Design Process for System of Systems Reconfigurations

Franck Petitdemange[1], Isabelle Borne[1], and Jérémy Buisson[2]

[1]IRISA, South Brittany University, Vannes, France
[2]IRISA, Écoles de Saint-Cyr Coëtquidan, Guer, France

DOI: `https://doi.org/10.1002/sys.21567`

Abstract

Systems of systems (SoSs) constitute a particular class of systems, whose constituents are themselves systems in their own right. Such systems present architectures that dynamically change in a way that is not necessarily intended at the time of design. We target a kind of SoS that is subject to evolutionary development and propose a reconfiguration design process that guides the architect in charge of SoS engineering by indicating the tasks that must be accomplished to develop a reconfiguration for evolutionary development. Finally, our process is applied to design a reconfiguration in the context of a realistic case study inspired by the French emergency services.

1 Introduction

Among the systems that an engineer can conceive of, systems of systems (SoSs) constitute a particular class, whose constituents are themselves systems in their own right. Such systems present architectures that dynamically change in a way that is not necessarily planned at the time of initial design. Maier [1998] and subsequent work argue that an SoS is operated and provides services even before being fully deployed; that is, the development of an SoS is evolutionary. For an SoS, whose life cycle may span decades, the capabilities are acquired and withdrawn, and the mission evolves during operation.

In the context of this paper, we focus on shorter-lived SoSs that are specifically targeted at one operation: here, a rescue operation inspired by the French civil protection [Petitdemange et al., 2018]. Figure 1 overviews this SoS, whose mission is to protect property and people in case of a flooding situation. The departmental operation center of fire and help (CODIS35 and CODIS56 – department numbers 35 and 56) oversees and coordinates the whole operation. Additional resources, such as all-purpose vehicles (VTU), unmanned aerial vehicles (UAV), lightweight inflatable boats (ERS), helicopters and ambulances, perform primitive field actions to fulfill the overall mission under the authority of a team leader. The medical emergency service (SAMU) ensures regulation of emergency rooms, which in turn provide medical assistance to victims.

The lines in figure 1 show interactions between these constituents, and the colored boxes denote the organizations to which the constituents belong. These interactions are intended for the fulfillment of the mission. This SoS highlights the managerial independence of the constituents, as each team leader, VTU, ERS and UAV belongs to a departmental fire and rescue service (SDIS35 and SDIS56), which is, in France, an autonomous agency funded by an administrative subdivision. When the situation is too large for the SDIS that is geographically responsible, it may request additional resources from a neighbor SDIS. For instance, in figure 1, the team leader is provided by SDIS35, while the VTU, UAV and ERS are provided by SDIS56. Furthermore, ambulances, SAMU and emergency rooms belong to a hospital, which is another independent organization. Helicopters belong to yet another organization, the general directorate for civil security and crisis management, which is a state-level entity in the Ministry of Interior dedicated to protecting citizens. Here, operational independence is illustrated by the fact that the hospital and its resources have to respond to all other medical requests in addition to participation in the SoS.

This SoS is subject to evolutionary development. Indeed, the field situation is not known when the operation begins: the SoS must reorganize depending on the severity of the flood and its evolution. For instance, at the time of the initial call to the emergency service, a small operation is deployed in response to a localized flood. The neighbor SDIS ensures coordination through operational collaboration, that

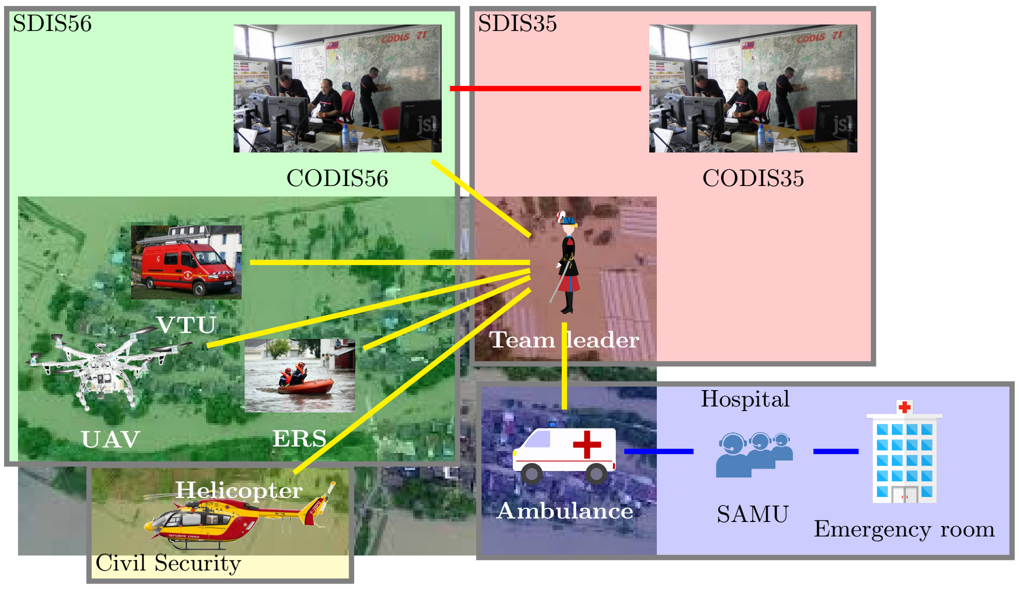


Figure 1: Overview of the French emergency services.

is, direct communication with respect to field actions. When the team leader realizes that the flood is spreading and risks covering several departments, the SoS is reorganized to introduce, e.g., tactical and strategic collaborations with a hierarchical command chain. At the tactical level, commanders give orders to their subordinates and collect reports. At the strategic level, several SDISs ensure consistent direction in addressing high-level objectives.

The question we address in this paper is: how does an architect[1] design reconfiguration in the context of the evolutionary development of an SoS?

Dynamic reconfiguration has been studied mainly in the context of distributed systems based on components. Whereas in the case of distributed systems, the focus is on autonomous reaction to environmental changes and automatic self-adaptation, the evolutionary development of an SoS turns reconfiguration into an engineering activity. When a constituent system decides to quit, the SoS may react and reconfigure itself automatically, but when a new constituent system must be recruited, the architect of the SoS may have to sign contracts for financial or legal reasons. Moreover, a new constituent system may even have to be designed and manufactured. Moreover, the SoS is also reconfigured when new requirements arise, resulting in the need for a new architecture. Therefore, in this paper, we view reconfiguration as the design, implementation, and deployment of any change in the SoS architecture. We extend our previous work [Petitdemange et al., 2018] by proposing a design process to guide the architect in charge of designing a reconfiguration, noting the tasks that must be accomplished to develop a reconfiguration for an evolutionary development. We apply our process to one of the configurations obtained from the case study scenarios.

The reminder of this paper is structured as follows. We present the background and related work in section 2 to introduce the concepts and modeling languages used. This section also reviews how the SoS architecture is modeled in configurations according to our previous work [Petitdemange et al., 2018]. Section 3 describes the dedicated design process for reconfiguration. In section 4, we show how the process is applied to design a reconfiguration in the context of our case study. Section 5 discusses threats to validity of the contribution. Finally, section 6 concludes the paper.

2 Background and related work

2.1 System of systems

Several previous works identified distinctive characteristics of SoSs, among which Maier [1998] specifically insists on operational and managerial independence of the constituent systems. Identifying such characteristics enables specific architectural principles and engineering practices to help SoS architects.

[1]In this paper, when we refer to “an architect” (singular form), we refer to the role, which may be played either by a single person or by a team.

For example, Maier [1998] identified four architectural principles: *"stable intermediate forms"* and *"policy triage"*, borrowed from Rechtin [1990], and *"leverage the interface"* and *"ensuring cooperation"* heuristics, which altogether focus the architects' attention on coping with the operational and managerial independence of the constituents.

Among these principles, the *"stable intermediate forms"* principle advocates that, even before its complete deployment, a SoS should be operable. In other words, evolutionary development means that a SoS is deployed step by step and is operated during the evolution. In each step, the architect evolves the architecture: she/he may have to respond to new requirements, seize new opportunities, and acquire new capabilities. While an evolution is being deployed, the SoS is already operating and providing services. Thus, the architect has to design the reconfiguration such that it does not harm the stakeholders that operate or use the services provided by the SoS.

2.2 The systems modeling language

The *systems modeling language*, SysML [OMG, 2017a], is an extension of the *unified modeling language*, UML [OMG, 2017b], targeted at providing a structured representation for the design of complex systems. Like other OMG modeling languages, SysML defines the concepts involved in the design, as well as graphical representations, in diagrams. At the core of SysML, a system is decomposed into a tree of *blocks*, each of which indifferently models either a software, hardware, human or composite component. To specify how blocks are assembled and interact within the system, each block contains *ports.* When a connector binds two ports together, an *item flow* specifies what kind of data, material, energy, or item is conveyed through the binding. *Allocation* is a dependency link that relates elements from different representations of the system, such as logical and physical representations. SysML defines additional concepts, which we omit because they are not considered in this paper.

Each SysML model of a system is represented by a collection of diagrams, where each kind of diagram focuses on a specific concern. We focus on two kinds of diagrams: *block definition diagrams* (BDDs) describe blocks, their relationships, and dependencies; *internal block diagrams* (IBDs) represent the internal structure of a composite block, that is, connectors and item flows connecting blocks and ports within the composite. We do not describe other kinds of diagrams.

2.3 The unified profile for DoDAF and MODAF

In conjunction with modeling languages such as SysML, the architecture frameworks of the US Department of Defense (DoDAF), the UK Ministry of Defence (MODAF), and NATO (NAF) help system architects to describe the design of a system using an extensive collection of views that cover the entire life cycle. In this paper, we consider the *Unified Profile for DoDAF and MODAF*, UPDM [OMG, 2017c], which is a merged metamodel for the two frameworks. The UPDM defines a collection of more than 40 views gathered into seven viewpoints. Each view in each viewpoint focuses on specific issues, and the architect selects them according to the information she/he wants to convey.

Previous works, such as the DANSE[2] and COMPASS[3] projects, select only a few views as artifacts in their respective design processes: we adopt the same approach in our work.

We consider operational views (OVs), which depict the logical architecture of the SoS, that is, tasks and activities that are required to conduct the operation. OV-1, which belongs to the OV viewpoint, is a use case diagram that enumerates functions and missions expected from the SoS to describe the operational context. Stakeholders involved with the SoS are also elicited. Later in the design process, OV-5 provides an operational activity model. By producing activity diagrams, the architect describes the detailed activities constituting each use case.

In addition to OVs, we also consider systems views (SVs), which describe the constituent systems and resources within the SoS. In this viewpoint, SV-1 describes the resource interaction specification; that is, it enumerates and describes the constituent systems and their interactions. SV-5 provides traceability between operational activities and system functions by allocating activities to constituent systems.

2.4 Modeling of SoS configurations

In this subsection, we describe how the architect produces a "*configuration*". In other sections of the paper, we refer to the "*architecture*". The distinction follows SysML's approach to configurations OMG

[2]`http://danse-ip.eu/`
[3]`http://www.compass-research.eu`

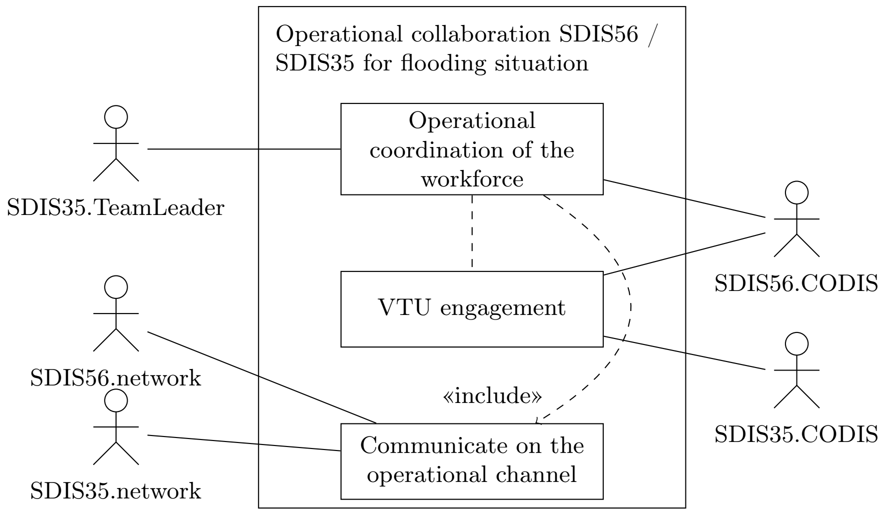

Operational collaboration SDIS56 / SDIS35 for flooding situation
Operational coordination of the workforce
SDIS35.TeamLeader
VTU engagement
SDIS56.CODIS
SDIS56.network
«include»
SDIS35.CODIS
Communicate on the operational channel
SDIS35.network

Figure 2: OV-1 view presented as a use case diagram.

[2017a]: the configuration denotes the actual constituents in the real SoS, as well as actual connections between these constituents, whereas the architecture abstracts over the actual SoS. SysML proposes that the configuration can be modeled as an IBD, in which the block properties identify instances. With respect to the UPDM, the configuration matches SV-1, which is intended to describe constituents and their interactions. We do not give the block properties in the IBD at the end of this subsection.

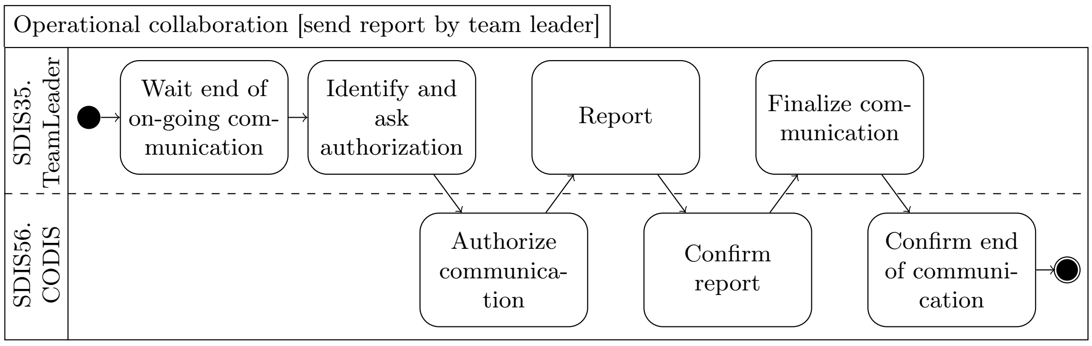
In previous work, we proposed a process targeted at modeling the current configuration of the SoS at a time when reconfiguration is being considered [Petitdemange et al., 2018]. This process selects a few UPDM [OMG, 2017c] views and reuses SysML diagrams [OMG, 2017a] to focus on elements that constitute the source configuration of the reconfiguration. Figure 1 and the accompanying text elicit the operational context of the SoS. The figure provides an overview of the operation that the SoS is intended to support and its general context, according to the mission, that is, the objectives of the operation. The process starts with bounding the SoS using UPDM view OV-1, given as a use case diagram, which for our case is shown in figure 2. In classical systems, the boundary denotes what constituents belong to the system and what constituents are outside the system, in the sense that the architect designs constituents that belong to the system under consideration. Due to the operational and managerial independence of the constituents, all the constituents are outside the SoS. Bounding the SoS, in this context, means deciding to abstract details about some constituents. In the model of our example, figure 2 depicts what actors interact with the functions of the SoS: for instance, SDIS35's team leader and SDIS56's CODIS both interact with the *"operational coordination of the workforce"* use case, an interaction that appears in figure 1. We use actors here to denote that we are not interested in further details about the team leader and the two CODISs. Even if only SDIS56's VTU are used in the operation depicted in figure 1, the CODISs of both SDIS56 and SDIS35 interact with the *"VTU engagement"* use case because both may provide VTU as needed. Likewise, the networks of the two SDISs interact with the *"communicate on the operational channel"* use case, meaning that the operational channel for this SoS is constructed from the interconnection of the two networks.

Then, by reusing UPDM view OV-5 (figure 3), given as a collection of activity diagrams, the architect identifies detailed activities composing each use case. Figure 3 illustrates the view with an activity diagram that describes the scenario when the team leader wants to report to his/her supervising CODIS.

Last, the architect uses UPDM view SV-1 (figure 6 in section 4) to enumerate constituent systems and their interactions. At the end of this process, the architect has modeled the configuration of the operating SoS, and this configuration can serve as the initial configuration for the reconfiguration.

In this paper, we provide only the most relevant and illustrative diagrams. Other diagrams can be found in Petitdemange et al. [2018].

This process forces the architect to address the SoS boundary by adopting a top-down modeling approach. The use case diagram of OV-1 (figure 2) contains the goals (modeled by use cases) of the SoS and the constituents (modeled by actors) that interact to contribute to these goals. The activity diagram of OV-5 (figure 3) helps to fully investigate the boundary of the SoS. It requires the architect to indicate which constituents contribute to each goal of the SoS and what is the contribution of each constituent, modeled by lanes in the activity diagram. This process provides an opportunity for the



Operational collaboration [send report by team leader]
SDIS35. TeamLeader
SDIS56. CODIS
Wait end of on-going communication
Identify and ask authorization
Report
Finalize communication
Authorize communication
Confirm report
Confirm end of communication

Figure 3: One of the activity diagrams belonging to OV-5.

architect to discover possibly missing constituents and, therefore, to ensure the boundary of the SoS is correctly defined. The BDD and IBD of SV-1 (figure 6) yield the configuration the architect intends to model: the BDD details the type of constituents that fulfill the goals previously described in OV-1; the IBD shows the dependencies between involved constituents, whose knowledge will later be useful to ensure the safety and continuity of operation during reconfiguration.

2.5 Dynamic reconfiguration

Previous work on dynamic reconfiguration, such as the seminal definition of the *"quiescent"* state by Kramer and Magee [1990] and subsequent work by, e.g., Vandewoude et al. [2007], Pissias and Coulson [2008], proposed a generic approach. Constituents can be either in an *"active"* state, which is the nominal state, or in a *"passive"* state when they service requests but do not issue requests. When the correct constituents are brought to a passive state during reconfiguration, those constituents affected by reconfiguration are not involved in any communication, thereby ensuring that no message is ever sent to unconnected ports. Thus, message loss and deadlocks in communication protocols are avoided, such that suspended constituents can recover activity after reconfiguration. Meanwhile, nonaffected constituents continue to provide services.

Designing a reconfiguration for this generic approach is fairly simple. Indeed, any reconfiguration begins with a *down* phase, during which the correct constituents are brought to a *passive* state, connections are unbound, and some constituents are removed from the configuration as needed. Then, the reconfiguration proceeds with an *up* phase, during which constituents are recruited in the configuration, connections are bound, and all the constituents are brought back to an *active* state. Boyer et al. [2017], Durán and Salaün [2016] defined reconfiguration in terms of such down and up phases, and Boyer et al. [2017] proposed a formally verified algorithm that automatically generates such reconfigurations. Arshad and Heimbigner [2005], André et al. [2010], da Silva and de Lemos [2011] proposed to automate the generation of reconfiguration in the same setup using generic action planning algorithms. Because reconfiguration via this generic approach is performed systematically or even automatically, no specific work or design process, such as the work described in this paper, is needed in this context.

However, having the SoS authoritatively placing constituents in a *passive* state contradicts the managerial independence of the constituents[4]. Thus, approaches based on a passive state may not be practical in the context of a SoS. Moreover, bringing constituents to a passive state is not the only proposal to proceed with dynamic reconfiguration. For instance, Ma et al. [2011], Ghafari et al. [2012], Wernli et al. [2013] proposed that constituent substitution can be implemented by having both old and new variants coexists during reconfiguration. Doing so mitigates service unavailability in the *passive* state but causes additional difficulties in ensuring the synchronization of coexisting variants. Additionally, Zhang and Cheng [2005] identified several alternative relevant behaviors during reconfiguration, and Buisson et al. [2016] proposed a framework in which constituents can be arbitrarily mutated when required by the reconfiguration.

The diversity of approaches to dynamic reconfiguration, each with its advantages and drawbacks, suggests that some decisions must be made when a reconfiguration is built, that is, the reconfiguration must

[4]Though one may object that, while not bringing a constituent to a *passive* state, a similar effect might be obtained by suspending communications at the boundary of the constituents, e.g., as done by Fractal's component membrane [Bruneton et al., 2006]. But doing so when the connection graph of the components contains cycles is known to be a challenging problem, as shown by, e.g, Boyer et al. [2017].

be designed. The goal of reconfiguration design is not to identify a sequence of reconfiguration actions (which we have seen can be automated according to previous work) but to identify what approaches to use to best fulfill the SoS mission as it evolves.

Furthermore, the evolutionary development of an SoS may involve the acquisition of new constituents, which in turn may include tasks, such as contract negotiation with providers, and manufacturing and deploying of newly acquired constituents. Furthermore, some of the constituents may be human individuals, not only technical systems. Therefore, automating the reconfiguration of an SoS is not as relevant here as it is in the context of some self-adaptive distributed software systems.

2.6 Design of dynamic reconfiguration and SoS evolution

Following their previous work on the specification of behavior during reconfiguration, Zhang and Cheng [2006] proposed a process for the design of self-adaptive systems. Because they consider self-adaptive systems, Zhang and Cheng [2006] separate the definition of the source and target architectures into a context-independent specification and a context-dependent specification. They consider the context-independent specification to be the specification of the system during reconfiguration. However, doing so prevents evolutionary development of the SoS, which may require a complete change of the SoS specification. The work of Zhang and Cheng [2006] helps the architect, as it states that the reconfiguration should begin with steps that restrict the source behavior to guide the system towards the context-independent specification (and towards reaching a quiescent state). However, the architect is not aided in decomposing the reconfiguration into simpler, easier-to-design tasks.

In the work of Buisson et al. [2016], the architect builds the reconfiguration and the proof of its correctness simultaneously. When she/he fails to complete the proofs, the architect analyzes the reasons why the proof is stuck, thereby providing hints to find missing properties that should have been previously established by the preliminary reconfiguration steps. Beyond repeated trials, no specific help is provided to the architect in designing the reconfiguration.

Gomaa et al. [2010], Petitdemange et al. [2016] proposed reconfiguration patterns to document design decisions within reconfiguration. While reconfiguration patterns provide reusable solutions to recurrent problems, these previous works do not provide any accompanying design processes.

The methodology from the DANSE project [Winokur et al., 2015] addresses the evolutionary development of SoSs. On the one hand, the SoS architect can identify parameters within the model of the SoS, as well as in the architecture patterns desired for the SoS. Then, a constraint solver is used to fit these parameters such that the architecture metrics are optimized, resulting in a new architecture. On the other hand, potential changes in the architecture are described by a graph grammar. Application of the graph grammar rules, pruned by architecture validation, results in a path towards the desired target architecture such that the architecture is validated against the SoS mission at each reconfiguration step. The design of the graph grammar is left to the architect with no help.

3 Reconfiguration Design Process

In this section, we present a process for designing an SoS reconfiguration. Figure 4 presents the overall life cycle of reconfiguration in four steps. The first step defines the initial and final architectures for the reconfiguration, when the architect detects that reconfiguration is needed. Then, the second step complements the reconfiguration specification with a description of the requirements that must be satisfied during reconfiguration, as well as the requirements that can be relaxed. The third step is the elaboration of the reconfiguration, which is followed by application in the fourth step.

We detail SoS-specific aspects of each of step in the subsequent subsections.

3.1 Trigger of the reconfiguration

The need for reconfiguration is the first step. In a SoS, two factors may lead to reconfiguration.

On the one hand, in step 1.A.1, some monitoring activity may detect that the configuration of the SoS no longer satisfies the architecture anymore. Indeed, due to managerial independence, the SoS architect cannot force the constituent systems to permanently contribute to the SoS as expected. Instead, systems management can independently decide to remove their own systems from a given SoS, e.g., to perform maintenance operations or to focus on their own operations or on another SoS. In such a situation, the SoS configuration may no longer conform to the architecture. When this situation occurs, reconfiguration must reestablish the SoS architecture, for instance, by recruiting other constituent systems. To do so,

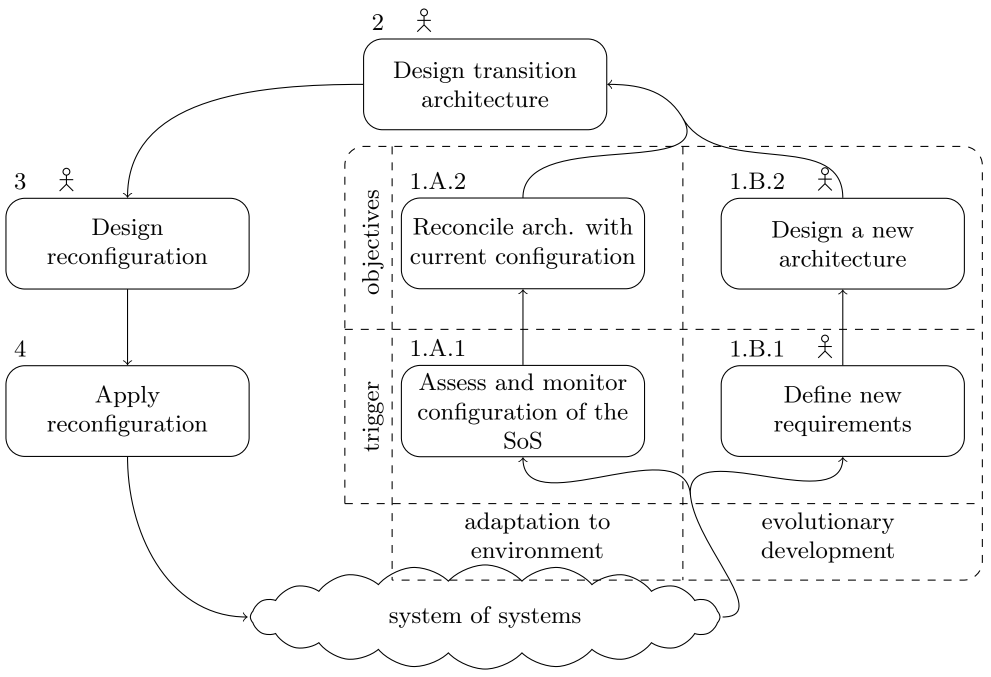

2
Design transition architecture
3
Design reconfiguration
4
Apply reconfiguration
objectives
trigger
1.A.2
Reconcile arch. with current configuration
1.B.2
Design a new architecture
1.A.1
Assess and monitor configuration of the SoS
1.B.1
Define new requirements
adaptation to environment
evolutionary development
system of systems

Figure 4: Lifecycle of an SoS reconfiguration.

in step 1.A.2, the architecture must be reconciled with the actual configuration. From the observed configuration, the architect recovers the actual SoS architecture, that is, the architecture to which the observed configuration conforms. In subsequent steps, the architect uses this reconciled architecture as the starting point for the reconfiguration. Steps 1.A.1 and 1.A.2 correspond to the case when the SoS must be adapted to its environment, e.g., to the willingness of the constituents to contribute to the SoS.

On the other hand, in step 1.B.1, the need for reconfiguration comes from the SoS architect herself/himself when she/he changes the requirements of the SoS. In step 1.B.2, the SoS architect then designs a new architecture to accommodate the newly defined requirements. In this context, reconfiguration aims to reorganize the SoS to establish the new architecture. Steps 1.B.1 and 1.B.2 correspond to evolutionary development.

Although the two scenarios (adaptation to the environment, evolutionary development) appear to be separatin figure 4, they may be combined. For instance, the architect may decide to revise the requirements to be able to adapt when some constituents decide to no longer contribute to the SoS. In this case, instead of reestablishing the architecture, the architect decides that a new architecture must be deployed to respond to the redefined requirements according to the constituent systems that are usable.

As shown by the highlights in figure 4, steps 1.A.1 and 1.B.1 trigger the reconfiguration, and steps 1.A.2 and 1.B.2 ensure that the objectives of the reconfiguration, that is, the initial configuration and target architecture, are known. Because branch 1.A aims to reestablish a broken architecture (e.g., when some constituent system quits the SoS), the target architecture in this branch is already known and need not be designed. By contrast, because branch 1.B aims to revise the deployed architecture, the actual configuration of the SoS still conforms to the architecture. Thus, the initial architecture is already known and need not be modeled again.

Regardless of the reason that triggers reconfiguration, the SoS is expected to be operable during reconfiguration. While reconfiguration cannot performed without having an impact on the SoS, it will not disturb the SoS beyond its architect's acceptance. Therefore, the reconfiguration objectives, which are the initial and target architectures, are not sufficient to specify the reconfiguration.

3.2 Transition architecture

The reconfiguration may not be feasible without breaking the requirements of the SoS. Even worse, the requirements before and after reconfiguration may be incompatible. Therefore, in step 2, the SoS architect defines a *transition architecture* that transiently enlarges the set of admissible configurations. The idea behind the transition architecture is to identify less-important requirements that can be tem-

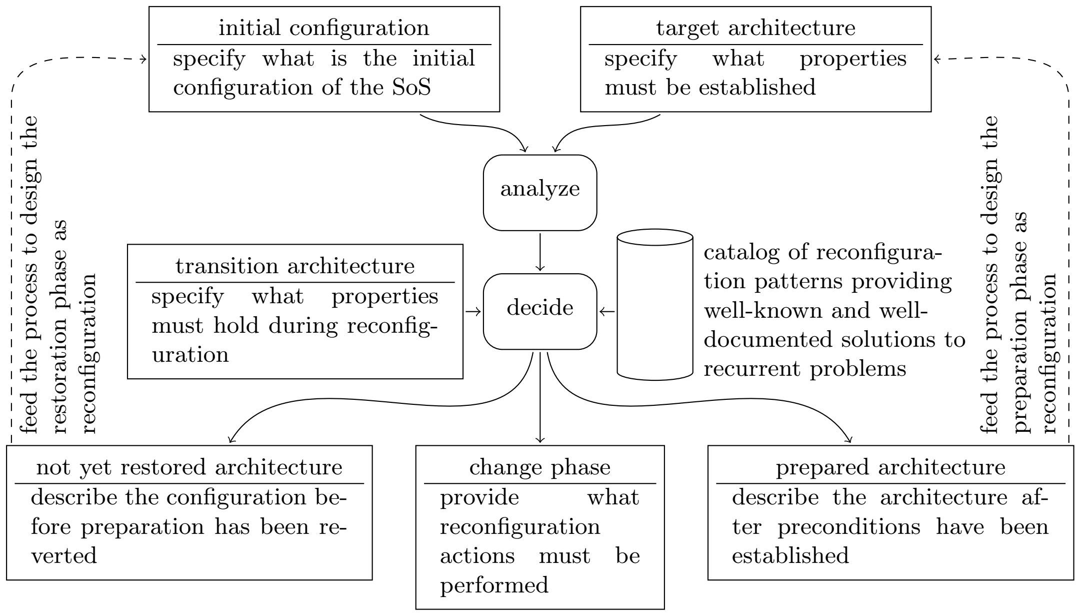

initial configuration
specify what is the initial configuration of the SoS
target architecture
specify what properties must be established
analyze
transition architecture
specify what properties must hold during reconfiguration
decide
catalog of reconfiguration patterns providing well-known and well-documented solutions to recurrent problems
feed the process to design the restoration phase as reconfiguration
feed the process to design the preparation phase as reconfiguration
not yet restored architecture
describe the configuration before preparation has been reverted
change phase
provide what reconfiguration actions must be performed
prepared architecture
describe the architecture after preconditions have been established

Figure 5: Design of an SoS reconfiguration.

porarily relaxed during reconfiguration and to express this relaxation in the architecture. This transition architecture expresses the architect's acceptance of the SoS during reconfiguration when reconfiguration is unfeasible without loosening the requirements. When the reconfiguration is triggered due to an action of constituent systems' management, reconciliation in step 1.A.2 relaxes *de facto* constraints within the architecture, thereby providing a basis for the transition architecture.

At the same time, by means of this transition architecture, the SoS architect specifies what constraints must be enforced even during reconfiguration. The transition architecture may also require constraints or services that do not appear in the SoS architecture, for instance, workarounds needed to balance service degradation during the reconfiguration.

For example, the transition architecture may relax a direct connection between two constituents by requiring only the existence of any communication path between these two constituents. However, for two other constituents, the architect may require that the communication path always includes encryption to ensure confidentiality, even during reconfiguration.

The architect can rely on previous work on requirement engineering, which proposed methodology to analyze and produce relaxed requirements, to produce the transition architecture. For instance, Whittle et al. [2010] proposed the RELAX requirement language that implements operators *AS EARLY* (or *LATE*, *CLOSE*, *MANY*, or *FEW*) *AS POSSIBLE* based on fuzzy branching temporal logic [Moon et al., 2004] to express requirements that can be relaxed. A requirement using these operators denotes that some quantifiable property is required but that an approximate value is sufficient if no better value can be implemented by the SoS, as specified by a fuzzy set. By defining such a requirement, the architect accepts that the requirement can be relaxed to any degraded value in a fuzzy set. As an extension of RELAX, Viana et al. [2016] proposed an extensive list of methods to resolve requirement conflicts. In our case, these would be conflicts between SoS requirements and reconfiguration requirements. RELAX's operators *BEFORE* and *AFTER* can be used to specify the dynamics during the whole reconfiguration such that the architect can express, for instance, that a sequence of degradations is acceptable, whereas their simultaneous occurrence is not.

The transition architecture implements such relaxed and sequenced requirements. Rather than connected instances, the constraint-based architecture description [Georgiadis et al., 2002, Waewsawangwong, 2004, Guessi et al., 2016] enables the architect to describe the space in which the SoS configuration is allowed to evolve during reconfiguration. The space initially includes the initial architecture and eventually includes the target architecture.

3.3 Design of the reconfiguration

Altogether, the transition architecture (resulting from step 2) and the initial and target architectures (resulting from steps 1.A.2 and 1.B.2) form the specification of the reconfiguration. In step 3, the architect designs a conforming reconfiguration; that is, she/he selects an overall approach to implement the changes to the SoS while it is operable, ultimately yielding reconfiguration instructions.

According to previous work concerning dynamic reconfiguration, some preparation may be needed to transiently bring the architecture into a suitable state during reconfiguration before changes can occur. For instance, Kramer and Magee [1990] proposed placing some constituents in a *passive* state, that is, switching these constituents to a mode of operation such that they do not issue any requests but continue to service incoming requests. Placing some constituents in a *passive* state results in a subset of these constituents reaching a *quiescent* state, in which they do not receive any incoming requests. Quiescent constituents are not involved in any communication, so they can safely be disconnected and reconnected in the architecture. Then, after the changes have been made, constituents are returned to an *active* state in which they perform normally.

To account for such preparation, Durán and Salaün [2016], Boyer et al. [2017] decompose reconfigurations into a *down* phase consisting of passivating, disconnecting and removing constituents and an *up* phase consisting of adding, connecting and activating constituents. As a more general approach, Buisson et al. [2016] proposed that constituents switch from one implementation to another when required by the reconfiguration, where passive and active states are simply two specific implementations. Implementation switching may affect ports, including the addition or removal of ports, and an implementation switch may also alter the constraints, e.g., making a port disconnectable or mandating that it be connected.

As illustrated by cited works, any reconfiguration prepares the architecture to enable the changes; then, after changes have been performed, the reconfiguration returns the architecture to normal operation. That is, a reconfiguration is composed of three phases: *preparation*, *changes*, and *restoration*. Kramer and Magee [1990], Durán and Salaün [2016], Boyer et al. [2017] considered the preparation and restoration phases to consist simply of switching between active and passive states. Buisson et al. [2016] viewed the preparation phase as any arbitrary modification that establishes the preconditions of the change phase. Thus, the preparation and restoration phases are also reconfigurations.

Accordingly, in step 3, we advocate the process described in figure 5. First, analysis of the specification of the reconfiguration results in the set of constituent systems to be integrated within the SoS and those constituent systems to be excluded. To do so, the architect compares the initial configuration of the SoS to the target architecture she/he wants to establish in the SoS. Furthermore, changes to connections and mediating entities are identified at the same time.

Depending on the identified changes and the constraints of the transition architecture, the architect must develop an overall approach to the reconfiguration under consideration. For instance, the architect decides whether some constituent systems should be transitioned to a quiescent state and if some constituent systems should be transiently used to ensure continuity of operation. The architect also schedules atomic reconfiguration operations such that changes are performed in the correct order for the transition architecture. To help in this task, a catalog of reconfiguration patterns [Petitdemange et al., 2016] prescribes well-known solutions to recurrent reconfiguration problems. Reconfiguration patterns provide documentation of these well-known solutions and notably concerning properties, such as service availability or degradation, such that the architect can more easily compare with the constraints specified in the transition architecture. Patterns also document assumptions made in the solution, such that the architect can match against the actual capabilities of (independently engineered) constituent systems.

Once the design decision is made, the preconditions of the reconfiguration might be unsatisfied by the source architecture, and it may be possible that the preconditions are established by preceding reconfiguration actions. For instance, the decision to rely on quiescence requires that some constituent systems are passive, while the *coevolution* pattern[5] [Petitdemange et al., 2016] requires the presence of a specific mediating entity that synchronizes the internal states of two constituent systems. In such cases, a preparation phase is required to establish the precondition, and a restoration phase may be needed to revert transient changes, e.g., to return constituent systems to an active state or to remove the state synchronization machinery.

Preparation and restoration can be considered to be reconfigurations. In figure 5, *prepared architecture* and *not yet restored architecture*, which denote the architecture after preparation and before restoration, respectively, are used to feed and recursively reapply the design process to design the preparation and restoration phases. The preparation phase is the reconfiguration that transitions the SoS from its initial configuration to the *prepared architecture*, and the restoration phase is the reconfiguration that transitions

[5]The *coevolution* pattern [Petitdemange et al., 2016] advocates that when a constituent has to be replaced with another, the two constituents coexist until the former becomes fully unused. Meanwhile, the two constituents mutually synchronize their respective internal states. This pattern models and documents techniques of Ma et al. [2011], Ghafari et al. [2012], Wernli et al. [2013].

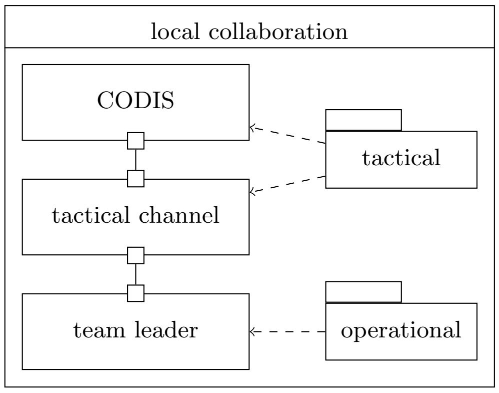


Figure 6: Initial configuration for the case study.

the SoS from the *not yet restored architecture* to the target architecture.

Recursive reapplication of the design process terminates when a reconfiguration is sufficiently simple such that it does not require preparation or restoration.

3.4 Execution

When the design of the reconfiguration is completed in step 4, it is submitted to the SoS for application. As this paper focuses on the design process, we do not discuss the execution of the reconfiguration.

4 Example

To illustrate how an SoS architect applies the process detailed in Section 3, we consider the reconfiguration design for the rescue operation presented in the introduction. We assume that the initial configuration of the SoS is the one shown in figure 6: the CODIS communicates directly through a tactical channel with team leaders concerning field actions performed to fulfill the mission. This configuration is typically the first deployed upon the occurrence of an incident when the severity is low or still unknown.

The following subsections follow the steps shown in figure 4.

4.1 Trigger and objectives

The scenario we consider to illustrate the process is evolutionary development; therefore, the architect applies only step 1.B (see figure 4).

Step 1.B.1: Define new requirements When a crisis becomes more serious than initially anticipated, the requirements assigned to the SoS must evolve to take into account the evolution of the crisis. For instance, local rescue services may no longer be sufficient to handle the crisis. In such a case, the architect requires an interdepartmental collaboration of rescue services from several neighbor departments.

Step 1.B.2: Design a new architecture This new requirement leads the architect to design the target architecture shown in figure 7. This evolved architecture introduces a group leader who is in charge of tactical collaboration such that CODIS can focus solely on strategic collaboration.

The strategic collaboration ensures that several collaborating SDISs adopt consistent directions to address the higher-level objectives defined by the mission. The group leader ensures tight coordination of field resources.

Regardless of the exact configuration, the architect models the communication discipline of the hierarchical command chain with the following constraints:

- Constraint 1: Each resource is a member of exactly one group. The *strategic* group contains resources that determine the resource allocation; the *tactical* group contains control and command resources; the *operational* group contains resources that operate directly in the field. We choose to model groups using packages (on the right-hand side of figures 6 and 7), along with architectural primitives of Zdun and Avgeriou [2008], which we omit to avoid overloading the diagrams.

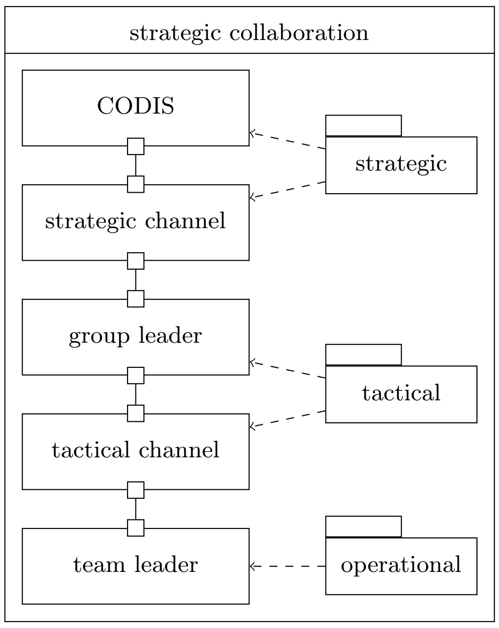


Figure 7: Target architecture for the case study.

- Constraint 2: A resource cannot be connected directly to another resource from another group; it can only be connected to another resource via a proxy. In our use case, communication channels play the role of proxies.
- Constraint 3: A resource can only access communication channels (or proxies) of groups at the level immediately above or immediately below its group. A resource can also communicate with other resources in the same group as itself. In the example, tactical resources can communicate with both operational and strategic resources, but operational resources cannot communicate with strategic resources.

4.2 Design transition architecture

Once the initial configuration and the target architecture are known, the architect collects constraints on the SoS during reconfiguration; that is, she/he designs a transition architecture. In the example:

- Transition constraint 1: Supervision of the team leader cannot suffer any discontinuity: no report from the team leader to her/his supervisor can be lost.
- Transition constraint 2: The direct supervisor (CODIS in the initial configuration, group leader in the target architecture) must have correct situation awareness such that the orders she/he gives to the team leader are always consistent with the real situation in the field.
- Transition constraint 3: At any time, to avoid any contradictory orders, exactly one supervising authority can give orders to the team leader.

Even if the architect takes into account the constraints identified in the architecture, the above-described communication discipline does not raise any specific issues: without relaxing the constraints, it can be addressed by considering that CODIS switches from tactical to strategic as soon as the team leader reports to the group leader.

4.3 Design reconfiguration

Figure 8 summarizes the complete reconfiguration and how it is designed. The reconfiguration transforms the SoS from the configuration of figure 6 to the architecture of figure 7 (dotted arrow labeled *reconf*).

The design process of subsection 3.3 is applied to refine the reconfiguration via a preparation phase (dashed arrow labeled *reconf/prep*) that transforms the SoS to the architecture of figure 9, followed by a change phase (arrow labeled *reconf/change*) that transforms the SoS to the architecture of figure 12,

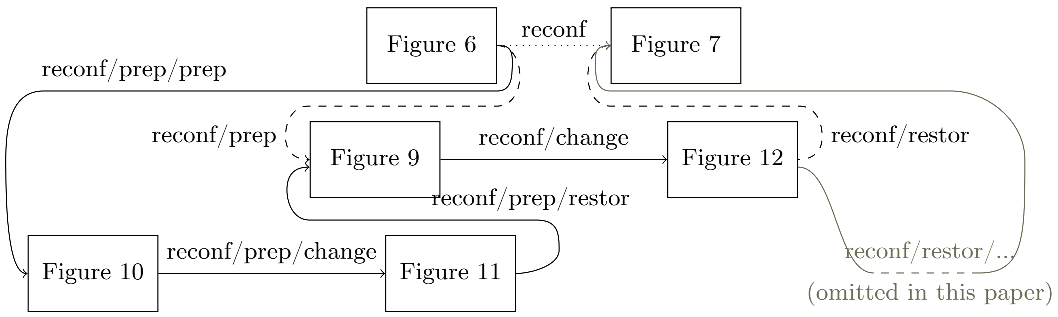


Figure 8: Construction of the reconfiguration by the proposed design process at step 3.

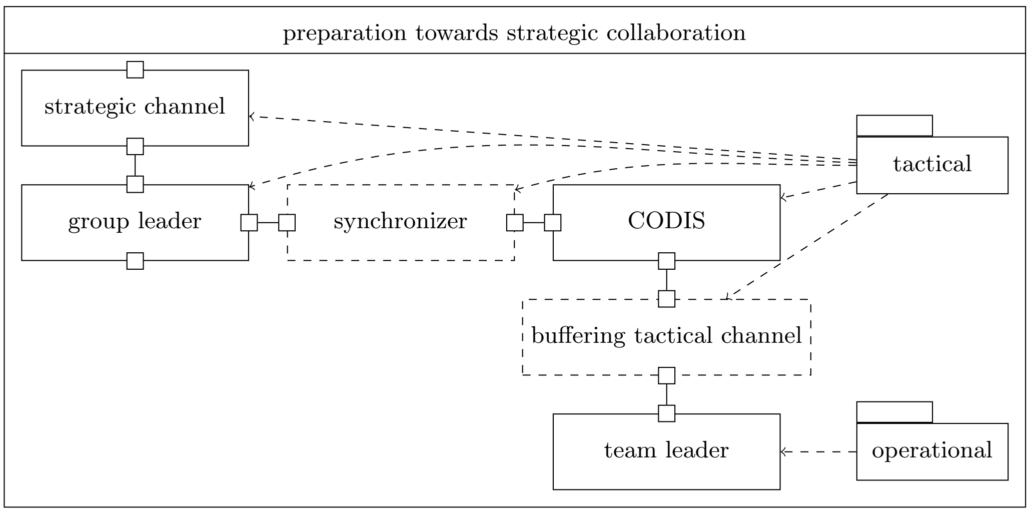


Figure 9: Prepared architecture.

followed by a restoration phase (dashed arrow labeled *reconf/restor*) that completes the reconfiguration and results in the architecture of figure 7. In this and the following figures, *reconf* denotes reconfiguration, *prep* is a preparation phase, *change* is a change phase, and *restor* is a restoration phase. The / symbol denotes a nesting of phases within a refined phase.

According to subsection 3.3, the preparation and restoration phases are reconfigurations and, therefore, might be further refined by applying the same design process. In figure 8, the preparation phase is further refined in this way: the preparation phase of the preparation phase (arrow labeled *reconf/prep/prep*) transforms the SoS to the architecture of figure 10, followed by a change phase (*reconf/prep/change*) that transforms the SoS to the architecture of figure 11 and a restoration phase (*reconf/prep/restor*) that yields the architecture of figure 9. The restoration phase *reconf/restor* is refined similarly. We omit the details in the paper.

In the following discussion, we explain how the architect designs and constructs this reconfiguration. We use a systematic nomenclature for the headings to provide a structured description. The step and substep numbers are followed by headings that refer to the labels of the arrows in figure 8, followed by the activity or artifact name in figure 5. For instance, "**3 - 1: reconf: Analyze**" means that step 3 substep 1 represents the *Analyze* activity in figure 5 in the context of the *reconf* arrow in figure 8, i.e., in the context of the global reconfiguration.

Step 3 - 1: reconf: Analyze Once the desired reconfiguration has been specified by the target architecture and the transition architecture, the architect analyzes the specification. By comparing the initial configuration and the target architecture, the architect finds that the reconfiguration must recruit a group leader and instantiate a strategic channel. The reconfiguration must also disconnect the CODIS from the tactical channel and establish new connections according to figure 7.

Step 3 - 2: reconf: Decide Consequently, according to the constraints described in the transition architecture, the architect decides to consider the *coevolution* pattern of Petitdemange et al. [2016]. In

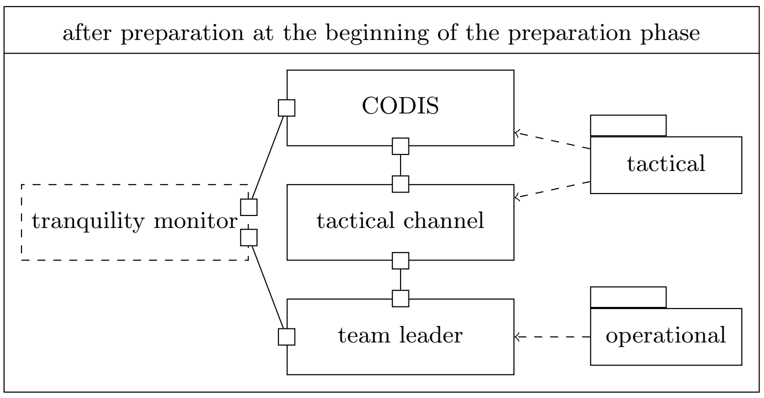


Figure 10: Prepared architecture for the preparation phase.

the general context of a hierarchical command chain, the coevolution pattern addresses the case where the command-and-control (C2) system must be replaced with a new C2 while avoiding any discontinuity of supervision (transition constraint 2).

In the case of the reconfiguration under consideration, the old C2 is played by CODIS, and the new C2 is played by the group leader. Team leaders play the role of operators under C2 supervision.

To avoid suspending services, the pattern proposes that operators under the supervision of the replaced C2 are migrated one by one to the supervision of the new C2. To allow reconnection of the team leader without disruption, the tactical channel mediating element is mutated to include a buffering facility. During reconfiguration, the two C2s, that is, both CODIS and the group leader, coexist in the SoS configuration. At any time, each team leader is connected to either CODIS or the group leader but never to both. In this way, the solution described in the pattern ensures that no team leader receives contradictory orders (transition constraint 3). The solution of the pattern also requires that the two C2s (here CODIS and group leader) exchange the reports they receive from operators (here team leaders) to synchronize their respective view of the situation. The coevolution pattern requires a specific mediating entity that is devoted to this task. Initially, this mediator migrates the internal state of the old C2 (CODIS) to the new C2 (group leader). Then, each time a C2 (CODIS or group leader) receives a report from an operator (team leader), the mediator intercepts the message to detect the desynchronization. The mediator propagates intercepted messages to the other C2 to resynchronize the states. By means of this mediator behavior, the solution proposed by the pattern ensures that no report is ever lost (transition constraint 1) and ensures situation awareness of the C2 (transition constraint 2).

Step 3 - 3: reconf/change According to the above-described decision made by the architect, the change phase of the reconfiguration consists of individually reconnecting team leaders from the CODIS to the group leader.

Step 3 - 4: reconf/prepared architecture The change phase assumes that the SoS has already recruited the group leader and that a mediating element ensures the prescribed state synchronization between CODIS and the group leader. Figure 9 shows the prepared architecture according to this precondition. Dashed boxes represent transient mediating elements that are used only during reconfiguration.

The architect reapplies the process of figure 5 to design the preparation phase as a reconfiguration that brings the SoS from its initial configuration depicted in Figure 6 to the architecture given in Figure 9.

Step 3 - 5: reconf/prep: Analyze A comparison of the initial configuration of figure 6 and the prepared architecture of figure 7 enable the architect to observe that a *synchronizer* mediating element must be inserted into the architecture and that the tactical channel must be replaced.

Step 3 - 6: reconf/prep: Decide On the one hand, addressing the synchronizer does not raise any challenges: assuming that CODIS and the group leader have the required capabilities, the mediating element can be instantiated and connected without any precaution.

On the other hand, enabling a buffering facility in the tactical channel requires in-place substitution. To perform this reconfiguration with minimal disruption, the architect decides to use the *tranquility*

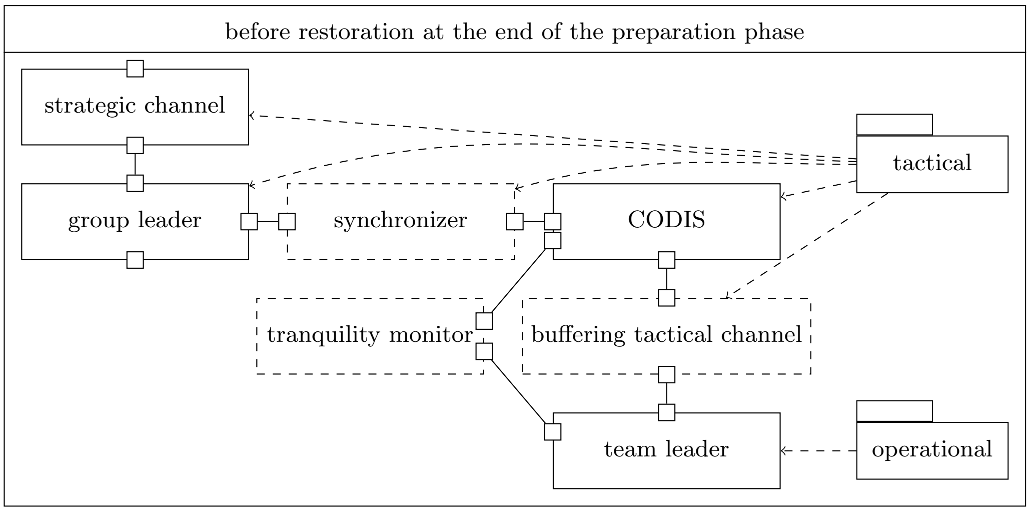


Figure 11: Not-yet-restored architecture for the preparation phase.

approach of Vandewoude et al. [2007] such that mutation of the tactical channel occurs opportunistically when neither CODIS nor the team leader have to communicate.

Step 3 - 7: reconf/prep/change When the reconfiguration detects that the tactical channel is (temporarily) unused, the channel is disconnected and replaced with a buffering channel, which is, in turn, connected to the architecture.

Step 3 - 8: reconf/prep/prepared architecture Assuming that the SoS does not provide the necessary infrastructure to implement the tranquility approach, the preparation phase itself requires preparation to transform the SoS to the architecture of figure 10.

Step 3 - 9: reconf/prep/prep: Analyze A comparison of the initial configuration of figure 6 and the architecture of figure 10 shows that the only change is the instantiation and connection of the tranquility monitor.

Step 3 - 10: reconf/prep/prep: Decide The architect can reasonably assume that CODIS and the team leader can report to the tranquility monitor when they are not involved in any communication through the tactical channel. Thus, the change in the preparation of the preparation of the reconfiguration does not require any further preparation or restoration. This change is straightforward.

Step 3 - 11: reconf/prep/not yet restored architecture A restoration phase removes the tranquility monitor that was transiently inserted into the SoS. This restoration phase starts with the architecture of figure 11 and transforms the SoS to the architecture of figure 9.

Step 3 - 12: reconf/not yet restored architecture Figure 12 shows the architecture obtained after the change phase. Transient mediating elements (synchronizer and buffering facility) are still present and have to be removed before the final architecture of figure 7 is effectively reached by the SoS.

Step 3 - 13: reconf/restor This restoration step is similar to "reconf/preparation", as it reverts the tactical channel to its nominal variant and removes the state-synchronizing mediating element between CODIS and the group leader. We omit the details in this paper.

4.4 Apply reconfiguration

The overall reconfiguration obtained by combining these operations and following the above-described design follows the solid-line arrows in figure 8. The process adapts the SoS through the sequence of architectures starting with figure 6, applying *reconf/prep/prep*, followed by figure 10, *reconf/prep/change*,

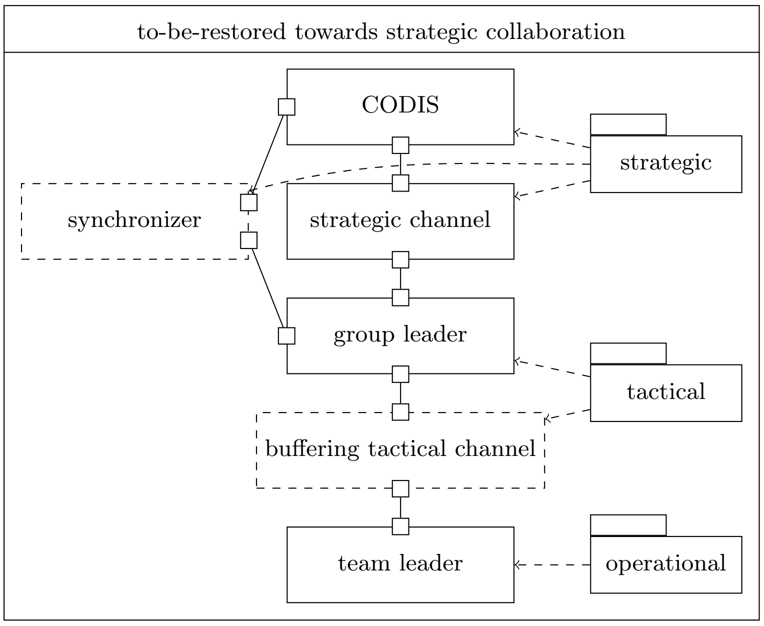


Figure 12: Not yet restored architecture.

figure 11, *reconf/prep/restor*, figure 9, *reconf/change*, figure 12, and finally, *reconf/restor*, yielding to the target architecture of figure 7.

The architect can be further refined as needed.

5 Threat to validity

To preemptively mitigate the threats to validity, we base our work on a realistic scenario. The organizational structure we describe is the actual one in France. The reconfiguration scenario models crisis management escalation, which is a typical scenario for crisis management in general, not only emergency rescue.

In the following subsection, we discuss the internal and external validity of our work.

5.1 Threat to internal validity

One may object that, in the particular situation we describe (flood), high severity may be suspected even before the first field actions. We also omit anticipation, noticeably the effect of weather alerts, which would typically lead to predeployment of field resources, group leaders, and political-level coordination (the next levels of escalation). We believe that this simplification does not harm the validity of our work in the context of the case we studied.

Our approach appears to be difficult to test in the context of a real emergency operation. To mitigate this inability, we use a scenario-based technique and simulation to validate our reconfiguration process. We proceed step by step through the process of reconfiguration of one of the configurations. In comparison to the real system, simulation can be used to verify that the reconfiguration actually conforms to the constraints expressed by the transition architecture. Our ad hoc simulator monitors the architecture of the simulated SoS in this respect, and the constraints are satisfied, even during reconfiguration.

To further improve our approach, two experiments could be conducted in future work. On the one hand, the experience feedback of a rescue operation could be exploited to develop a scenario of a real operation. Even without considering the real-time aspect, such experimentation would allow us to compare our approach for the evolution of the SoS to the reality of the rescue operation. On the other hand, setting up an experiment during a training exercise would allow us to test our approach with the time constraints of an actual operation.

5.2 Threat to external validity

The main limitation regarding external validity is that we played the roles of both architect and reconfiguration engineer ourselves. Further experiments with several architects are needed to qualify and quantify the actual assistance provided to the architect for the design of the reconfiguration.

Further experiments with other cases would also allow us to ensure that the design process is sufficiently general to accommodate any (class of) SoS.

This threat is partly mitigated by the fact that some of our reconfiguration patterns are inspired by existing techniques from related work on the reconfiguration of software systems.

6 Conclusion

The presented work is part of a project to address the question: How should the architect proceed to evolve a SoS after deployment as part of evolutionary development? In our previous work, we proposed an SoS architecture modeling framework to obtain different SoS configurations [Petitdemange et al., 2018]. Configurations are useful to study how an architect could address reconfiguration in an SoS environment. This leads us to define the concept of reconfiguration pattern [Petitdemange et al., 2016] and reconfiguration process in section 3 of this paper. We explained that the design of a reconfiguration requires the definition of constraints by the architect for reconfiguration. These constraints evolve during the reconfiguration and are conceptualized by a transition architecture.

This view of reconfiguration led to the development of an iterative and recursive reconfiguration process. The proposed process helps to guide the architect in the design of the reconfiguration scripts, structuring the activities and controlling the effects of reconfiguration on the reconfigured SoS. These activities consist of preparation, change and restoration phases. The preparation and restoration phases can be considered as reconfiguration. This recursion forces the architect to explicitly document service degradation of the SoS and to control the impact of the reconfiguration on SoS operation.

In the future, we envisage a second set of experiments that will make it possible to qualify and quantify the assistance provided to the architect for the design of the reconfiguration.

The patterns we have defined are all at the same level of granularity. The *coevolution* pattern addresses the global objective of the reconfiguration. By contrast, the *quiescence* pattern provides a solution to a problem that is only a small reconfiguration step. Further investigation could provide information to answer the question of the composition of reconfiguration patterns.

References

Françoise André, Erwan Daubert, Grégory Nain, Brice Morin, and Olivier Barais. F4plan: An approach to build efficient adaptation plans. In *7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 386–392, 2010. doi: 10.1007/978-3-642-29154-8_47.

Naveed Arshad and Dennis Heimbigner. A comparison of planning based models for component reconfiguration. Technical report, University of Colorado, 2005.

Fabienne Boyer, Olivier Gruber, and Damien Pous. A robust reconfiguration protocol for the dynamic update of component-based software systems. *Software: Practice and Experience*, 47(11):1729–1753, 2017. doi: 10.1002/spe.2499.

Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12): 1257–1284, September 2006. ISSN 1097-024X. doi: 10.1002/spe.767.

Jérémy Buisson, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. Safe reconfiguration of coqcots and pycots components. *Journal of Systems and Software*, 122:430–444, 2016. doi: 10.1016/j.jss.2015.11.039.

Carlos Eduardo da Silva and Rogério de Lemos. A framework for automatic generation of processes for self-adaptive software systems. *Informatica*, 35(1):3–13, 2011.

Francisco Durán and Gwen Salaün. Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122:524–537, 2016. doi: 10.1016/j.jss.2015.09.020.

Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising Software Architectures for Distributed Systems. In *Proceedings of the First Workshop on Self-healing Systems*, pages 33–38, 2002. ISBN 978-1-58113-609-8. doi: 10.1145/582128.582135.

Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi, and Hassan Haghighi. An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 177–182, 2012. doi: 10.1145/2304736.2304765.

Hassan Gomaa, Koji Hashimoto, Minseong Kim, Sam Malek, and Daniel A. Menascé. Software adaptation patterns for service-oriented architectures. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 462–469, 2010. doi: 10.1145/1774088.1774185.

M. Guessi, F. Oquendo, and E. Y. Nakagawa. Checking the architectural feasibility of Systems-of-Systems using formal descriptions. In *2016 11th System of Systems Engineering Conference*, pages 1–6, June 2016. doi: 10.1109/SYSOSE.2016.7542939.

Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990. doi: 10.1109/32.60317.

Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*, pages 245–255, 2011. doi: 10.1145/2025113.2025148.

Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998. doi: 10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

Seong-ick Moon, K. H. Lee, and Doheon Lee. Fuzzy Branching Temporal Logic. *IEEE Transactions on Systems, Man, and Cybernetics Part B*, 34(2):1045–1055, April 2004. ISSN 1083-4419. doi: 10.1109/TSMCB.2003.819485.

OMG. Systems Modeling Language version 1.5. Technical Report formal/2017-05-01, OMG, May 2017a.

OMG. Unified Modeling Language version 2.5.1. Technical Report formal/2017-12-05, OMG, December 2017b.

OMG. Unified Profile for DoDAF and MODAF version 2.1.1. Technical Report formal/2019-05-04, OMG, May 2017c.

Franck Petitdemange, Isabelle Borne, and Jérémy Buisson. Assisting the evolutionary development of sos with reconfiguration patterns. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, page 9, 2016.

Franck Petitdemange, Isabelle Borne, and Jérémy Buisson. Modeling system of systems configurations. In *13th Annual Conference on System of Systems Engineering*, pages 392–399. IEEE, June 2018. ISBN 978-1-5386-4876-6. doi: 10.1109/SYSOSE.2018.8428737.

Petros Pissias and Geoff Coulson. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *IET Software*, 2(4):348–361, 2008. doi: 10.1049/iet-sen:20070046.

Eberhardt Rechtin. *Systems Architecting: Creating and Building Complex Systems.* Prentice Hall, 1990. ISBN 978-0-13-880345-2.

Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007. doi: 10.1109/TSE.2007.70733.

Thiago Viana, Andrea Zisman, and Arosha K. Bandara. Towards a Framework for Managing Inconsistencies in Systems of Systems. In *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, pages 8:1–8:7, 2016. ISBN 978-1-4503-6399-0. doi: 10.1145/3175731.3176177.

P. Waewsawangwong. A constraint architectural description approach to self-organising component-based software systems. In *Proceedings. 26th International Conference on Software Engineering*, pages 81–83, May 2004. doi: 10.1109/ICSE.2004.1317430.

Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. *Journal of Object Technology*, 12(3):1: 1–27, 2013. doi: 10.5381/jot.2013.12.3.a1.

Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15 (2):177–196, 2010. doi: 10.1007/s00766-010-0101-0.

Michael Winokur, Revital Goldberg, Nir Ben Dov, Leonardo Mangeruca, Roberto Passerone, Valerio Senni, Christoph Etzien, Tayfun Gezgin, Thomas Peikenkamp, Martin Jung, Arnold Alexandre, René Bullinga, Sanduka Imad, Eric Honour, Stéphane Paul, Sebastian Klaas, Benoît Boyer, and Stephanie Kemper. DANSE Methodology V03. Technical Report D4.4, February 2015.

Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information & Software Technology*, 50(9-10):1003–1034, 2008. doi: 10.1016/j.infsof.2007.09.003.

Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005. doi: 10.1145/1082983.1083220.

Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering*, pages 371–380, 2006. doi: 10.1145/1134285.1134337.