

[3]Department of Language and Systems, University of Seville, Spain

[4]Department of Systems and Computing Engineering, University of the Andes, Colombia

{fernando.parahyba, eldair.dornelles}@sou.unijui.edu.br

{frfrantz, rzfrantz, sawicki}@unijui.edu.br

carlos.molina@cl.cam.ac.uk, reinaqu@us.es

j.bocanegra@uniandes.edu.co

Abstract. *Integration processes involve Business Constraints and Service Level Agreements that, with current technology, are not monitored or enforced automatically at run–time. This approach leaves the participants with no means of supervising the development of their interactions or of collecting indisputable evidence to ease the resolution of disputes that can potentially emerge. In this paper, to address the issue, we suggest the inclusion of smart contracts in integration processes to supervise and mediate, at run–time, the agreements to which the participants commit. We discuss the requirements that smart contracts for integration processes need to meet and the challenges involved in writing, executing, deploying, and verifying them.*

Keywords: Smart Contract. Blockchain. Trusted Third Parties. Integration Process. Integration Solution. Service Level Agreement. Business Process. Enterprise Application Integration.

1. Introduction

We conceive the Internet as a repository of applications that are independently run by their owners and offer different computational services and data, for example, business events. It is often convenient for a company to aggregate, i.e., integrate two or more externally provided applications, say for an agreed-upon period, to implement a service rather than implement it from scratch.

The result of an integration is an integration process composed of several independently owned applications, normally conceived without foreseeing possible integration. Therefore, the integration exercise is challenging and the subject of study of Enterprise Applications Integration (EAI) – a branch of Software Engineering that provides methodologies and software tools aimed at helping software engineers in this endeavour [Zimmermann et al. 2016].

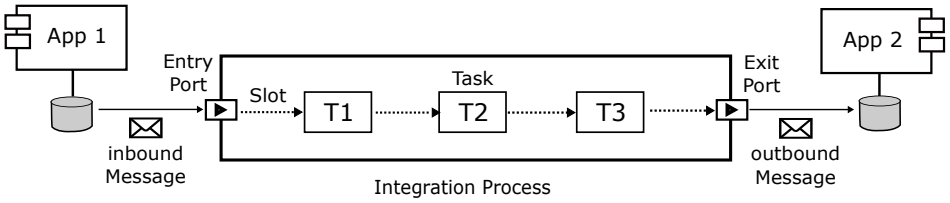


Figure 1. A purchase order integration process.

The challenge is to design an integration process that orchestrates the participating applications to keep them synchronised to provide new functionalities built from their aggregation. The quality of the integration process depends on the quality of the service delivered by the participating applications. Therefore, the owners of the participating applications need to commit to providing data (or services) with certain Quality of Service (QoS) that is specified at: Business Level and Technical Level. At Business Level, QoS is usually specified as Business Constraints (BC), which include commercial rules that must be observed, e.g., "The integration process is prohibited from sending data to the application if the value is less than US$1,000". At Technical Level, QoS is generally specified as a Service Level Agreement (SLA), which defines performance requirements, e.g., "The application is obligated to respond in less than 30 seconds".

An integration process consists of four basic concepts: Messages, Tasks, Slots and Ports. A Message is a container of data that flows through an integration process. Therefore, Messages are abstractions for data. A Message consists of two parts, namely: header and body. The header holds meta-data that identifies the data carried in the body. An integration process includes a set of atomic Tasks that manipulate the body of the Messages, such as: router; spliter; translater and merger. A Slot is a buffer that links one Task to another. Finally, a Port is an abstraction of a communication channel.

Figure 1 shows an example of a typical integration process. Let us assume that the owners of *App 1* and *App 2* have agreed that *App 1* is allowed to send purchase order data to *App 2*, but under certain restrictions: only at certain times of the day and above certain value. *App 1* sends the inbound Message which arrives at the Entry Port and travels through the Task workflow through the Slots. The outbound Message is delivered to the Exit Port and arrives at *App 2*.

The EAI field has been intensively studied and enriched with several integration platforms and tools that simplify the software engineer's work. However, unfortunately, they operate under assumptions that have become unrealistic. For example, some tools assume that a single company owns the applications, e.g., *App 1* and *App 2* and the integration process (see Figure 1). Other tools relax this assumption but consider that the owners of the applications and the integration process trust each other about the QoS delivered by the applications to the integration process and resource consumption of the latter.

With current practice, the integration process and the applications typically belong to companies that are reluctant to trust each other unguardedly. Therefore, these integration processes demand agreements to regulate the interaction to ensure that the participants behave as expected. To be of practical interest, these agreements need to be digitally automated in a manner that potential violations are *detected* and, if possible,

prevented [Antonopoulos and Wood 2018].

Smart contracts are an emerging technology that can be used to automate the supervision of agreements. Intuitively, a smart contract is an executable piece of code deployed to mediate and regulate contractual interactions. They can be used to automate contracts that include both BC and SLA agreed between two parties. Some progress has been achieved in this direction. For example, some authors have studied Business Level [Khan et al. 2021, Weber et al. 2016, Molina-Jimenez et al. 2018] while others have focused on Service Level Agreements [Kochovski et al. 2020, Uriarte et al. 2018, Tan et al. 2021].

We are unaware of scientific proposals or integration platforms that use smart contracts to specify and mediate agreements between the integration process and the applications. In this paper, to address the gap, we propose enhancing integration platforms with smart contracts that mediate and regulate agreements automatically. The challenge here is to devise languages to encode the original text of the agreement and mechanisms for executing the resulting code (the smart contract).

Ideally, the programmer of smart contracts is provided with a Domain–Specific Language (DSL) with constructs that ease the encoding of the original text. Several DSLs have been proposed for different fields [Varela-Vaca and Quintero 2021]: financial, internet of things, games, data provenance, public sector and business processes. Unfortunately, no DSLs for smart contracts in integration process are available.

In this paper, we argue that a DSL for writing smart contracts should automate the execution of the clauses included in EAI agreements. The smart contract will be responsible for notifying the participants of violations when they take place and for collecting control data. The latter can help to solve legal disputes. Currently, there is no consensus about the legal status of smart contracts [Law Commission 2020, Filippi and Wright 2018]. This discussion falls outside the scope of our work. We only focus on discussing the requirements that smart contracts for integration processes need to meet and the challenges involved in writing, executing, deploying, and verifying them. Our proposal does not consider the automation of penalty payment, as this requires a solution to several technical and legal issues.

The remainder of this paper is organised as follows: Section 2 shows related work. Section 3 defines the basic concepts for Smart Contracts. In Section 4 we discuss the implementation, execution, deployment, and verification of smart contracts for EAI. Section 5 concludes this paper.

2. Related Work

Research on smart contracts to enforce Business Constraints (BC) and Service Level Agreements (SLAs) was pioneered by Minsky in the mid 80s [Minsky and Lockman 1985] and followed by Marshall [Marshall 1993]. Though some of the smart contract tools exhibit some decentralised features [Minsky 2010], those systems took mainly centralised approaches. Within this category falls the contract enforcers discussed in [Molina-Jimenez et al. 2012], [Governatori et al. 2006] and [Perrin and Godart 2004].

Smart contracts are used to monitor Service Level Agreements [Kochovski et al. 2020] to ensure that a service is delivered with the expected Quality of Service (QoS).

[Uriarte et al. 2018] proposes a framework for managing dynamic SLAs in a distributed manner; they rely on a rich and dynamic SLA formalism expressed in smart contracts. [Scheid et al. 2019] proposed an approach which was evaluated by simulating the management of a QoS between a Service Provider and a customer; they successfully automated using a decentralised solution.

[Tan et al. 2021] propose a novel Service Level Agreement model that integrates blockchains and smart contracts where trust issues among cloud service providers, cloud service consumers, and third parties can be supervised through blockchain technologies. The proposed method strengthens data security and enables cloud service providers to provide better services.

In Business Constraints, [Weber et al. 2016] uses smart contracts and blockchain to mediate and execute the steps of a business process. The main contribution is to provide an auditable trail of all stages of the process. Neither party can unfairly accuse the other of breaches of the business process.

[Molina-Jimenez et al. 2018] discuss the advantages and disadvantages of deploying smart contracts on blockchains and alternatively on trusted third parties. They argue that hybrid platforms that integrate on and off–blockchain platforms are more adequate in several applications. They show how a smart contract can be split and executed partially on an off—blockchain contract compliance checker and partially on the Rinkeby Ethereum network. They argue that the hybrid approach avoids the difficulties that afflict blockchains, namely, scalability, performance, transaction costs, and others.

[Khan et al. 2021] discuss the challenges to be faced in the implementation of smart contracts in different areas, including business process and legal issues in the application of automated penalties. In addition, they claim that uniform application of the same rules may not be possible in different countries and that governments attempts at digital standardisation have not been successful.

In our work, we aim to implement smart contracts for integration processes. We are interested in using smart contracts to monitor automatically the observance of clauses included in agreements signed between the participants of an integration process. Due to time and space limitations, leave out of this discussion the automation of penalties. We want to provide a mechanism that either prevents the occurrence of agreements breaches or notifies the parties of their occurrence and provides evidence to help the parties sort out disputes. Existing work fails to address both aspects of BC and SLAs; smart contracts are used for monitoring either of them but not both like in our work.

3. Basic Smart Contract Concepts

Conceptually, smart contracts are similar to conventional agreements. They may include two or more participating parties and clauses. However, smart contracts are executed automatically by means of self-executing computer codes. Therefore, their correctness requirements are higher. Firstly, there should not be logical errors in their original clauses written in natural language. For example, all events that may occur must be foreseen [Mattila 2016], and clauses should not conflict with each other. Secondly, the code that implements the clauses should be written correctly. The term smart contract is attributed to Nick Szabo. He first mentioned smart contracts in [Szabo 1997] and defined them as a computerised transaction protocol that executes the terms of an agreement.

Smart contracts can be modelled as a Finite State Machine or as an Event Condition Action; both models can capture the "if this then that" construct, which means that if a condition is met, an autoresponder is triggered by the program [Molina-Jimenez et al. 2018]. For complete operability, smart contracts need intelligible and unambiguous logic, complete and accurate information. They leave no room for interpretation; they are programs that are always executed exactly as previously programmed. Smart contracts can be modelled at different levels, such as: technological, legal, business, among others. We argue that they should be modelled at the business level, where they expose meaningful concepts to business people, rather than modelling them at lower levels, where they expose meaningful implementation details to software engineers. We believe that, at this level, smart contracts should include constructs to model the following concepts:

- **Contract:** An agreement between two or more parties who do not trust each other unguardedly. A contract can be modelled as an Event Condition Action (ECA) system where events trigger the execution of actions when certain conditions are satisfied.
- **Action:** An action is an operation, for example, send, deliver, pay, delete, among others. Parties execute actions as the contract dictates and only when certain conditions hold. The execution of an action included in a clause produces an event that is expected to enable or disable an action to be executed in another clause.
- **Party:** An entity (typically an enterprise or a human) that agrees with another to sign an agreement with clauses that stipulate terms and conditions.
- **Clause:** A statement that stipulates one or more Rights, Obligations and Prohibitions that the parties are expected to observe.
  - A **Right:** An action (operation) that a party can perform if it wishes to and a condition holds. The party is free to execute the action (for example, send a purchase order) but can choose not to without negative consequences for the party. The execution of a right is illegal if the party tries to execute it when the conditions are not satisfied.
  - An **Obligation:** An action (for example, pay a bill) that a party is expected to execute to comply with the smart contract, when a condition holds. A failure to execute the action that fulfils and obligation results in penalties to be paid by the irresponsible party.
  - A **Prohibition:** An action that a party is not expected to execute when certain conditions hold unless it wishes to take the risk of being penalised.
- **Condition:** A binary statement that can be evaluated to either true or false. For example, the cost of the item included in the purchase order is less than N dollars. In smart contracts, the evaluation of a condition typically enables or disables the execution of an action related to a right, obligation or prohibition. For example, a participant has the right to send (this is the action) a purchase order only if the price of the item is less than N.
- **Event:** An occurrence of something that changes the state of a system. For example, a purchase order has been placed. The occurrence is notified to the interested parties through a Message that is recorded and alters the state of the system, that is, changes the current conditions. In smart contracts, the occurrence of events impacts the evaluation of the conditions.

3.1. Smart Contract Execution

In smart contracts, operations are encoded in executable code that is executed automatically. There are two possible approaches to check if the execution of a given operation (for example, request data from an application) is contract compliant:

- **Smart contract enforcement:** a technique to ensure that a smart contract is never breached. Enforcement is preventive, therefore, it is intrusive rather than non–intrusive like contract monitoring.
- **Smart Contract monitoring:** a technique used to passively observe the execution of a smart contract. Monitoring does not interfere with the development of the smart contract; it only observes and keeps records for potential post–mortem examination, for example, when a dispute is raised.

The two approaches have advantages and disadvantages. We will discuss them within the context of EAI in subsequent sections.

3.2. On TTP and on–blockchain Smart Contract Deployment

Smart contracts deployed on Trusted Third Parties (TTPs) are relatively easy to develop because, with this approach, the TTP operates as a centralised authority responsible for mediating the interaction of the signatories; therefore, it saves the designer from all the issues (for example, consensus, immutability, etc.) that emerge in the absence of a central authority.

Implementing decentralised smart contracts (with no TTP) is far more challenging yet possible. The task significantly simplifies if the designer takes advantage of the facilities offered by emerging blockchain platforms underpinned by Decentralised Ledgers (DLs). Ethereum (one of the leading blockchains) offers immutability, decentralisation, consensus, ubiquity, reliability, security, auditability and other fundamental services that ease the implementation of decentralised contracts.

Blockchains are known to suffer from several limitations like lack of scalability. To address these issues, some authors have suggested a hybrid approach [Molina-Jimenez et al. 2018] where the contract is partially deployed on a TTP (off–blockchain) and partially on a blockchain (on–blockchain).

3.3. Smart Contract Verification

Smart contracts that account only for smooth executions are relatively simple to implement and run. However, they are far too unrealistic to be useful in practical applications. The latter demand smart contracts that include contingency clauses that account exceptional situation that emerge from intentional and accidental diversions from the ideal execution path. These contracts tend to be complex and consequently prone to include logical errors introduced at design time in their original clauses and programming errors introduced at implementation time.

To reduce the risk of implementing unsound smart contracts, we propose formal validation at design time by means of **model checking**. Model checking is aimed at uncovering typical errors that impact contractual clauses, such as contradictions, omissions, and replications [Molina-Jimenez et al. 2012]. To reduce the risk of implementation errors, we suggest systematic testing of the actual executable code. Several authors has

studied contract correctness. For example, [Molina-Jimenez and Shrivastava 2013] discusses the implementation of a contract validator tool referred to as **contraval** that accepts abstract models written in epromela (enhanced Promela) and presents them to the Spin model checker [SPIN 2022] to check **safety** and **liveness** properties of claims expressed as basic assertions and Linear Temporal Logic (LTL) [Abdelsadiq et al. 2010]. An example of a typical **liveness** property that can be expressed in LTL is *"always a cancellation operation will be eventually followed by a refund operation"*; likewise *"the auctioneer will never withdraw his item before being sold"*, is an example of a **safety** property.

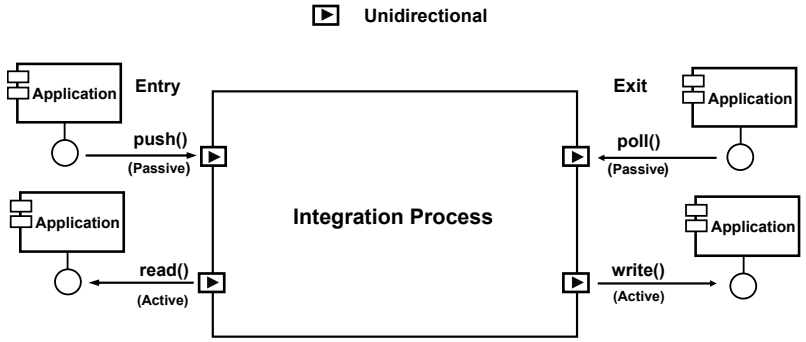
4. Smart Contracts in Enterprise Application Integration

Enterprise application integration aims at accomplishing an exogenous orchestration of software applications to support companies' business processes. Application integration is normally conducted with the help of integration platforms which are specialised software that helps to design, implement, and execute integration processes. In an integration process, data flows are enveloped in the Message format. A Message has a header and a body. The header usually contains the origin and destination of the Message and an identifier. The body contains the data that is usually structured in XML. An integration process communicates with applications through communication channels that are typically abstracted as a Port. Ports are generally configured in two modes: **unidirectional** or **bidirectional**.

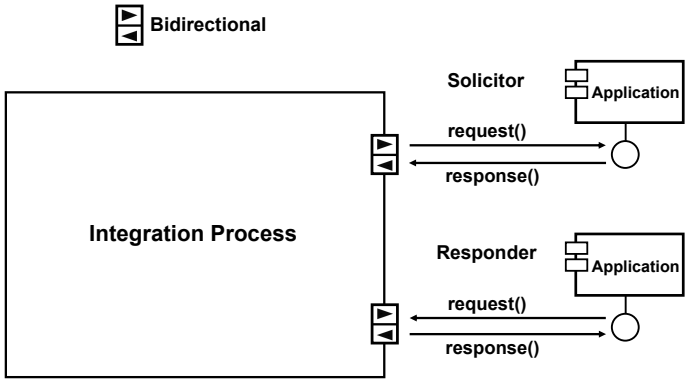
- **Unidirectional** the Messages flow only in one direction, that is, the communication channel either sends or receives data. In unidirectional mode, Ports can be **Entry** or **Exit**, and can have *active* or *passive* behaviour. If the **Entry** Port is *passive*, the application injects data into the integration (operation *"push"*). If the **Entry** Port is *active*, the integration process reads the data that is available in the application (operation *"read"*). If the **Exit** Port is *passive*, the application reads the data that is waiting in a queue at this **Exit** Port (operation *"poll"*). If the **Exit** Port is *active*, the integration process writes the data to the application (operation *"write"*). Fig. 2(a) shows the unidirectional mode and operations discussed.
- **Bidirectional** communication flows in two directions, therefore, there are a sending and a receiving channels. **Bidirectional** Ports can be classified into **Solicitor** (*active*) and **Responder** (*passive*), in both there are two operations, *"request"*, in which one of the parties executes a request, and *"response"*, which is executed by a party when it needs to respond. On the **Solicitor** Port, the integration process performs the *"request"* operation and the application performs the *"response"* operation. On the **Responder** Port, the application performs the *"request"* operation and the integration process performs the *"response"* operation. Fig. 2(b) shows the bidirectional mode and operations discussed.

An integration process belongs to a company, and it is used to support business processes that involve third-party applications. Integration processes involve agreements to be fulfilled by both parties. The challenge is to supervise these agreements to mitigate possible breaches of agreements.

An agreement in EAI involves only two parties. Firstly, the integration process represents the company that owns it. Secondly, there are the applications that will be integrated. We believe that a well written agreement should include clauses that stipulate the



(a) Unidirectional.



(b) Bidirectional.

Figure 2. Ports and types operations.

behaviour of the participants at two different levels: Business Level and Technical Level. Intuitively, the clauses at Business Level constrain the state of the business process in terms of Business Constraints (BC): they dictate what operations executed by the participants are valid or invalid. For example, what Messages the parties are expected to send to each other and what their contents should be; i.e., the agreement between the integration process and the application may express that the integration process is only allowed to execute write operations if value of the order is greater than US$1,000.

On the other hand, clauses at the Technical Level stipulate the behaviour of the technical infrastructure of the participants, that is, the state of their local systems. In other words, these clauses deal with the Quality of Service (QoS) that the participants' systems are expected to deliver and are expressed typically in terms of Service Level Agreements (SLAs). For example, the integration process may issue a request against an application that is required to respond within 30 seconds. Likewise, the contract might dictate that the integration process is not allowed to issue more than 500 requests per second.

Other researchers have studied the encoding of agreements as smart contracts. However, they either focus on agreements rich in BC or SLAs. That is, they fail to account for both types of clauses [Weber et al. 2016,Kasinathan and Cuellar 2018,Tan et al. 2021]. Consequently, existing work falls short to meet the needs of the smart contracts used in integration processes which include a combination of SLA and BC.

These contracts can be modelled based on the Event Condition Action paradigm. In our model, clauses stipulate Rights, Obligations and Prohibitions. Conditions are

Boolean statements that evaluate either to true or false and enable or disable operations. The events that trigger the clauses are Messages. The workflow tasks have already processed each Message that arrives at a Port, and the data present in the Message body is recorded as a log. This data will be taken by a tool (that has yet to be created to work collaboratively with the integration processes) to be examined by the smart contract that will verify compliance and allow or disallow the execution of operations. When the smart contract is deployed for monitoring, it only notifies of contract breaches and store evidences. Alternatively, when the smart contract is deployed to enforce an agreement, it prevents contract breaches.

4.1. Contract Execution for Enterprise Application Integration

Smart contracts in EAI can be deployed to enforce or monitor agreements signed by the participants. In this section, we discuss the advantages and disadvantages of the two approaches.

4.1.1. Enforcement Execution

With this approach, the smart contract is deployed to act intrusively to prevent the occurrence of contract breaches. We have elaborated on this point in Section 3.1. Therefore, this section will focus only on the particularities that EAI introduces.

The smart contract arbitrates between the integration process and the application so that it will check the clauses and conditions of the agreement. In EAI, enforcement can be implemented through communication channel adaptors. When a Message has already passed through the workflow, it waits in a slot until the communication channel adaptor can perform the operation against the Message. The smart contract is activated when the operation takes place, to verify if it is *contract compliant*, if it is, the execution is enabled. If it is not, the operation is disabled. The Message content that needs to be evaluated to declare the operation *contract compliant* or *non–contract compliant* is made available through the log.

For example, imagine that in an agreement, the integration process has the right to place a request against the application, as long as it is during business hours (from 8:00 am to 6:00 pm). If the operation is requested within business hours, the contract allows the communication channel adaptor to perform the operation on the Message. If it is not, the requested operation is denied to prevent a contract violation.

To use this strategy, the software engineer responsible for the integration project would need to know the implementation details of the adaptors. A technical difficulty is that adaptors vary from one integration platform to another and there are many of them. Another difficulty is that some adaptors are not available as open–source code; therefore, their manipulation is problematic. To avoid these difficulties, we rule out enforcement in favour of monitoring.

4.1.2. Monitoring Execution

With this approach, the smart contract is deployed to passively observe and collect records of operations (legal or illegal) executed by the participants against each other. We

have elaborated on this point in Section 3.1. Therefore, this section will focus only on the particularities that EAI introduces.

A smart contract deployed for monitoring in EAI is just a passive observer of the operations that the application and the integration process execute against each other. When a clause is breached, the smart contract notifies the parties that a breach has occurred.

For example, imagine that an integration process has the right to make 1000 requests per day against an application. If this number is exceeded, the owner of the integration process is obliged to pay a fine to the application owners for each extra request. The smart contract detects the breach and log the corresponding records for post–mortem examination.

The main advantage of using monitoring in EAI is that the designer does not need to modify the adaptors that carry out the communication between the integration process and the applications. The information that the contract needs for monitoring can be retrieved from the logs of the operations of each Message. The designer needs only to configure the logs to record all the needed information, for example, the content of the Messages. Log configuration is simpler than adaptor manipulation, consequently, monitoring contracts are easier to deploy in EAI that enforcing contracts. Therefore, we advise the use of monitoring contracts (rather than enforcing) when the consequences of contract breaches are not severe.

4.2. Contract Deployment for Enterprise Application Integration

Smart contracts can be deployed following two approaches: on-blockchain and off-blockchain which is also known as on–TTP deployment. In this section, we discuss the advantages and disadvantages of each approach.

4.2.1. On-blockchain deployment

There are several advantages in deploying smart contracts on public blockchains. Firstly, the blockchain provides a ready to use execution environment. Incidentally, smart contracts written in Solidity language can be deployed on the Ethereum blockchain to take advantage of the Ethereum Virtual Machine available to verify the conditions included in contract clauses to enable or disable the execution of the corresponding operations.

Another advantage is the security provided by the decentralised ledger of the blockchain. The blockchain also provides data immutability and auditability. However, there are some difficulties inherent to the context of EAI. For example, it is well-known that some EAI integration processes involve a large number of Messages that are mapped into events. In these situations, on–blockchain deployment can be costly and therefore impractical because the cost of processing Messages in blockchains is high. Another concern is that in some EAI applications, the arrival rate of Messages is high, of the order of 10.000 msg/s [Parahyba et al. 2021, Frantz et al. 2022]. With arrival rates of these orders, the blockchain is likely to cause unacceptable delays.

Another issue is the language for encoding the smart contracts. Solidity (one of the main contract languages in the Ethereum blockchain) is an alternative. However, being

a general purpose contract language, it is also a hurdle for software engineers unfamiliar with smart contract programming. We believe that a domain–specific language for EAI, with constructs for expressing concepts that frequently appear in EAI contracts, is a better approach. Such a language should allow the representation of agreements in EAI at a Platform Independent Model (PIM) level [Wöhrer and Zdun 2020].

A smart contract modelled at the PIM level can be translated automatically by means of model transformations into blockchain specific languages like Solidity. Regarding blockchain transaction costs, it is worth mentioning that an alternative to avoid it is to deploy the smart contract on a private blockchain. This is an suitable alternative to consider if the blockchain offers a sound execution environment for the smart contract and the cost of running the private blockchain is affordable.

4.2.2. On-TTP deployment

As explained in Section 3.2, smart contracts can be deployed on Trusted Third Parties that have the facilities for hosting contracts and provide the security guarantees needed. With this approach, one avoids all the problems that afflict blockchains [Mattila 2016]. However, generally, TTPs are unlikely to provide run-time mechanisms to execute the smart contract. Therefore, such approach would require the implementation of the run-time mechanisms.

An integration process that needs to execute many Messages per second may not be effective if deployed on-blockchain: the waiting time can be too high. A TTP can execute many Messages without long delays. Another factor is that a high volume of data can generate unaffordable costs on a blockchain platform. The use of a TTP can prevent this from happening; however, all the costs required to build or rent a TTP must be analysed by the software engineer. Although, the main issue is that the TTP acts as a central authority that mediates the interaction between the parties and therefore represents a potential risk to privacy. [Molina-Jimenez et al. 2018].

4.3. Formal Verification and Typical Errors in Enterprise Application Integration

A practical approach to reduce the risk of deploying unsound smart contracts is to use model checking techniques to uncover design errors at design time and testing of the actual executable code before production deployment.

Contraval is a tool that has been successfully used to model, check, and test smart contracts deployed on trusted third parties [Abdelsadiq et al. 2010, Abdelsadiq et al. 2011]. We believe that smart contracts for EAI can be validated with *contraval*, possibly, after tuning it and enriching it with new constructs to take into consideration the particularities of EAI. Regarding agreements used in EAI, we have examined several examples and have noticed that their clauses are likely to suffer from the following errors.

- **Message losses:** The arrival rate of Messages in an integration process might be extremely high (of the order of 10.000 msg/s). Such rates can cause resource saturation that results in message loss. However, some integration processes can tolerate the loss of some Messages provided that it is anticipated at design time. Incidentally, the smart contract that is responsible for supervising should be able to ignore missing Messages and continue its work.

- **Conflicting clauses:** The inclusion of clauses that conflict with each other is a typical error in EAI. For example, a clause allows the placement of requests on weekends, and another prohibits them on Saturdays.
- **Incomplete conditions:** A clause can include one or more conditions that constraint the execution of the operations included in the clause. At run time, these conditions are verified to determine if the execution of the operations is contract compliant or not. Conditions might be specified incorrectly. For example, imagine that the integration process has a right to perform a request only during business hours (from 8 am to 6 pm, Monday to Friday). A condition error will occur if only the hours are specified and the days of the week are missing.
- **Missing content:** A particularity of EAI is that to performance of conditions requires examining the actual content of the Messages. For example, a typical error is that a request to read the log returns the wrong content or no content at all, say because the integration process can not obtain this content, the smart contract must have specified how to handle if a condition fed with empty content.

The approach that we intend to follow in the validation of smart contracts for EAI will be based on the extension of the mentioned work. We will evaluate the capacity of *contraval* [Molina-Jimenez 2022] to handle the new challenge. Hopefully, the addition of some new features will suffice. However, we will need to use a more universal tool such as a *Petri Net* based tool, for example, "Maria: The Modular Reachability Analyzer" [Mäkellä 2005]. If this happens, we will take inspiration from [Kasinathan and Cuellar 2018]. Our intention is to select a *Petri Net* tool, use it for validating our use cases, explain how it can be used, including its strengths and weaknesses and produce recommendations to enhance it with features to make it more convenient and intuitive for modelling smart contracts. The actual implementation of the recommendations would be ideal but is something we would consider in the future.

This is an intellectually challenging and innovative task. It demands a deep understanding of i) model checking and testing, and ii) the particularities of model checking and testing smart contracts. This is currently an emerging research field. For example, there is little experience in modelling contract clauses with strict time constraints and contingency execution paths; there is no maturity or tool support either in expressing **safety** and **liveness** properties aimed specifically at uncovering typical bugs that impact smart contracts: What could possibly or likely go wrong in the implementation of smart contracts? What parts of the smart contract can and should be thoroughly scrutinised?

We clarify that *contraval* is a promising tool, but we are not excluding other formalisms and tools for formal validation. Incidentally, we have *Petri Net* in mind also because of two following reasons: i) we are interested in engineering solutions rather than in long-term research in smart contract validation. Thus, we are interested in a mature formalism that has been proven to solve industry problems (as opposed to illustrative academic examples), that is expressive enough to capture smart contract concepts, simple enough to master and, more importantly; ii) *Petri Nets* are supported by good and user–friendly tools that help the user interpret results; these requirements seem to be satisfied by *Petri Net* [Model Checking Contest Committee 2019, CPN Tools 2022];

5. Conclusion and Future Works

In the field of Enterprise Application Integration, companies use the services and data provided by applications owned by other companies. However, relationships are not always between companies that trust each other. Therefore companies that participate in an integration process usually sign legal agreements to stipulate Service Level Agreement and Business Constraints to be observed during their interactions.

Traditional agreements are not digitally automated, therefore, the parties have no mechanisms (software tools) to prevent breaches of contracts. Furthermore, neither they are notified when agreements are breached. To address the problem, in this paper we have proposed the use of smart contracts in EAI to automated the execution of the legal agreements used in the integration process.

For future work, we suggest developing a mathematical formalism that can assist in writing and verifying smart contracts for EAI. In addition, we suggest the use of domain-specific languages for writing smart contracts for EAI. These languages should be high level and with constructs that help the programmer to express, intuitively, concepts that frequently appear in EAI, such as: Operation, Port, Message, among others.

We suggest the use of platform–independent DSLs. The benefit of this approach is that the translation to different contract languages such as Solidity, can be automated relatively simple. We also suggest the use of tools that can help verify smart contracts for EAI with consideration of their particularities. Unfortunately, such tools are not available. Therefore, we suggest the adaptation of existing ones (for example, *contraval*, *Petri Net* and others model–checkers) to tune or tailor them to capture typical errors that afflict smart contracts used in EAI.

Acknowledgements

This work was supported by the Research Support Foundation of the State of Rio Grande do Sul in Brazil (FAPERGS), under grant 19/2551-0001782-0; the Coordination for the Brazilian Improvement of Higher Education Personnel (CAPES); and, the Brazilian National Council for Scientific and Technological (CNPq) under grant number 309315/2020-4. Furthermore, the work of Antonia M. Reina has been funded by projects AETHER-US (PID2020-112540RB-C44/AEI/10.13039/501100011033), COPERNICA (P20_01224) and METAMORFOSIS (US-1381375). Carlos Molina has been partially funded by UKRI CAMB (G115169) project[1].

References

Abdelsadiq, A., Molina-Jimenez, C., and Shrivastava, S. (2010). On model checker based testing of electronic contracting systems. In *12th IEEE International Conference on Commerce and Enterprise Computing (CEC'10)*, pages 88–95.

Abdelsadiq, A., Molina-Jimenez, C., and Shrivastava, S. (2011). A high–level model–checking tool for verifying service agreements. In *6th IEEE International Symposium on Service–Oriented System Engineering (SOSE'2011)*, pages 297–304.

Antonopoulos, A. M. and Wood, G. (2018). *Mastering Ethereum*. O'Reilly Media, Inc.

CPN Tools (2022). Colored Petri nets and CPN tools. http://cpntools.org/2018/01/16/publications/. Visited on 04 Jan 2022.

Filippi, P. D. and Wright, A. (2018). *Blockchain and the Law: The Rule of Code*. Hardvard University Press.

Frantz, R. Z., Sawicki, S., Roos-Frantz, F., Basso, F. P., Zucoloto, B., and Pillat, R. M. (2022). On the analysis of makespan and performance of the task-based execution model for enterprise application integration platforms: An empirical study. *Software: Practice and Experience*, pages 1–19.

Governatori, G., Milosevic, Z., and Sadiq, S. (2006). Compliance checking between business processes and business contracts. In *10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 221–232.

Kasinathan, P. and Cuellar, J. (2018). Securing the integrity of workflows in IoT. In *International Conference on Embedded Wireless Systems and Networks (EWSN'18)*, pages 252–257.

Khan, S. N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., and Bani-Hani, A. (2021). Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, 14(5):2901–2925.

Kochovski, P., Stankovski, V., Gec, S., Faticanti, F., Savi, M., Siracusa, D., and Kum, S. (2020). Smart contracts for service-level agreements in edge-to-cloud computing. *Journal of Grid Computing*, 18(4):673–690.

Law Commission (2020). Smart contracts call for evidence. https://www.lawcom.gov.uk/project/smart-contracts/. Visited on 19 Dec 2020.

Mäkellä, M. (2005). Maria: The modular reachability analyzer. http://www.tcs.hut.fi/Software/maria/index.en.html.

Marshall, L. F. (1993). Representing management policy using contract objects. In *IEEE First International Workshop on Systems Management*, pages 27–30.

Mattila, J. (2016). The blockchain phenomenon – the disruptive potential of distributed consensus architectures. *Berkeley Roundtable of the International Economy*, 1(38):1–26.

Minsky, N. (2010). A model for the governance of federated healthcare information systems. In *IEEE International Symposium on Policies for Distributed Systems and Networks (Policy'10)*, pages 111–119.

Minsky, N. H. and Lockman, A. D. (1985). Ensuring integrity by adding obligations to privileges. In *8th International Conference on Software Engineering*, pages 92–102.

Model Checking Contest Committee (2019). Model checking contest 2019: 9th edition, prague, czech republic, april 7, 2019. https://mcc.lip6.fr. Visited on 10 Jan 2019.

Molina-Jimenez, C. (Visited in Jan 2022). Contraval. https://github.com/carlos-molina/contraval.

Molina-Jimenez, C., Sfyrakis, I., Solaiman, E., Ng, I., Wong, M. W., Chun, A., and Crowcroft, J. (2018). Implementation of smart contracts using hybrid architectures with on and off–blockchain components. In *8th IEEE International Symposium on Cloud and Service Computing (SC2) 2018*, pages 83–90.

Molina-Jimenez, C. and Shrivastava, S. (2013). Establishing conformance between contracts and choreographies. In *IEEE Conference on Business Informatics (CBI'13)*, pages 69–78.

Molina-Jimenez, C., Shrivastava, S., and Strano, M. (2012). A model for checking contractual compliance of business interactions. *IEEE Transaction on Service Computing*, 5(2):276–289.

Parahyba, F., Frantz, R. Z., and Roos-Frantz, F. (2021). On the estimation of makespan in runtime systems of enterprise application integration platforms: a mathematical modelling approach. *International Journal of Computer Applications in Technology*, 67(1):17–28.

Perrin, O. and Godart, C. (2004). An approach to implement contracts as trusted intermediaries. In *1st IEEE International Workshop on Electronic Contracting (WEC'04)*, pages 71–78.

Scheid, E. J., Rodrigues, B. B., Granville, L. Z., and Stiller, B. (2019). Enabling dynamic sla compensation using blockchain-based smart contracts. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM'19)*, pages 53–61. IEEE.

SPIN (2022). On-the-fly, ltl model checking with spin. `http://spinroot.com`. Visited in Jan 2022.

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First monday*, 2(9).

Tan, W., Zhu, H., Tan, J., Zhao, Y., Xu, L. D., and Guo, K. (2021). A novel service level agreement model using blockchain and smart contract for cloud manufacturing in industry 4.0. *Enterprise Information Systems*, 0(0):1–26.

Uriarte, R. B., Nicola, R. D., and Kritikos, K. (2018). Towards distributed sla management with smart contracts and blockchain. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 266–271.

Varela-Vaca, Á. J. and Quintero, A. M. R. (2021). Smart contract languages: A multivocal mapping study. *ACM Computing Surveys (CSUR)*, 54(1):1–38.

Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., and Mendling, J. (2016). Untrusted business process monitoring and execution using blockchain. In *International Conference on Business Process Management (BPM'16)*, pages 329–347. Springer.

Wöhrer, M. and Zdun, U. (2020). From domain-specific language to code: Smart contracts and the application of design patterns. *IEEE Software*, 37(5):37–42.

Zimmermann, O., Pautasso, C., Hohpe, G., and Woolf, B. (2016). A decade of enterprise integration patterns: A conversation with the authors. *IEEE Software*, 33(1):13–19.