Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 201 (2022) 795–800

Procedia Computer Science

www.elsevier.com/locate/procedia

The 6th International Workshop on Agent-based Modeling and Applications with SARL (SARL)
March 22-25, 2022, Porto, Portugal

Exception Handling in SARL as a Responsibility Distribution

Matteo Baldoni, Cristina Baroglio, Giovanni Chiappino, Roberto Micalizio, Stefano Tedeschi*

Università degli Studi di Torino – Dipartimento di Informatica, Corso Svizzera 185, 10149 Torino, Italy

Abstract

Exception handling has been successfully proposed in software engineering practice as a simple, but effective, technology to address abnormal situations possibly occurring at runtime. Such mechanisms support the robust composition of heterogeneous software components, promoting code modularity, decoupling, and separation of concerns. Multi-agent systems bring these features to an extreme, but often lack systematic mechanisms for treating exceptions as part of their design. In this paper, we show how exception handling mechanism can be introduced in SARL, leveraging the abstractions that characterize its programming model. We introduce a new kind of space, supporting the responsibility distribution among agents concerning the handling of exceptional situations.

Keywords: Exception Handling; Responsibility; Software Composition; Robustness; SARL

1. Introduction

Exception handling is a well-known technology, adopted in software engineering practice to achieve robustness. Broadly speaking, it amounts to equipping a software system with the capabilities needed to tackle, at runtime, classes of abnormal situations, identified at design time. An *exception* is an "event that causes suspension of normal program execution" [17]. Therefore, the purpose of an exception handling mechanism is to provide the tools to (i) identify when an exception occurs, and (ii) apply suitable handlers, capable of treating the exception and recover. Thus, when an exception breaks the normal flow of execution, a pre-defined *exception handler* is typically executed to manage the specific situation. On its completion, the execution is possibly directed back to the normal flow of the program. *Raising* an exception is a way to signal that a given piece of the program cannot be performed normally; whereas, *handling* an exception refers to the set of instructions to be performed to restore the normal execution flow [11].

Taking a wider perspective, the need for treating exceptions emerges from the desire of structuring and modularizing software, separating concerns into independent components that interact with each other. Indeed, the seminal work

* Corresponding author.
E-mail address: stefano.tedeschi@unito.it

1877-0509
10.1016/j.procs.2022.03.112

by Goodenough on exceptions in programming languages [11, 12, 13], points out how exceptions are an important enabler for robust software composition. They allow the user of an operation to extend the operation domain (the set of inputs for which effects are defined), or its range (the effects obtained when certain inputs are processed). They allow the invoker tailoring an operation's results or effects to the particular purpose for which the operation is used, thus making it usable in a wider variety of contexts than would otherwise be the case. Consequently, an exception full significance is known only outside the detecting operation: the operation is not permitted to determine unilaterally what is to be done after an exception is raised. The invoker controls the response to the exception, that is to be activated. This increases the generality of an operation because the appropriate "fixup" will not be hard-coded inside the operation itself but, rather, it will vary from one use to the next, depending on the invoker's objectives, with straightforward benefits in terms of robustness. To make this possible the invoker must be provided with sufficient contextual information about the exceptional situation at hand.

Multi-Agent Systems (MAS), in this perspective, bring software structuring, modularization, and separation of concerns to an extreme. They embody a computational model that offers effective high-level abstractions for conceptualizing distributed, autonomous systems, characterized by multiple autonomous threads of execution that run in parallel. Here, agents amount to *loci* of decision, and they possibly need to interact and rely on one another to pursue their aims. In SARL [18], for instance, a MAS is conceived as a collection of agents interacting together through a set of shared distributed *spaces*, which realize a support layer for interaction. A specific type of space, the *event space*, is natively defined in SARL and supports event-driven interactions. Agents are equipped with *behaviors*, which map perceived events to sequences of *actions*.

Indeed, robustness is a serious concern in SARL: an agent inability to complete some behavior may have an impact on the tasks carried out by others interacting with it. Currently, the SARL programming model and its platform do not encompass an exception handling mechanism, but just the possibility for an agent to emit failure events. In this paper, we show how an exception handling mechanism can be introduced in the SARL platform, and its benefits in terms of increased robustness during the execution. To this end, we leverage the high-level abstractions that characterize its programming model; namely spaces, events and behaviors.

2. Exception Handling in SARL: a Social Attitude

At the language level, SARL supports exception throwing and catching within an agent's code, in a similar way to what is done in Java (upon which SARL is built). This language feature, however, operates at a lower level of abstraction than the agent's computational model. It deals, in fact, with exceptions at the level of the threads, that constitute each single agent, rather than with exceptions between the agents themselves. To deal with this issue, the SARL authors recently introduced the notion of *failure event*. These events represent any failure, or validation errors, that an agent could face. Each time an agent needs to be notified about a failure, an occurrence of this event type is internally fired, and eventually handled through some suitable behavior. Similarly, agents may emit failure events directed towards other agents while executing their behaviors. In particular, since agents can form holons [8, 10, 19], (failure) events can propagate from one agent to its parent in a holarchy.

Failure events alone, however, are not enough to realize Goodenough's vision, who underlines how exception handling always involves two parties: a party that is *responsible for raising* an exception, and another party that is *responsible for handling* it. The raised exception represents a *feedback* concerning the situation at hand, that the exception raiser has to provide to the exception handler, and which is seen as crucial to enable a successful handling [1, 4]. In other terms, Goodenough's vision suggests that any exception handling mechanism must be a social structure, where the parties accept to act so as to discharge responsibilities they have towards the society. While in agent organizations such social structures can be built upon institutional norms [5, 6, 7, 9], it is more challenging to map them into the SARL model, which, purposely, lacks of primitive notions of organizations and norms. Indeed, SARL allows its agents to raise and capture failure events that may resemble exceptions, but no social structure is established between the involved agents. Without a social perspective, agents are not committed either to raise or to handle failure events. Thus, failures can even get unmanaged, and no cues to understand why are available. On the other hand, when a social structure is established, agents are committed to play their roles in the systems. This enables one to check, at runtime, whether possible failures are matched with committed agents, and when a failure is not managed, it is possible to isolate the involved agents and understand why they did not behave as expected. Broadly speaking, what is

currently missing in SARL is a clear distribution of *responsibilities* among agents for raising and handling exceptions. This makes the whole system fragile since no mechanism is established to assess, at runtime, whether (and in what circumstances) such failure events will be actually raised and captured.

Since in SARL interaction among agents is based on events and supported by spaces, we propose to exploit the very same elements to encode a social structure upon which an exception handling mechanism can be realized. In particular, we introduce a dedicated space to allow agents to explicitly take responsibility w.r.t. the raising and handling of exceptions. Intuitively, by registering to this space, agents living in the same context take on the responsibility to (i) provide feedback (i.e., emit specific kinds of failure events) about the context where they detected exceptions, while executing some behaviors, or (ii) handle some of the exceptions raised by others, once the needed information (i.e., the feedback) is available. The proposed exception space realizes a channel through which relevant contextual information concerning an exception can flow from the agent detecting it to the ones impacted by and able to treat it.

More in detail, we defined an *exception space* interface, describing the following functionalities provided by the exception space:

def registerAsRaiser(ev : Class<? extends Event>, ex : Class<? extends Failure>, ag : EventListener)

By executing this action over an exception space instance, a *listener* agent takes on the responsibility for possibly raising an exception *ex* in response to an event *ev*. This creates an expectation within the society that the agent is equipped with a behavior to react to the occurrence of event *ev*. At the same time, the execution of such behavior could eventually end in the raising of an exception. Every time an agent registers as exception raiser, all the agents participating in the exception space are notified through the emission of an *ExceptionRaiserRegistered* event. The event attributes specify the agent identifier, the exception type and the type of event whose the exception could be a response to.

def registerAsHandler(ex : Class<? extends Failure>, listener : EventListener)

Similarly, by executing this action, a *listener* agent explicitly takes on the responsibility for handling an exception *ex*, should it be raised by some other agent. By doing so, the agent denotes to be recipient of the exception at hand, as exception handler, and establishes a social expectation concerning its possibility (i.e., willingness, ability, etc.) to handle it. As before, every time an agent registers as a handler for some exception, an *ExceptionHandlerRegistered* event is fired and propagated to the participating agents.

def raiseException(f : Failure, u : UUID)

This action allows a previously registered exception raiser agent to actually raise an exception instance *f* (i.e., a failure event of some kind). The event is then delivered to all the registered exception handler agents (if any). If no handler is registered for the given exception, a *NoHandlerAvailable* event is fired back to the agent who raised the exception.

The exception space interface has been concretely implemented in the Janus runtime platform[1]. In particular, we extended the *OpenLocalEventSpace* implementation of the event space provided by SARL. Once instantiated in a given context, along with the execution, the space keeps track of the agents which register as exception raisers and handlers, and propagates exceptions accordingly.

Similarly to what achieved in programming languages through constructs such as, e.g., try...catch and throws (in Java), the proposed space allows to clearly identify situations, possibly occurring during the agent interaction, which could hinder robustness, thereby making the system fragile. Moreover, agents are put in condition to reason on the expected behavior of the others, also in presence of exceptions, and calibrate their behavior accordingly. In Java, declaring that a method throws an exception allows any invoker to be aware of a potential source of fragility. If needed, the method invocation can be then surrounded by a try/catch block to handle the eventuality in a straightforward way. Analogously, if an agent registers as exception raiser in response to some event, the agents interacting with it are made aware of the fact that their interaction could end in the occurrence of an exceptional situation, and therefore may enact suitable handling strategies according to their objectives.

[1] The proposed implementation is available at https://di.unito.it/sarlexceptions.

3. An Illustrative Example

To illustrate the usage and benefits of the exception handling mechanism in a practical use case, let us consider the following scenario. Money withdrawal at an ATM involves two steps: (i) the user types the desired amount; (ii) the money is provided. Suppose the typed amount is fed as a string (e.g., "100"), whose characters correspond to digits, and then it is parsed. A SARL application realizing the ATM could consist of a *user agent*, in charge of interacting with the user–namely gathering the input, and providing the money–, and a *parser* agent, that receives the input string from *user agent* and converts it into a number. If the string, that is inserted by the human user, is not a number in digits (e.g., "one hundred"), parsing fails. A desirable behavior would be that the system, instead of crashing, were able to cope with the situation. To make this possible, the user agent should be made aware that, in some cases, parsing requests could raise exceptions (e.g., not a number). Thus, for any possible exception raised during parsing, the user agent should be equipped with a suitable handling strategy. For instance, in the case of the not a number exception, the user agent could ask the human user for another amount. Note that this would be completely transparent to the parser agent, which is unaware of how its inputs are generated, and of how its outputs are used. The exception space we propose enables this mechanism: it makes an agent aware of the exceptions that may be generated by other agents as a response to events it has previously emitted. Awareness, thus, puts the agents in the position to make the overall system more robust when they assume the responsibilities of handling exceptions.

Without such a mechanism, the potential fragility concerning the handling of non-numeric strings would remain opaque. The user agent could not leverage any social expectation on the parser's behavior to determine that the emission of a failure event could be due to its previous request. A failure event, thus, could be left unhandled because the other agent in the space would not recognize it as an event pertaining its objectives.

The following listing shows an excerpt of the parser agent's code using the exception space we propose.

```
1  agent Parser {
2
3    uses DefaultContextInteractions, Behaviors,Logging
4    var exSpace : ExceptionSpace
5
6    on Initialize {
7      exSpace = defaultContext.getOrCreateSpaceWithSpec(
8                                  typeof(ExceptionSpaceSpecification), occurrence.parameters.get(0) as UUID)
9      exSpace.registerAsRaiser(ParsingRequest,NotANumberException, asEventListener)
10   }
11
12   on ParsingRequest {
13     var amountString = occurrence.amount
14     var amountInt : int
15     var parsingResult : boolean
16     // Perform the parsing ...
17     if(parsingResult) {
18       emit(new ParsingDone(amountInt))
19     }
20     else {
21       exSpace.raiseException(new NotANumberException(index), ID)
22     }
23   }
24 }
```

Listing 1. Excerpt of the *parser* agent's code.

Both the parser and the user agent interact through the same instance of an exception space, created by following the corresponding space specification (see Line 7). As soon as the agent is initialized, it registers itself as raiser of a NotANumberException in response to a ParsingRequest event (Line 9). Indeed, the agent is equipped with a behavior triggered by the very same event, whose body attempts to perform the parsing of the string specified as attribute in the event occurrence. If the string is not a number, the agent raises a NotANumberException. The exception is a specialization of SARL failure event and includes an index attribute, denoting the index of the first non numeric digit found in the string.

As soon as the exception is raised, the space instance propagates it to all the agents registered as handlers, if any. Listing 2, below, shows an excerpt of the user agent's code, responsible for exception handling.

```
1  agent UserAgent {
2
3    uses DefaultContextInteractions, Schedules, Behaviors, Logging
```

```
4   var exSpace : ExceptionSpace
5   var attempts = 1
6
7   on Initialize {
8     exSpace = defaultContext.getOrCreateSpaceWithSpec(
9                                   typeof(ExceptionSpaceSpecification), occurrence.parameters.get(0) as UUID)
10    exSpace.registerStrongParticipant(asEventListener)
11  }
12
13  on RequestInput {
14    // Gather input from user
15    emit(new ParsingRequest(amount))
16  }
17
18  on ExceptionRaiserRegistered[occurrence.ev === ParsingRequest && occurrence.ex === NotANumberException] {
19    exSpace.registerAsHandler(occurrence.ex, asEventListener)
20    wake(new RequestInput)
21  }
22
23  on ParsingDone {
24    // Provide money
25    wake(new WithdrawalCompleted)
26  }
27
28  on NotANumberException {
29    if(attempts++ < 3) {
30      wake(new RequestInput)
31    } else {
32      wake(new CloseATM)
33    }
34  }
35
36  on CloseATM { ... }
37  on WithdrawalCompleted { ... }
38 }
```

Listing 2. Excerpt of the *user agent*'s code.

The code snippet above highlights the usefulness of our exception space. The user agent, in fact, is aware that when it emits a ParsingRequest event, a NotANumberException could be generated as a response, and hence it takes on the responsibility for handling this exception. This is apparent in Lines 18-21, where the agent registers itself as handler for NotANumberException as soon as another agent has registered as a parser. Thus, should the exception be raised by the parser, the space would propagate the failure event to the user agent, triggering the behavior in Lines 28-34. In detail, the exception is actually handled by keeping track of the attempts made by the user, and requesting another input up to three times.

It is worth noting that, although this simple scenario involves two agents only, an exception space instance could support the composition and collective exception handling of multiple interacting agents. More sophisticated implementations could include, e.g., a *money keeper* agent, in charge of interacting with the strongbox and providing the money, or a *balance monitor* agent, in charge of authorizing the withdrawal if the user's account has sufficient balance. Each one could leverage the exception space to raise or handle those exceptions impacting its field of operation.

4. Discussion

Goodenough's work points out that any exception handling mechanism is essentially a social structure, where independent, but interacting stakeholders assume the responsibility to handle specific situations by raising or capturing exceptional events. Such a vision is particularly interesting when applied to the development of distributed systems, where two major conceptual models have been proposed in the literature: *actors* [16] and *agents*. In the actor model, computational entities are modeled as independent *actors*, communicating with others through message passing. For instance, in Akka [14, 15], actors are organized into a *supervision hierarchy*, which forms the basis of Akka's exception handling model. Specifically, actors are always created as children of some other existing actor, which supervises them and manages their lifecycle. Each time an actor faces a failure during a task, it can notify an exception to its parent actor, which, in turn, either implements suitable *supervision strategies*, or escalates the exception to its own parent. This supervision technique can be conceived as a way to move the responsibility of handling an exception from the component that fails [14] to the one that can determine the best way to treat the issue.

A substantial difference between actors and agents is that agents are not structurally bound by parent-child relationships. Even in presence of a holarchy, like in SARL, agents are autonomous in the management of their own lifecycle. Thus, exception handling must be leveraged on some explicit social structure. Multi-agent organizations (MAO) [5, 6, 7, 9], for instance, are built upon the distribution of responsibilities (i.e., task allocation). An organization typically encompasses a decomposition of a global task into sub-tasks. Sub-tasks are, then, assigned to agents by means of *norms*, that orchestrate the execution: as soon as a specific organizational task is needed to be achieved, the normative system generates an *obligation* towards some agent to achieve that task. In essence, norms shape the scope of the responsibilities that agents take when joining the organization, capturing what they should do to contribute to the achievement of the organizational goal, including the responsibilities for raising or handling exceptions [2, 3, 4].

Agents in SARL, however, cannot exploit a normative system. Thus, the distribution of responsibilities over the treatment of exceptions must be obtained in a different way. In this paper, we have proposed an *exception space* where some specific actions performed by agents assume a social value, that is, they correspond to responsibilities toward the other agents in the space. This increases the awareness of an agent about its context: an agent knows whether some of events it emits could induce exceptions, and hence it can decide to handle these exceptions. More generally, since the registrations as raiser/handler of an exception are public events, any agent into a space knows what exceptions are possibly raised and handled and by what agents. The use of failure events only, instead, is opaque to the agents: agents can neither know whether a specific failure event will be emitted, nor whether such an event will be intercepted by some other agent in the space.

Acknowledgements. Stefano Tedeschi's research project has been carried out thanks to the grant "Bando Talenti della Società Civile" promoted by Fondazione CRT with Fondazione Giovanni Goria.

References

[1] Alderson, D.L., Doyle, J.C., 2010. Contrasting views of complexity and their implications for network-centric infrastructures. IEEE Tr. on Sys., Man, and Cyber. 40.
[2] Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S., 2021a. Demonstrating Exception Handling in JaCaMo, in: Proc. of PAAMS 2021, Springer. pp. 341–345.
[3] Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S., 2021b. Distributing Responsibilities for Exception Handling in JaCaMo, in: Proc. of AAMAS 2021, IFAAMAS. pp. 1752–1754.
[4] Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S., 2021c. Robustness Based on Accountability in Multiagent Organizations, in: Proc. of AAMAS 2021, IFAAMAS. pp. 142–150.
[5] Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A., 2013. Multi-agent oriented programming with JaCaMo. Sci. Comput. Program. 78, 747–761.
[6] Dastani, M., Tinnemeier, N.A.M., Meyer, J.J.C., 2009. A programming language for normative multi-agent systems, in: Handbook of Research on Multi-Agent Systems: semantics and dynamics of organizational models. IGI Global, pp. 397–417.
[7] Dignum, V., Vázquez-Salceda, J., Dignum, F., 2004. OMNI: introducing social structure, norms and ontologies into agent organizations, in: ProMAS 2004, Selected Revised and Invited Papers, Springer. pp. 181–198.
[8] Fischer, K., Schillo, M., Siekmann, J., 2003. Holonic multiagent systems: A foundation for the organisation of multiagent systems, in: Holonic and Multi-Agent Systems for Manufacturing, Springer. pp. 71–80.
[9] Fornara, N., Viganò, F., Verdicchio, M., Colombetti, M., 2008. Artificial institutions: a model of institutional reality for open multiagent systems. Artificial Intelligence and Law 16, 89–105.
[10] Gerber, C., Siekmann, J., Vierke, G., 1999. Holonic multi-agent systems. Technical Report. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.
[11] Goodenough, J.B., 1975a. Exception handling design issues. SIGPLAN Not. 10, 41–45.
[12] Goodenough, J.B., 1975b. Exception handling: Issues and a proposed notation. Commun. ACM 18, 683–696.
[13] Goodenough, J.B., 1975c. Structured exception handling, in: Proc. of the 2nd Symposium on Principles of Prog. Languages, ACM. p. 204–224.
[14] Goodwin, J., 2015. Learning Akka. Packt Publishing Ltd.
[15] Gupta, M., 2012. Akka essentials. Packt Publishing Ltd.
[16] Hewitt, C., Bishop, P., Steiger, R., 1973. A universal modular actor formalism for artificial intelligence, in: Proc. of IJCAI 1973, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. p. 235–245.
[17] ISO/IEC/IEEE, 2010. Systems and software engineering - Vocabulary. 24765:2010(E) - ISO/IEC/IEEE International Standard .
[18] Rodriguez, S., Gaud, N., Galland, S., 2014. SARL: A general-purpose agent-oriented programming language, in: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), pp. 103–110.
[19] Schillo, M., Fischer, K., 2002. Holonic multiagent systems. Manufacturing Systems 8, 538–550.