Towards Exception Handling in the SARL Agent Platform

Matteo Baldoni[0000−0002−9294−0408], Cristina Baroglio[0000−0002−2070−0616], Roberto Micalizio[0000−0001−9336−0651], and Stefano Tedeschi[0000−0002−9861−390X] (✉)

Università degli Studi di Torino – Dipartimento di Informatica, Torino, Italy
firstname.lastname@unito.it

Abstract. We demonstrate how exception handling can be realized in the SARL agent platform. We see exception handling as a mechanism binding some agents, entitled to raise given exceptions, to the ones entitled to handle them. To this end, we define dedicated *exception spaces* through which defining the agents' behavior in presence of exceptions.

Keywords: Exception Handling · Engineering MAS · SARL

1 Introduction

Broadly speaking, exception handling 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" [9]. 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 recovering. *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 [6]. Exception handling mechanisms usually rely on some relationship between the side raising the exception and the side handling it. In programming languages, for instance, the exception raised by a function is expected to be caught by the function invoker. In the actor model [8], a similar relationship is captured by the relation parent-child among actors: the exception raised by an actor is reported to its parent. Multi-Agent Systems (MAS), on the contrary, are particularly challenging under this respect because such kinds of relationships are not part of the paradigm: agents are autonomous entities which just exchange each other messages. Moreover, in a distributed system, the component in which a failure occurs, the one entitled to raise the corresponding exception, and the one(s) entitled to handle it may amount to different agents. Our claim is that exception handling in MAS has to leverage on dedicated structures binding the agents raising given exceptions with the ones entitled to handle them. We demonstrate this in SARL [14], which currently does not include an exception handling mechanism, by specifying appropriate structures, exception spaces, whose conceptual

meaning is to distribute responsibilities among agents about the raising and handling of specific exceptions.

2 Main Purpose

In SARL [14], a MAS is conceived as a collection of agents that interact by way of a set of shared *spaces*. 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 into sequences of *actions*. At the language level, SARL only supports exception throwing and catching within an agent's code, in a similar way to what is done in Java. This language feature, however, operates at a lower level of abstraction than the agent computational model. It deals with exceptions within the threads that constitute each single agent. Recently, the notion of *failure event* has been introduced [1]. Failure events represent any failure an agent could face while executing a behavior. They can be fired and captured by the same agent, and they can also be directed towards other agents.

Failure events alone, however, are not enough to implement an exception handling mechanism. The raising of a failure event, in fact, does not create any binding between the raiser and the catcher. Indeed, when an agent issues a failure event, it does not even know whether another agent will capture that. In our vision, the structure supporting exception handling has, first of all, to identify which agent is capable of raising a specific exception, and which other(s) will be capable of handling it. Note that, in a MAS the agents raising or handling exceptions may change according to the specific situation at hand. These are, in fact, tasks that can only be performed by agents having the right capabilities. For instance, consider a patient agent asking for a prescription to a doctor agent. In case of no answer (for instance because the request gets lost), it is the patient that can realize something has gone wrong, and raises an exception that the doctor can handle. Instead, in case the doctor cannot prescribe what requested, it is the doctor that can raise an exception. When the doctor raises the exception concerning the impossibility of making a prescription (because he/she lacks expertise), the patient is the one that should handle it. The example shows that raising exceptions and handling exceptions are tasks, and that such tasks should be under the responsibility of specific agents.

To realize this picture, we exploit the same interaction spaces through which SARL agents perceive events generated by others. In particular, we define an *exception space* as a dedicated space that agents can instantiate upon specific types of exceptions and where, by means of appropriate functionalities, they can register as either raisers or handlers of such exceptions. Broadly speaking, following [3], we may interpret the registration of an agent to an exception space as an assumption of responsibility about that exception. According to this view, the registration creates an expectation in other agents of the MAS that either the agent will raise the corresponding exception, when it is the case, or it will handle it, when raised by others.

Practically, the exception space interface is implemented in SARL's runtime platform[1], extending the *OpenLocalEventSpace* implementation of the event space. Specifically, our exception space interface for an exception of type T introduces three new actions:

- def registerAsRaiser(listener : EventListener): a *listener* agent registers as raiser of exceptions of type T. The agent should be equipped with some behaviors that allow the exception raising. Whenever an agent registers as exception raiser, all the agents participating in the exception space are notified through the emission of an *ExceptionRaiserRegistered* event.
- def registerAsHandler(listener : EventListener): a *listener* agent registers as handler of exception of type T. Whenever an agent registers as handler for some exception, an *ExceptionHandlerRegistered* event is fired and propagated to the participating agents. Note that many agents may be registered to handle the same exception. In this case, once the exception is raised, it is delivered to all the registered handler agents, each one implementing a suitable handling.
- def raiseException(ex : T, u : UUID): allows an agent registered as exception raiser to actually raise an exception instance *ex* of type T. The event is delivered to all the suitable registered exception handler agents (if any). If no handler is registered for the given exception, a *NoHandlerAvailable* event is fired back to the agent that raised the exception.

Every time an exception space instance is created, an *ExceptionSpaceCreated* event is fired to notify the other agents about the type of exceptions the space aims at treating. Along the execution, an instantiated exception space keeps track of the agents which register as raisers and handlers for the corresponding exception type, and propagates exceptions accordingly.

3 Demonstration

We consider a scenario from [5], involving a patient, a doctor, and a pharmacist: the patient complains of some symptoms with the doctor, expecting the doctor to provide a prescription to the pharmacist, who should then fill it and send the medicines to the patient. A possible exception in this scenario is related to the patient not receiving the expected medicines. The following listing shows an excerpt of the patient agent's code using the exception space we propose.

Patient, doctor and pharmacist interact in the same context. The patient creates an exception space instance for exception LostMedicines (Line 9). It also registers as raiser of the exception (Line 10) and notifies the other agents about the creation of the exception space by emitting a dedicated *ExceptionSpaceCreated* event (Line 11).

```
1 agent Patient {
2   uses DefaultContextInteractions, Behaviors, Logging, Schedules
```

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

```
3   var exSpace : ExceptionSpace<LostMedicines>
4   var prescriptionReceived = false
5   on Initialize {
6     val type = new ExceptionSpaceSpecification.class
7       as Class<ExceptionSpaceSpecification<LostMedicines>>
8     exSpace =
9       defaultContext.getOrCreateSpaceWithID(type, UUID::randomUUID)
10    exSpace.registerAsRaiser(asEventListener)
11    emit(new ExceptionSpaceCreated(LostMedicines, exSpace.spaceID.ID))
12    emit(new GetMedication)
13  }
14  on GetMedication {
15    emit(new Consult)
16    in(5000) [
17      if (!prescriptionReceived) {
18        exSpace.raiseException(new LostMedicines, ID)
19      }
20    ]
21  }
22  on FillPrescription {
23    prescriptionReceived = true
24    emit(new FollowTherapy)
25  }
26  on FillPrescriptionAgain {
27    emit(new FollowTherapy)
28  }
29 }
```

In principle, both the doctor and the pharmacist may register as exception handlers. In this case, both agents would be notified about the exception in order to put in place suitable handling behaviors in accordance to their goals. The patient raises the exception if no medicines are received within a given amount of time since the prescription request (in the simulation 5 seconds). If the LostMedicines exception is raised (Line 18), the space instance propagates it to all the agents registered as handlers.

The listing below shows an excerpt of the pharmacist agent's code. The agent registers as handler of LostMedicines by reacting to the *ExceptionSpaceCreated* event emitted as soon as the space is instantiated (Lines 16-19).

```
1 agent Pharmacist {
2   uses DefaultContextInteractions, Behaviors,Logging
3   var exSpacePatient : ExceptionSpace<LostMedicines>
4   var exSpaceDoctor : ExceptionSpace<MissingPrescription>
5   var prescriptionReceivedFromDoctor = false
6   var medicinesSent = false
7   on Initialize {
8     val type = new ExceptionSpaceSpecification.class
9       as Class<ExceptionSpaceSpecification<MissingPrescription>>
10    exSpaceDoctor =
11      defaultContext.getOrCreateSpaceWithID(type, UUID::randomUUID)
12    exSpaceDoctor.registerAsRaiser(asEventListener)
13    emit(new ExceptionSpaceCreated(MissingPrescription,
14                                   exSpaceDoctor.spaceID.ID))
15  }
16  on ExceptionSpaceCreated[occurrence.ex == LostMedicines] {
17    exSpacePatient = defaultContext.getSpace(occurrence.id)
18    exSpacePatient.registerAsHandler(asEventListener)
19  }
20  on Prescribe {
21    prescriptionReceivedFromDoctor = true
22    // Prepare medicines
23    emit(new FillPrescription)
24    medicinesSent = true
25  }
```

```
26    on LostMedicines {
27      if(medicinesSent) {
28        emit(new FillPrescriptionAgain)
29      }
30      if(!prescriptionReceivedFromDoctor) {
31        exSpaceDoctor.raiseException(new MissingPrescription, ID)
32      }
33    }
34  }
```

When the pharmacist receives a LostMedicines exception event, an appropriate handling behavior is triggered (Lines 26-33). The agent checks if the medicines were already sent; in this case a second attempt is made, by filling the prescription (and sending the medicines) again. It is worth noting that medicines may not have been sent because the pharmacist did not previously receive the prescription from the doctor. Such an eventuality can be modeled by introducing an additional MissingPrescription exception and a corresponding exception space shared among pharmacist and doctor. Here the exception space is instantiated by the pharmacist upon initialization, who also registers as exception raiser. As a result, while handling the exception raised by the patient, the pharmacist may end in raising a further exception addressed to the doctor and concerning the missing prescription (Line 31).

The doctor will be programmed by following a similar approach. Upon instantiation, it will register as handler of both the exceptions raised by the patient and by the pharmacist. Upon raising, in turn, it will be notified about the occurrence of exceptions so as to activate proper handling behaviors. It is worth noting that the scenario can be easily extended to encompass further exceptions (e.g., raised by the doctor and handled by the patient if the request is made when the doctor is on vacation). For each exception deemed to possibly occur during the interaction a suitable exception space can be defined in order to establish a binding among the agent in charge of raising such exception and the agent(s) in charge of handling it.

4 Conclusions

Exception handling has been addressed by a number of papers in the MAS literature (see e.g., [5,7,10,11,12,13]). In this paper, we have demonstrated in SARL how exception handling can be achieved by exploiting dedicated structures, i.e., exception spaces, binding raisers and handlers, and interpreting these spaces as a distribution of responsibilities among the agents.

The exception spaces address situations, occurring during the agent interaction, which make the overall system fragile, with the aim of making the system more robust. In fact, the registrations as raiser/handler of an exception are public events, therefore any agent into a space knows what exceptions are possibly raised and handled and by what agents. This increases the awareness of the agents about their context: on the one side, by creating an exception space and registering as a raiser, an agent discloses to others what exceptions could be raised during an interaction. On the other side, being aware of an exception

space, agents may assess whether they possess the right capabilities to carry out the interaction, even in case of the raise of an exception, and decide to register as handlers. Failure events are essentially weaker than exception spaces because they cannot create a binding among the agents. The use of failure events, in fact, is opaque to the agents: agents can neither know whether a specific failure event will be emitted, nor whether such an event will ever be intercepted by some other agent in the space.

As future work, it would be interesting to study the introduction of *accountability* in the SARL framework. Accountability allows agents to get runtime information to be used in their decision making in ways that go beyond the exception handling mechanisms. So far, indeed, accountability has been studied only for frameworks (e.g. JaCaMo [2,3,4]) that provide an organizational infrastructure: roles, norms, organizational goals and how they are structured into sub-goals that are associated with roles, etc.

References

1. Management of the failures and validation errors, SARL general-purpose agent-oriented programming language ("specification"). http://www.sarl.io/docs/official/reference/Failures.html, accessed: 2023-04-20
2. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Reimagining robust distributed systems through accountable MAS. IEEE Int. Comp. **25**(6), 7–14 (2021)
3. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Exception handling as a social concern. IEEE Int. Comp. **26**(6), 33–40 (2022)
4. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Accountability in multi-agent organizations: from conceptual design to agent programming. JAAMAS **37**(1), 7 (2023)
5. Christie V., S., Chopra, A.K., Singh, M.P.: Bungie: Improving fault tolerance via extensible application-level protocols. Computer **54**(5), 44–53 (2021)
6. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Commun. ACM **18**(12), 683–696 (Dec 1975)
7. Hägg, S.: A sentinel approach to fault handling in multi-agent systems. In: Multi-Agent Systems Meth. and Appl. vol. 1286, pp. 181–195. Springer (1997)
8. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. of IJCAI 1973. p. 235–245. Morgan Kaufmann (1973)
9. ISO/IEC/IEEE: Systems and software engineering - Vocabulary. 24765:2010(E) - ISO/IEC/IEEE International Standard (2010)
10. Kalia, A.K., Singh, M.P.: Muon: designing multiagent communication protocols from interaction scenarios. JAAMAS **29**(4), 621–657 (2015)
11. Mallya, A.U., Singh, M.P.: Modeling exceptions via commitment protocols. In: Proc. of AAMAS 2005. pp. 122–129. ACM (2005)
12. Miller, R., Tripathi, A.: The guardian model and primitives for exception handling in distributed systems. IEEE Trans. on Soft. Eng. **30**(12), 1008–1022 (2004)
13. Platon, E., Sabouret, N., Honiden, S.: An architecture for exception management in multiagent systems. IJAOSE **2**(3), 267–289 (2008)
14. Rodriguez, S., Gaud, N., Galland, S.: 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). vol. 3, pp. 103–110 (2014)