This is a post-peer-review, pre-copyedit version of an article published in Software and Systems Modeling.
The final authenticated version is available online at: http://dx.doi.org/10.1007/s10270-019-00764-7

Ark: a Constraint-based Method for Architectural Synthesis of Smart Systems

Milena Guessi · Flavio Oquendo · Elisa Yumi Nakagawa

Received: date / Accepted: date

Abstract As smart systems leverage capabilities of heterogeneous systems for accomplishing complex combined behaviors, they pose new challenges to traditional software engineering practices that considered software architectures to be mostly static and stable. The software architecture of a smart system is inherently dynamic due to uncertainty surrounding its operational environment. While the abstract architecture offers a way to implicitly describe different forms taken by the software architecture at run-time, it is still not sufficient to guarantee that all concrete architectures will automatically adhere to it. To address this issue, this work presents a formal method named Ark supporting the architectural synthesis of smart systems. This is achieved by expressing abstract architectures as a set of constraints that must be valid for any concrete architecture of the smart system. This way, we can benefit from existing model-checking techniques to guarantee that all concrete architectures realized from such an abstract model will comply with well-formed rules. We also describe how this method can be incorporated to a model-driven approach for bridging the gap between abstract and concrete architectural models. We demonstrate our method in an illustrative case study, showing how Ark can be used to support the synthesis of concrete architectures as well check the correctness and completeness of abstract architecture descriptions. Finally, we elaborate on future directions to consolidating a process for the synthesis of run-rime architectures that are correct-by-construction.

Keywords Smart system · Software Architecture · Formal Method · Architectural Synthesis · Constraints · Alloy

1 Introduction

A smart system combines elements of sensing, acting, and control that can lead to the development of innovative solutions for addressing social, economic, and environmental challenges [63]. To provide more sophisticated capabilities, smart systems require support for [22]: (i) instrumentation, which enables to collect timely, high-quality data through embedded sensors and enact planned strategies accordingly; (ii) interconnection, which enables to create links among data, systems, and people; and (iii) intelligence, which enables to devise new computing models, algorithms, and analytics to process collected data and support decision making. In this scenario, software plays an important role in smart systems for dynamically interconnecting systems and leveraging individual capabilities to accomplish a desired combined behavior. In this scenario, complex smart systems, such as smart cities [42], smart devices, and smart grids [60], can also be considered as Systems-of-Systems (SoS), i.e., systems that are formed by heterogeneous and independent systems [5, 31].

Software architectures are valuable assets to cope with the increasing complexity of smart systems. The

This work was supported by the São Paulo Research Foundation (FAPESP), grants 2012/24290-5, 2017/22107-2, and 2017/06195-9.

Milena Guessi
ICMC, University of São Paulo, Brazil
IHPME, University of Toronto, Canada
E-mail: milena@icmc.usp.br

Flavio Oquendo
IRISA - UMR CNRS/Université de Bretagne-Sud, France
E-mail: flavio.oquendo@irisa.fr

Elisa Yumi Nakagawa
ICMC, University of São Paulo, Brazil
E-mail: elisa@icmc.usp.br

software architecture is embodied in the parts of a system and their relationships with each other, and to the environment, as well as in the principles guiding its design and evolution [27]. The activity concerned with the synthesis of a software architecture is supported by architecture descriptions [24], which comprise the set of artifacts documenting the software architecture and often include several models [27].

Specifically for SoS engineering, three types of models can be useful [68]: (i) *normative models*, describing norms and standards for how SoS should be; (ii) *descriptive models*, describing how SoS are by showing deviations from the normative model; and (iii) *prescriptive models*, describing how one can achieve the normative model given the descriptive model. For instance, normative models can be used for the dissemination of best practices that will be used as the basis for the evaluation of a descriptive model and/or refinement of a prescriptive model. As normative models, *abstract architectures* can be used to define, at design time, the baseline for dynamically interconnecting systems within the SoS. In turn, a *concrete architecture* can be regarded as a descriptive model of a particular setting, which is also referred to as a coalition. The coalition defines how individual actions of constituent systems are to be combined within the SoS for accomplishing its mission. Thereby, a number of concrete architectures can be created and validated against the needs of a smart system.

The software architecture of SoS represents a departure from traditional software engineering practices that assumed software architectures to remain relatively stable throughout their entire life cycle [58, 61]. Because the evolution of SoS architectures can have deeper repercussions in regards to both its structure and behavior, tailored means are needed to support *rearchitecting*, a process that encompasses the modification, substitution, reconstruction, and/or addition of any element of the software architecture [1]. Specifically in the safety-critical domain, such as in a smart system for monitoring environmental conditions to determine whether there is an imminent risk of harming people and/or damaging homes and businesses, a rigorous rearchitecting approach is essential for guaranteeing that the system will continue to behave as expected despite changes to its constituent systems. Moreover, it can help prevent that changes to the software architecture can eventually put the system's mission at risk.

Architecture descriptions can play a central part of a rigorous approach for the evolution of software architectures [39]. For instance, target architectures can be represented using formal notations and change requests can be defined as refinement relations between intermediary architectures, thus guaranteeing consistent models of the system throughout its entire life cycle [9, 41].

The nature of SoS complicates the adoption of traditional practices that have been employed in the description of SoS software architectures [58]. First, the emergent behavior of a SoS requires models that can simulate the behavior of a coalition at run-time, which makes solely static, design time models unsuitable for capturing the dynamism of SoS architectures [67]. Second, the evolutionary development of a SoS requires software architectures that can support multiple coalitions to emerge over time [7, 51]. In this sense, dynamic software architectures can be investigated for promoting SoS resilience in face of uncertainty [14, 49, 51]. Nonetheless, most notations used for describing software architectures, including formal notations (i.e., Architectural Description Languages, ADLs) [46] and semi-formal notations (e.g., UML), only support the description of concrete architectures [18, 51]. Third, the frequent pace in which evolution takes place in SoS can accelerate the architecture decay, which is linked to the degradation of internal and external system properties [1]. For instance, smart system developers must check if architectural properties, such as performance or safety, and external conditions (e.g., geographical distribution of constituent systems), that were defined at the abstract architecture are preserved by the concrete architecture. As a consequence, synthesizing, assessing, and comparing alternative concrete architectures becomes a challenging and time consuming task for SoS designers [11].

Aiming to support the design and development of smart systems, tailored means are needed to detect when a concrete architecture deflects from its normative model. In our previous work [20], we experimented the formal method Alloy [30] for investigating whether a concrete architecture complying with an abstract architecture existed or not, which aligns with the research on autonomous systems architecture [39], i.e., providing mechanisms to either automatically realize a concrete architecture for which purposes, properties, and constraints of the abstract architecture description are satisfied, or reporting that such an architecture is not feasible. In particular, the abstract architecture of the SoS was expressed in SosADL [56], a formal notation that has been specifically developed to support partial descriptions of coalitions. We described how the abstract architecture can be manually translated into a constraint satisfaction problem that can be processed by the Alloy Analyzer, one of the constraint solvers available for Alloy. As a result, we were able to ran an exhaustive search for any concrete architecture that

comply with the abstract architecture within a predefined analysis setting.

In this article, we aim to address two limitations of our previous work. First, our previous work requires an in-depth understanding about Alloy in order to tailor a constraint satisfaction problem for each SoS. Second, our previous work presents the solution returned by the Alloy Analyzer only as an XML model or as a box-and-lines diagram, hence introducing an intermediary model to the SoS architecture description. In this work, we establish Ark, a constraint-based method that supports the transformation of the key concepts of a SosADL abstract architecture into a constraint satisfaction problem expressed in Alloy. In addition, we discuss how this method can be incorporated in a model-driven approach to further bridge the gap between the description of abstract and concrete architectures. Therefore, this work makes the following contributions:

- Formalization of abstract architectures for SoS expressed in SosADL in terms of Alloy;
- Definition of a model-driven approach for the translation of the abstract architecture expressed in SosADL as a constraint satisfaction problem and of the solutions obtained by the constraint solver as a concrete architecture expressed in SosADL;
- Extension of an integrated environment for SoS design and development with support for automated synthesis of concrete architectures that comply with an abstract architecture description.

1.1 Motivating Example

The city of São Carlos, located in the southeast of Brazil, has been consolidated in recent years as a technological hub, centralizing universities and jobs on technology in the state of São Paulo. The Urban River Monitoring (URM) is one of such systems that have been placed on the Monjolinho River in São Carlos, Brazil [26] to support local enforcement and rescue teams during a flooding event, in which rivers that cross inhabited regions overflow, endangering people and businesses in the area. A joint effort among researchers on embedded systems and systems engineering is putting in place a distributed, yet reliable, Wireless Sensor Network (WSN) that is expected to collect timely observations of the river, such as depth and average current speed [26]. These networks have been widely used in river monitoring and warning systems as they support distributed data collection thanks to assorted communication capabilities, such as WiFi, ZigBee (IEEE 802.15.4), GPRS, or Bluetooth, embedded in sensor motes [25].

The URM architecture should be designed to deal with changes in its constituents over time. Thereby, this architecture is dynamic in that new sensor motes may be added or removed from the system, e.g., to save battery power of connected sensor nodes. Moreover, its architecture is expected to evolve over time, e.g., to take advantage of new types of systems that can be incorporated as they become available to the organization running the SoS or to replace or disconnect systems aiming to maintain and/or increase the overall system performance. Due to the criticality of the URM mission, verifying the correctness and completeness of the abstract architecture constitutes an important step for engineering this SoS. In particular, automated support is needed to guarantee that changes to the abstract architecture can still meet the original intent of the SoS architect. In this scenario, architects of such system can benefit from having an abstract architecture to reason about required interconnections between systems, specially since the way in which systems are interconnected impacts SoS-wide behaviors that can emerge [58].

The remainder of this paper is organized as follows. First, Section 2 presents essential concepts on software architectures and SoS for the development of this work. Section 3 explains the rationale for the definition of Ark, a constraint-based method that formalizes SosADL abstract architectures in terms of a constraint satisfaction problem. Afterwards, Section 4 discusses how Ark can be supported by a model-driven approach for the synthesis of SoS architectures. Then, using the dynamic architecture of the URM as an illustrative case, Section 5 demonstrates how this method allows one to generate concrete coalitions that are correct-by-construction. Section 6 positions our method in regards to other works on the synthesis of dynamic architectures. Finally, Section 7 elaborates on future directions for advancing the state-of-the-art on architectural synthesis of smart systems.

2 Background

This section presents the main topics regarding software architectures of SoS that are addressed in this paper. In particular, we contextualize our research on model-based approaches and ADLs for representing run-time coalitions.

2.1 Definitions of SoS and Software Architectures

There are five characteristics that distinguish SoS from traditionally complex, monolithic systems [7]: (i) coalitions produce an emergent behavior that cannot be pro-

vided by any constituent alone; (ii) constituents retain their operational independence since they can still operate even when detached from the SoS; (iii) constituents also retain their managerial independence since they can be developed, maintained, and evolved independently from the SoS; (iv) coalitions are evolutionarily developed by continuously changing in response to different environments and needs; and (v) constituents can be geographically distributed as they can only exchange information with each other.

The ISO/IEC/IEEE 42010 [27] defines a software architecture as the "fundamental organization of a system, embodied in its components, their relationships to each other and to the environment, and the principles governing its design and evolution over time." Software architectures have been associated with diverse quality attributes, e.g., safety [10], maintainability [3], and longevity [1]. Aiming to establish a disciplined approach for the design, evaluation, and evolution of software architectures, several processes have been defined over the years [24]. Following, we summarize the main steps of a generic model [24] for architecting software systems.

The architectural analysis uses the context (i.e., the environment) and architectural concerns to formulate the set of Architecturally Significant Requirements (ASR) in terms of desired systems properties that must be fulfilled by an architecture. The architectural synthesis builds upon ASR to outline potential solutions satisfying these requirements. Many decisions are taken as part of the design of software architectures in regards to which patterns, styles, reference architectures, and platforms are more suitable for a given software system that ultimately determine the final shape of software architectures [32]. Therefore, a description of the software architecture should ease the communication of its design to stakeholders as well as support its analysis [44]. Finally, the architectural evaluation checks candidate solutions against ASRs until selecting one architecture that is more suitable for the system, which is not trivial given the diversity of competing requirements, quality attributes, and concerns.

The evolution of software architectures is a natural step in long lived software systems as it is intended to keep the architectural design aligned with systems goals and technologies [52]. Without support for evolution, architectures become obsolete, decreasing internal and external system qualities [1]. Despite its importance, the evolution is often a secondary concern in most studies on software architectures [8].

2.2 Challenges for Architecting SoS

The software architecture of an SoS encompasses the structure of constituent systems, the relationships that exist among them, and the principles and guidelines governing its design and evolution over time [17]. As a baseline for developing constituents and a shared infrastructure for distributing work, the architecture of a SoS is a key artifact during SoS engineering processes [15]. The dynamic architecture required by SoS represents a shift from architectures of traditionally large and complex software systems that can be determined early and remain relatively stable throughout their entire life cycle [61]. In this scenario, current practices lack the means to cope with the unpredictable ways in that SoSs architecture can evolve and grow at run-time [67].

The lack of support for evolution is particularly harmful in the context of SoS since their software architecture is constantly changing as a consequence of their evolutionary nature. Therefore, even though a consolidated discipline has already been established for supporting software architectures, architecting SoS brings additional, major challenges [6] that remain open due to the conjunction of their inherent characteristics [51]. For instance, a process for engineering SoS must support incremental development and evolution between iterations, continuous analysis against changing contexts and requirements, and continuous input from the external environment [16].

To achieve interoperability among constituent systems, SoS require efficient means to mediate the collaboration among heterogeneous constituents as well as tailor these interactions according to specific operational environments and needs [4, 6, 43]. For instance, legacy constituent systems may resist to make the required changes for their interaction with other constituents of the SoS [52]. In this case, SoS architects may decide to define a gateway or encapsulate the constituent system so that required changes to the actual constituent are minimized. Moreover, specific architecture styles can be used to foster connectivity among constituent systems, such as net-centric architectures, layered architectures, and agent-based architectures [14, 38].

2.3 Architectural Description of SoS

As tangible artifacts expressing software architectures, architecture descriptions provide concrete ways for assessing systems qualities, sharing architectural knowledge, and preventing software systems decay [12, 27,

40]. To disseminate best practices regarding the creation of such artifacts, the ISO/IEC/IEEE 42010 [27] establishes the main elements framed in architecture descriptions and the relationships that exist among them. Moreover, architecture descriptions can be customized for a particular domain or stakeholder community. In the context of SoS, architecture descriptions must detail functionality that is performed by key constituent systems, data and control flow, externally visible properties and interfaces of constituents (e.g., behaviors, dependencies, and use of shared resources), relationships among organizational entities and constituents as well as rationale and governance policies that apply to the SoS, including guidelines for acquisition, termination, and replacement of constituents [17].

The broader definition of an ADL as any technique employed for expressing software architectures [27] allows one to categorize them among formal, semi-formal, and informal languages [3]. Formal languages are distinguished by a precise syntax and semantics, supporting simulation and verification of architectural models [70]. Often, the precision and rigor that can be achieved with formal models is essential for obtaining higher reliability in software systems [29, 45]. Such languages have become more accessible with the development of tailored tools that hide their complexity from users [33]. Conversely, informal languages, such as box-and-lines notations, have been widely employed for describing software architectures despite their lack of precise syntax and/or semantics. In this sense, informal notations offer limited support for analysis and reuse. Semi-formal languages can be seen as a compromise between formal and informal notations, presenting a well-defined syntax but lacking a complete semantics. Semi-formal languages, such as UML (Unified Modeling Language) [54] and its extensions (e.g., SysML [55]), have been widely used in industry [12] and also for the description of SoS [19].

The architecture description of SoS must deal with the intrinsic characteristics of such systems, in particular the independence of constituent systems, evolutionary development, and emergent behavior [51]. Nonetheless, traditional ADLs such as UML lack the means to abstractly describe coalitions so that they can be dynamically created based on existing constituents and targeted emergent behaviors [18]. In this scenario, a novel ADL, named SosADL [56], has been developed to overcome most of the limitations found in traditional ADLs that limit their application for the description of SoS architectures in formal terms. This language is grounded on a novel process calculus of the family of the π-Calculus for formally describing the architecture of Software-intensive SoSs, named π-Calculus for SoS [57].

The core concepts modeled with SosADL are the ones of [56]: (i) *system*, to represent potential constituents; (ii) *mediator*, to represent potential connectors among constituents; and (iii) *SoS*, to represent a potential coalition. More precisely, constituent systems are architectural elements defined by intention (declaratively in terms of abstract system types) and selected at run-time (concretized). Mediators are architectural elements defined by intention (declaratively in terms of abstract mediator types) and created at run-time (concretized by the SoS) to mediate the interaction between constituent systems and create an emergent behavior. Systems-of-systems are architectural elements defined by intention (declaratively in terms of abstract architecture types and the policies/constraints to select and bind concrete constituents within a coalition using the mediators created by the SoS itself) and evolutionarily created at run-time (concretized) to achieve the SoS mission in an operational environment.

In this scenario, tool support is needed to investigate a broader spectrum of run-time scenarios that could be uncertain or unknown at design time. This situation characterizes a satisfiability problem (SAT) in that we attempt to determine if there is any concrete architecture that complies with a SosADL description. As a NP-complete problem [13], all known algorithms for SAT face state space explosion issue as the number of variables exponentially increases in the worst-case scenario.

3 The Ark Method for Architectural Synthesis

Architecture descriptions of the SoS can be used to support the synthesis of architectural configurations that meet the requirements of a SoS. In particular, our approach seeks the use of architecture descriptions for representing SoS from two abstraction levels: (i) an *abstract architecture*, which provides a normative model for the SoS that can be formed from abstract constituent and mediators types; and (ii) a *concrete architecture*, which provides a descriptive model of a coalition that exists at run-time. Following, we further elaborate on the characteristics of each abstraction level.

Abstract architectures describe constituent systems and communication links supported by the SoS. Nielsen et al. [51] refer to an abstract architecture as a conceptual description of an envisaged SoS. This description comprises abstract types of systems, which are characterized in terms of properties, interfaces, and constraints that must hold in the environment. Because

actual constituent systems of the SoS are not necessarily known at design time, the abstract architecture focuses on the identification of key constituents that support the SoS mission. Abstract types of mediators are also specified at this abstraction level, characterized as communication links within the SoS and also with the environment. Finally, the abstract architecture also defines abstract types of coalitions in terms of policies that govern how constituents may interact within the SoS. Therefore, the coalition specification dictates which sorts of configurations can emerge at run-time, such as a layered or client-server architecture.

In turn, concrete architectures explicitly define the architecture configuration of a SoS in terms of known constituents and communication links between them. This can be accomplished at run-time by enabling to deploy mediators on-the-fly based on specific needs of the SoS or current constituent systems of the coalition. Therefore, the concrete architecture is also referred to as a run-time software architecture of the SoS. Alternatively, Kenley et al. [34] refer to concrete architectures as allocated architectures or candidate solutions that potentially show varying levels of quality. In this sense, multiple candidate solutions can be realized for the SoS but they might show different emergent behaviors depending on which types of mediator are in place.

3.1 Steps for Synthesizing Concrete Architectures

To analyze and/or simulate the emergent behavior of a SoS, one must first instantiate the abstract architecture into a concrete architecture that represents a potential coalition. This activity should be repeated at run-time whenever there is a change to the abstract architecture, e.g., a change in the policies (also referred to as bindings/interconnections) that govern the composition between architectural elements of the SoS. Since these modifications cannot be predicted at design time, tool support must be provided for helping SoS architects in confirming the impact of these changes against their original intent. Hereinafter, we present the method Ark supporting the synthesis of concrete architectures that are correct-by-construction. This method is based on two formal notations for expressing architectural models of the SoS: SosADL, an ADL tailored for the description of SoS architectures, and Alloy, a formal method for systems specification.

Figure 1 summarizes the four steps of the method: (i) describe the abstract architecture of the SoS which is expressed in the SosADL notation; (ii) describe the abstract architecture of the SoS in terms of a CSP which is expressed in the Alloy notation; (iii) solve the CSP using off-the-shelf constraint solvers available for Alloy; and (iv) evaluate the outcome of the constraint solver in regards to the original goals of the SoS abstract architecture. Therefore, the method bridges the gap between abstract and concrete architectures by means of the translation of the abstract architecture in terms of a Constraint Satisfaction Problem (CSP), which can be automatically processed by a constraint solver for realizing concrete architectures that adhere to such an abstract description. As a result, Ark can be repeated whenever there is a change to the abstract architecture so that one can validate if desired properties are still preserved by a potential coalition. Otherwise, Ark can detect when an abstract architecture evolves into a state from which no desirable coalition exists. Following, we describe the conceptual model for SoS architectures implemented in Alloy.

3.2 A Conceptual Model for SoS Architectures in Alloy

To automatically resolve the problem of finding a concrete architecture that satisfies the abstract architecture, we formalized the abstract architecture of the SoS in terms of a Boolean Satisfiability Problem (SAT) and adopted Alloy as the underlying constraint solver. In particular, this language is used for expressing policies as *facts*, i.e., constraints that always hold. The Alloy Analyzer supports the generation of instances of model invariants, the simulation of operations defined as part of a model, and the verification of user-specified properties of a model. In addition, the Alloy Analyzer supports the analysis of partial models and thereby can perform incremental analysis of models as they are constructed.

Aiming to facilitate the representation of abstract architectures of SoS in terms of constraints, we created TASoS[1] as an intermediary representation for SosADL models in Alloy. This representation, which is illustrated in Fig. 2, captures the basic building blocks of the SosADL notation in terms of signatures, i.e., indivisible atoms that behave as sets. Overall, TASoS contains 20 signatures, 14 relations, and eight assertions that can be used for checking optional properties about the abstract architecture. TASoS also defines predicates and functions which describe operations over these signatures. When executed in the Alloy Analyzer, a predicate will instruct the solver to produce an instance, i.e., a concrete architecture, that satisfies the constraints framed by the predicate. In turn, a function defines an expression whose evaluation yields a subset of atoms and, thus, may be called in predicates, assertions, and facts to improve modularity and readability

[1] The source code for TASoS is available at `http://goo.gl/5ZNgDQ`

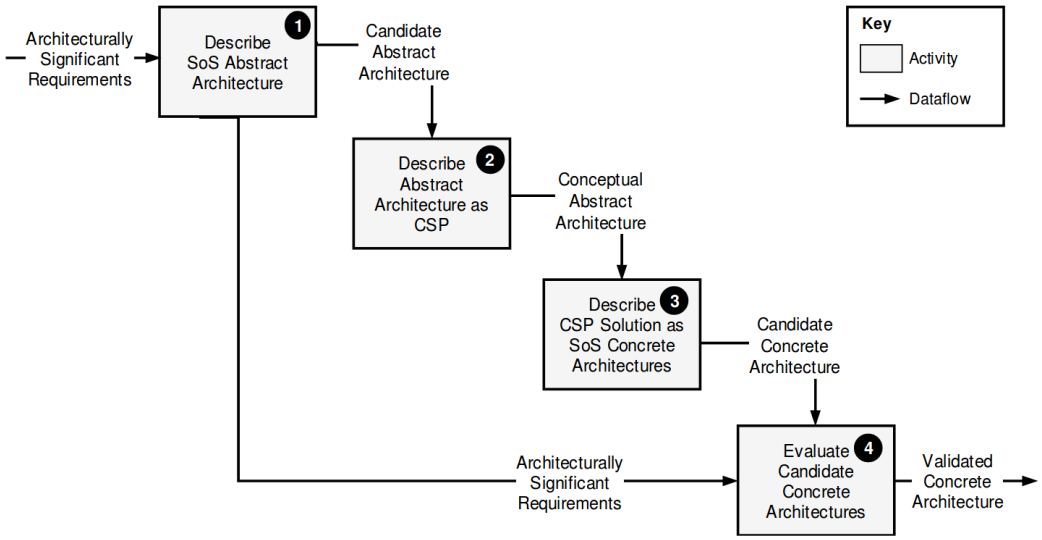


Fig. 1: General model of Ark for SoS architectural synthesis [21]

of the model. TASoS is defined as a module that can be referenced by other Alloy models, thus facilitating its reuse.

Inspired by the SosADL language, TASoS provides individual signatures for each abstract architecture element in SosADL, namelly *System*, *Mediator*, and *Sos*. These three signatures derive from the same parent signature that is named *ArchitecturalElement*. Therefore, when we define the *hasPort* relation between their parent signature and the *Port* signature, we are actually defining a relation that applies to all three child signatures. The reasons for the selection of SosADL as the basis for the creation of this metamodel are threefold: (i) the notation has explicit constructs for abstract types of constituents, mediators, and SoS, including a mathematical foundation [57] that enables to define dynamic connectivity in terms of constraints; (ii) the notation supports the description of SoS abstract architectures from a dynamic as well as static viewpoint; and (iii) the notation is supported by a tool that can be extended to work with different modeling and implementation languages, such as UML, Java, and C. Following, the central elements of TASoS are detailed.

The main signature of TASoS is *Architecture*, which captures the abstract architecture of a SoS in terms of potential elements (i.e., systems and mediators), abstract interconnections between these elements, and their configuration, which expresses a concrete architecture of the SoS (i.e., *Topology*). Thereby, this signature is defined in terms of three relations, namely: (i) *contain*, a binary relation of type $Architecture \rightarrow (System + Mediator)$ relating each architecture to elements of system or mediator types that can participate in the coalition; (ii) *bindings*, a binary relation of type $Architecture \rightarrow Unification$ mapping the interconnections between systems and mediators that have been defined by the abstract architecture; and (iii) *unifiedAs*, a binary relation of type $Architecture \rightarrow Topology$ describing different ways (i.e., candidate concrete architectures) in which systems and mediators can be arranged together to form a coalition.

The metamodel defines several facts, i.e., constraints, about the SoS architecture that must hold in any given concrete architecture, instructing the solver about desirable and undesirable configurations. Listing 1 shows an excerpt of the facts in which *Architecture* plays a role. For instance, constraint #28 instructs the solver to consider any two architectures to be equal if the set of elements assigned to their topologies, which is returned by calling the `elems` function over the `unifiedAs` relation, is the same for both architectures. Thereby, one can further instruct the solver to only search for unique architectures. Constraint #31 instructs the solver to only find topologies in which all systems engage in at least one interconnection with another element of the architecture, hence discarding disconnected concrete architectures as valid solutions. To guarantee that this is the case, TASoS forces the set of all unifications in which a system is either the recipient or the source of an unification (which is returned by the `participatesInTopology` function) to not be null. Constraint #32 instructs the solver to only take into account topologies that use all types of abstract interconnections. In particular, the `isUnifiedTo` function is referenced in this rule to obtain the elements that are linked by an unification in the topology. Finally, constraint #34 ensures that for any given topology of an architecture, each interface of a system (referred to as *Gate*) can

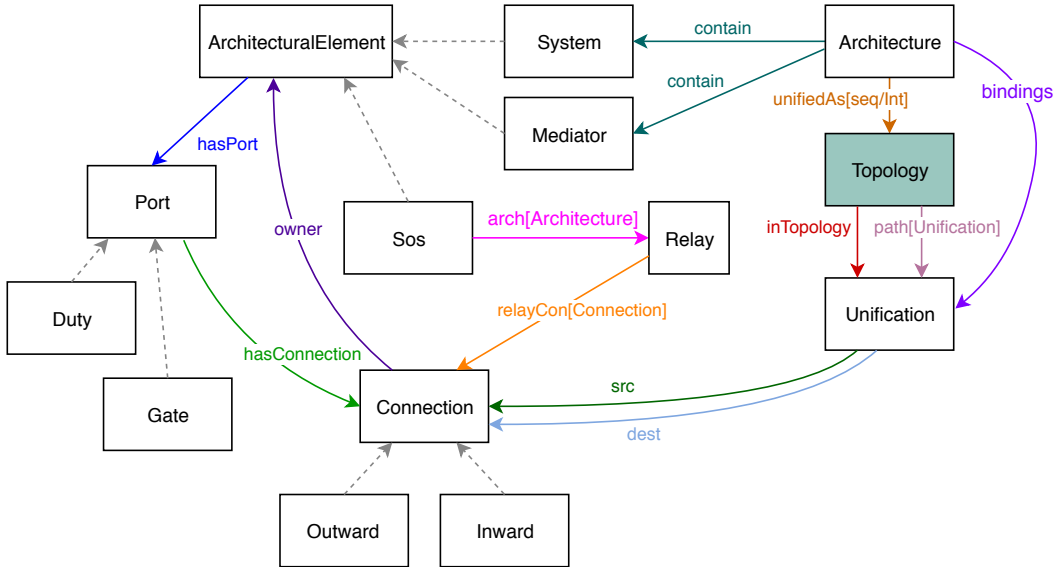


Fig. 2: TASoS, an Alloy metamodel for SosADL. Dashed arrows represent an extend relation. While based on SosADL, TASoS introduces a new architectural element (colored) for the definition of concrete configurations [21]

be related to exactly one interface of a mediator (referred to as *Duty*), which is returned by calling the `dutiesOfUnification` function over *Suni*, i.e., the set of unifications of a given port returned by the `unificationsOfPort` function.

The *Topology* is the signature in TASoS supporting the description of concrete architectures. Atoms of this signature have two relations, namely: (i) *inTopology*, a relation in the form $Topology \rightarrow Unification$ which contains the set of intentional bindings that can be realized between abstract types of systems and mediators; and (ii) *path*, a ternary relation of type $Topology \rightarrow Unification \rightarrow Unification$ which connects atoms of unification in order to compose a network of systems and mediators. Thereby, this signature is responsible for merging the description of an abstract architecture with a concrete architecture in that its first relation lists abstract types of bindings that can be established at run-time (which is taken from the abstract description) and its second relation forms a network that can be created on top of these bindings (which results in the concrete description). To complete the specification of well-formed concrete architectures for an SoS, TASoS defines the policies (i.e., constraints) governing the topology in terms of facts. Listing 2 shows an excerpt of constraints that apply to topology. For instance, constraint #29 instructs the solver to allow a topology to be empty if, and only if, there is no system in the coalition. Constraint #30 instructs the solver to look for topologies in which all defined types of mediators are present. This is achieved by checking if the mediator is the recipient or the source of any unification in the topology, which is returned by the `owner` function. Finally, constraint #33 instructs the solver to only accept connected topologies by restricting which unifications can be associated by the *path* relation. Specifically, two unifications can only be associated if they share the same origin and/or destination with each other. To do so, it calls the `isUnifiedTo` function over each unification to verify which architectural elements they bind together.

As an intermediary model for SosADL abstract architectures expressed in terms of constraints, TASoS can be further refined (i.e., instantiated) for representing the abstract architecture of a particular SoS. The execution of TASoS in the constraint solver is expected to produce one or more concrete architectures, i.e., solutions, that satisfy these constraints. Therefore, if no solution is found, we can state that no coalition satisfying the abstract architecture exists under the specified execution bound, i.e., scope. In this case, the architect can either investigate if the scope is sufficiently large to hold at least one solution of this problem or if the abstract architecture is not suitable (i.e., if it is over constrained). This metamodel can support both investigations by means of *assertions*, i.e., optional constraints that can be verified. An assertion can check, for instance, if non-empty instances or multiple mediators are allowed for a coalition. If a given assertion does not hold in the SoS, a counterexample depicting a concrete architecture that violates the abstract architecture is generated. This outcome can be useful for correcting or refining the abstract architecture. Conversely, if no counterexample is found, the architect can infer that

Listing 1: Excerpt of constraints on architectures in TASoS

```
//#28 Two architectures should be equal if they have the same set of topologies
all a,a': Architecture|
 (a.unifiedAs).elems=(a'.unifiedAs).elems implies a=a'
//#31 All systems that participate in an architecture must engage in at least one unification in all candidate topologies
all e:System,a:Architecture,t:Topology | e in a.contain
 and t in (a.unifiedAs).elems implies
  some participatesInTopology[e,t]
//#32 All unifications in the set of bindings must be used in the topology.
all u:Unification,a:Architecture,e:System | u in a.bindings and
 e in a.contain and e in isUnifiedTo[u] implies
  all t:Topology | t in (a.unifiedAs).elems implies
   u in path[t].Unification
//#34 Connections of the same gate can only be unified to connections of the same duty in a given Architecture.
all g: Gate, a: Architecture, t: Topology |
 let Suni=unificationsOfPort[g,t] |
  g.~hasPort in (a.contain) and t in (a.unifiedAs).elems
   implies one dutiesOfUnification[Suni]
```

the assertion holds for that particular scope and increase the analysis scope aiming to gain more confidence about the architectural design. Thereby, checking these assertions can provide concrete evidences about the feasibility and soundness of the SoS.

4 Tool Support for Ark

To perform the steps recommended by Ark, architects create at least three models of the SoS architecture. First, architects manually create an abstract architecture for the SoS under analysis in SosADL. Then, architects represent this architecture as a set of constraints, creating an instance module of TASoS. If this resulting representation is syntactically correct, architects can analyze this model in external constraint solvers. Finally, architects have to translate the solution returned by the constraint solver (i.e., concrete architectures) back into a format that they can more easily analyze and communicate with stakeholders. In this scenario,

Listing 2: Excerpt of constraints on topologies in TASoS

```
//#29 The relations inTopology and path must be empty if no system participates in the coalition
all t:Topology, a:Architecture | t in (a.unifiedAs).elems implies {{no t.path} iff {no a.contain & System}}
//#30 For all topologies of an architecture, a mediator engages in at least one unification
all t: Topology, m: Mediator |
 let Suni=participatesInTopology[m,t] |
  some Suni implies
   m in owner[Suni.src] or m in owner[Suni.dest]
//#33 The path of a topology connects unifications that originate or end in the same architectural element
all t:Topology, a:Architecture | {some a.contain} and
 t in (a.unifiedAs).elems implies {
  all u,v: Unification | u->v in t.path implies
   some isUnifiedTo[u]&isUnifiedTo[v]
 }
```

these activities can be time consuming and prone to error.

An automated process for applying Ark encompasses two model transformations: first, an abstract architecture must be translated into a CSP that extends TASoS; secondly, solutions for this problem must be translated back into SosADL as concrete architectures. Hence, we devised a tailored software tool named SoSy that uses constraint solving techniques under the hood. More precisely, SoSy implements two model transformations (illustrated in Figure 3): (i) a Model-to-Text (M2T) transformation from SosADL to TASoS, in which instance modules can be dynamically created for abstract architectures; and (ii) a Text-to-Model (T2M) transformation from the solution returned by the constraint solver to SosADL, in which concrete instances found by the tool are represented in SosADL. The first transformation is implemented as an Xtend[2] class named `SosADL2AlloyGenerator`, whereas the second is implemented as a Java method named `Solution2SosADL-Generator`. Moreover, this tool is provided as an extension to SosADE[3] (Architectural Framework for SoS Design) [59], an integrated environment for the design, validation, and simulation of SoS that supports SosADL as the modeling notation.

Figure 4 illustrates the model-driven process implemented by SoSy, which is inspired on the four steps

[2]Xtend, `http://www.eclipse.org/xtend/`

[3]SosADE tool, `https://www-archware.irisa.fr/software/`

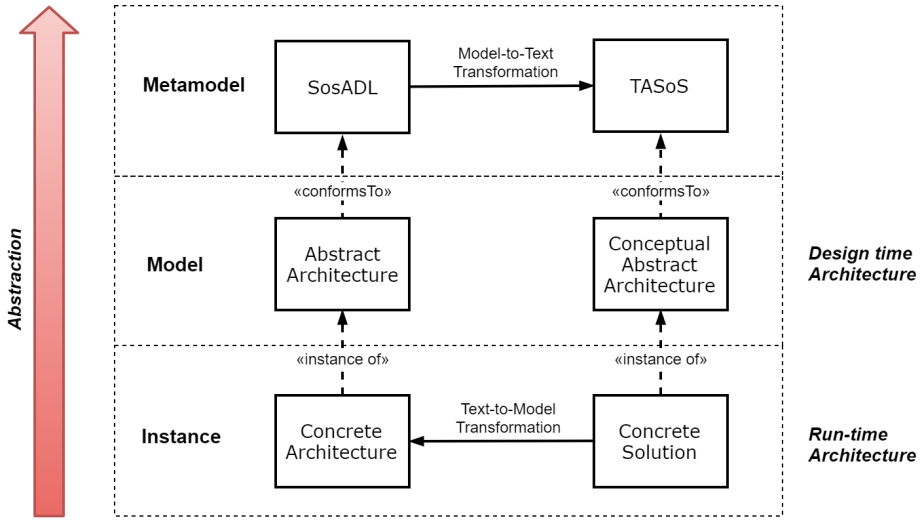


Fig. 3: Relationship between abstract and concrete architecture models in SosADL and TASoS [21]

required by Ark. This process is represented in SPEM [53], an OMG standard for the specification of software methods and processes. In particular, this figure describes the sequence in which *tasks* (i.e., work that has a particular purpose) and *activities* (i.e., group of related tasks) must be carried out by architects. The first activity in this process concerns the creation of an abstract architecture for the SoS, which automatically triggers the task for generating an instance of TASoS. In parallel to this task, the tool also generates a Java[4] class that when executed will call the Alloy Analyzer from within the SosADL development environment for processing the instance module. In addition, this class temporarily stores data (namely, datatype functions) that are not processed by the constraint solver but that are needed for the reconstruction of the concrete architecture. These tasks are followed by the manual execution of the Java class. As a result, if any solution is returned by the solver, it will be automatically translated into a concrete architecture expressed in SosADL. Otherwise, architects can decide to manually repeat this activity with a different analysis scope or repeat the process for a new abstract architecture, triggering the update of the related artifacts in the project directory. The process ends when the evaluation of a candidate concrete architecture terminates with a valid concrete architecture of this SoS.

5 Demonstration of Ark

The analysis performed by the Alloy Analyzer is based on checking the validity of a property (i.e., a constraint). To guarantee efficiency, the analysis (i.e., model checking and model-finding) is completed under a execution scope that constrains the investigated solution space, thereby finite. The scope for the analysis must be carefully selected according to each problem at hand. Even a small scope yields a sizable solution space (in the order of 10^8 clauses) which is sufficiently large for discovering problems in small instances of models [28]. In fact, the analysis performed by the tool is grounded on the premise that even small instances of a model can illustrate flaws, which arise from incorrectly handling types [30]. Furthermore, the power of this analysis is greater than one that could be achieved by specifying the problem in Java, which would lack support for generating arbitrary samples, performing exhaustive checking on test cases, and visualizing the results.

Following, we describe the URM system (Section 5.1), which has been selected as an illustrative example to demonstrate the Ark method. Then, Sections 5.2 through 5.5 explain step-by-step how one can use this method for checking the feasibility of an abstract architecture and producing a correct concrete architecture for the URM. In Section 5.6, we report the performance of this method after running the constraint solver over several analyses scope and in Section 5.7 we discuss threats to the validity of this quasi-experiment. Finally, in Section 5.8, we discuss advantages, limitations, and lessons learned of our method and tool.

5.1 Urban River Monitoring System-of-System

Emergency management and response is a relevant application domain for complex SoS [51]. The design of such SoS often requires the coordination among hetero-

[4] Java, `http://www.java.com`

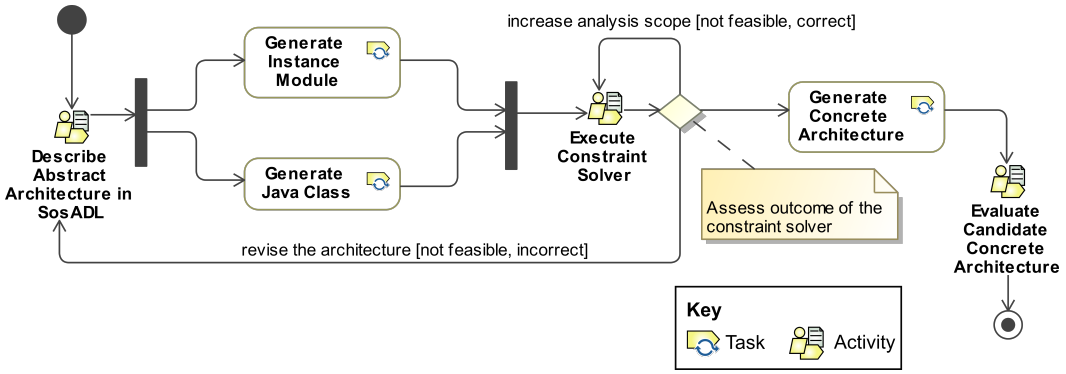


Fig. 4: Workflow of a model-driven process for the synthesis of concrete architectures in SosADE [21]

geneous constituent systems (e.g., surveillance, weather forecast, and river monitoring systems) and protocols (e.g., traffic management, first aid, and rescue teams). An Urban River Monitoring SoS plays a key role in obtaining precise, real-time data that supports authorities' timely and organized response, specially in case of flash floods. To achieve this goal, data provided by independent, heterogeneous sensors (e.g., water level, current, and pollutant sensors) are combined by the SoS to monitor the flood risk. Ultimately, the effectiveness of such SoS has also a relevant impact on costs incurred by flood events, which can be greatly diminished by sending out warnings in advance [62].

To detect an imminent flood risk, data collected from stationary sensors are forwarded to a gateway station, which has dedicated resources for processing, integrating, and publishing this information. Specifically, the gateway station can transform raw data collected by sensors for determining the relative height reached by the water level and publish this information to the authorities. If a gateway station cannot be reached by the individual communication capabilities of a sensor, the sensor will forward these data to its neighboring motes until the gateway station is reached. Since this system operates in a highly dynamic environment, its architectural configuration must be continuously changed for ensuring: (i) efficiency in the use of the available resources, mainly in terms of power consumption and communication; (ii) resilience in case of temporary unavailability of motes during operation; (iii) accuracy in flood detection; and (iv) autonomy in adapting to dynamic environmental conditions while minimizing manual intervention.

When compared against the five characteristics presented by SoS [43], the Urban River Monitoring meets all five criteria. Each sensor mote operates in a way that is independent of other sensor motes, since they belong to different city councils and could have different missions, e.g. pollution control or water supply. Each one has its own management strategy for transmission vs. energy consumption and acts under the authority of the different city councils. New sensor motes may be installed by the different councils as well as existing ones may be changed or deactivated without any control from the system. Finally, the sensor motes, coordinated by the gateway, make emerge the behavior of flood detection. This behavior is collectively achieved by the distributed, independent sensors working together with the gateway station rather than being provided by any of them working in isolation. Thus, as a collaborative SoS that has no central authority coordinating the constituents operation to achieve the SoS goal [43], the URM must be able to dynamically assemble new coalitions from the sensor motes that voluntarily decide to forward data to the gateway station.

5.2 Step 1: Describe the URM Abstract Architecture

The first step of the method concerns the description of an abstract architecture for the SoS under analysis. To perform this task, the architect uses the SosADE tool to create SosADL models for the URM[5]. In particular, the architect should begin by the definition of abstract types for the architectural elements that form the URM SoS, i.e., sensor, mediator, and gateway. However, the abstract architecture does not identify the actual elements that participate in the coalition since these are not necessarily known at design time. These abstract types are defined within a SosADL library, named `urm-Library`, which comprises two abstract types of systems, namely sensor and gateway, and an abstract type of mediator, named transmitter, which represents the interconnections supporting their collaboration. Following, the description of these abstract types is further elaborated.

[5]The complete source code for the URM models referenced in Section 5 are available at `http://goo.gl/5ZNgDQ`.

A *gateway* is an abstract type of system that requests observations from sensors and publishes collected data. The declaration of this system, shown in Listing 3, encompasses one gate named *alerting* that has three connections: (i) *request* receives queries from other systems about current condition of the river; (ii) *measure* handles observations returned by other systems; and (iii) *alert* sends out warning messages when an internal parameter of the gateway is violated by environmental conditions. All connections have an associated direction (inward or outward) and data type, which is limited to an abstract integer type named t for the sake of simplicity. For the behavior declaration, the architect specifies a sequence in which the system is expected to interact with the environment. The main behavior of the gateway is depicted in lines 18-24. It begins by receiving an observation from another systems via its *measure* connection. Then, the gateway evaluates if this observation surpasses the local depth threshold (line 22). If so, it publishes a warning message via its *alert* connection of the same gate. These actions can be repeated indefinitely.

Listing 3: Excerpt of abstract gateway type for the URM in SosADL

```
3   system gateway() is {
4    gate alerting is {
5     connection measure is in {t}
6     connection request is in {t}
7     connection alert is out {t}
8    } guarantee {
9     protocol alertingpact is {
10     repeat {
11      via request receive any
12       repeat {
13        via measure receive any
14        repeat { anyaction }
15       }
16       via alert send any
17   } } }
18   behavior main is {
19    value depththreshold : t = 3
20     repeat {
21      via alerting::measure receive v
22      if (v > depththreshold) then {
23       via alerting::alert send v
24   } } } }
```

A *sensor* is also an abstract type of system that collects observations from the environment. The declaration of this system, shown in Listing 4, encompasses two gates: (i) *measuring*, that comprises one environment connection named *sense* that reads observations and one connection named *measure* that handles these observations over to a neighboring system; and (ii) *passing*, that has two connections *pass* and *measure* for just handling observations over to other systems. These gates declarations are complemented by a guarantee protocol, which describes assumptions (i.e., properties) that must be fulfilled by the environment. For instance, the *measuring* gate guarantees that while the sensor is operational (i.e., if all assumptions that a sensor makes about the environment hold), it will continually receive observations from the environment via its connection *sense* and transmit these data via its connection *measure*. The main behavior of this system is declared in lines 45-53. The sequence of interactions is given by a choice between collecting an observation via its *sense* connection of the *measurement* gate and sending it to one of its neighboring systems via the *measure* connection of the same gate, or receiving an observation via its *pass* connection of the passing gate and forwarding it to one of its neighboring systems via the *measure* connection of the same gate.

Listing 4: Excerpt of abstract sensor type for URM in SosADL

```
25   system sensor() is {
26    gate measuring is {
27     environment connection sense is in {
       t}
28     connection measure is out {t}
29    } guarantee {
30     protocol measuringpact is {
31      repeat {
32       via sense receive observation
33       repeat { anyaction }
34       via measure send observation
35   }}}
36   gate passing is {
37    connection pass is in {t}
38    connection measure is out {t}
39   } guarantee {
40    protocol passingpact is {
41     repeat {
42      via pass receive data
43      via measure send data
44   }}}
45   behavior main is {
46    repeat {
47     choose {
48      via measuring::sense receive
       observation
49      via measuring::measure send
       observation
50     } or {
51      via passing::pass receive data
52      via passing::measure send data
53   }}}}
```

A *transmitter* is an abstract type of mediator that forwards observations from sensors to a gateway, either directly or by means of other sensors. The declaration of this mediator, shown in Listing 5, comprises one duty, named *transmitting*. This duty has two connections named *fromSensors* and *toGateway* that handle the transmission of observations from sensors to a gateway. A duty can also declare assumptions that must

be fulfilled by constituents in the environment. In this case, this duty requires that the distance between the source and target of this communication link do not exceed a predefined range threshold. If this property holds, then the abstract declaration of mediator can guarantee that the protocol declared in *transmittingpact* is also fulfilled. This protocol defines that all observations received via the *fromSensors* connection are forwarded via its *toGateway* connection. Then, transmitting behavior of this mediator states that this action is continuously repeated, with no processing in between.

Listing 5: Excerpt of abstract mediator type for the URM in SosADL

```
54    mediator transmitter() is {
55     duty transmitting is {
56      connection fromSensors is in {t}
57      connection toGateway is out {t}
58     } assume {
59      property inrange is {
60       repeat {anyaction}
61      }
62     } guarantee {
63      protocol transmittingpact is {
64       repeat {
65        via fromSensors receive measure
66        via toGateway send measure
67     }}}
68     behavior transmitting is {
69      repeat {
70       via transmitting::fromSensors
        receive measure
71       via transmitting::toGateway
        send measure
72    }}}
```

Without further directions, a concrete architecture for the URM can potentially be any combination of elements defined in this library. Therefore, the URM description cannot be considered complete without the specification of an abstract coalition type, which identifies a family of concrete architectures that are desirable at run-time. For instance, one can specify an abstract coalition type in which constituent systems must be located within 5 meters of each other and/or that have replied to the SoS within the past 5 minutes in order to attain to the goals of the SoS, such as promote rational use of resources or guarantee overall performance. In either case, the formal specification of such a coalition type supports evaluating the correctness of potential concrete architectures in regards to the original intent of the SoS architect.

Listing 6 shows an excerpt of the abstract architecture named *simple* for the URM. The *serving* gate of this architecture enables the communication of the Flood Monitoring coalition with other systems and the environment. This gate declares two connections named *request* and *alert* that handles external queries about the conditions of the river and publish warning messages in case one of the sensors has identified a flood event. The behavior of an architecture is different from the one of systems and mediators in that it contains: a *compose* declaration, which states constituents that are allowed to participate in the coalition (lines 10-14); and, a *binding* declaration, which defines policies for organizing systems and mediators into a cohesive whole (lines 14-28). The simple type of this coalition is composed of any number of sensors and transmitters but has only one gateway, named gateway1. Therefore, any instance of this abstract coalition type may only exercise the unifications defined in the bindings declaration.

Listing 6: Excerpt of abstract coalition type for the URM in SosADL

```
1  with urmLibrary
2  sos FloodMonitoring is {
3   architecture simple() is {
4    gate serving is {
5     connection request is in {t}
6     connection alert is out {t}
7    } guarantee {
8     ...
9    }
10   behavior main is compose {
11   sensors is sequence{sensor}
12   transmitters is sequence{transmitter}
13   gateway1 is gateway
14   } binding {
15   forall { t in transmitters suchthat
16    forall { s1 in sensors suchthat
17     forall { s2 in sensors suchthat (
18     // t receives from s
19     ( unify one {s1::measuring::measure}
        to
20     one {t::transmitting::fromSensors}
21     or unify one {s1::passing::measure}
       to
22     one {t::transmitting::fromSensors} )
23     and // and sends to s or g
24     ( unify one {t::transmitting::
       toGateway} to
25     one {gateway1::alerting::measure}
26     xor unify one {t::transmitting::
       toGateway} to
27     one {s2::passing::pass} )
28     ) } } }//close forall
29 } } }
```

5.3 Step 2: Describe URM as a Constraint Satisfaction Problem

The second step of Ark requires the transformation of the URM abstract architecture described in SosADL into a constraint satisfaction problem described in Alloy. To accomplish this task, the abstract signatures in TASoS must be extended with the elements defined by the URM abstract architecture. Figure 5 shows how

TASoS is extended to represent the URM abstract architecture: (a) shows that *gateway* (Listing 3) and *sensor* (Listing 4) extend the *System* signature; (b) shows that *transmitter* (Listing 5) extends the *Mediator* signature; and (c) shows that *flood monitoring* and *simple* (Listing 6) extend the *Sos* and *Architecture* signatures, respectively. Listing 7 shows an excerpt of the generated Alloy model for the *simple* coalition abstract type.

Because Alloy is a declarative language, it is important to guarantee that signatures naming are unique. SoSy accomplishes that by composing the names of the elements. Take for instance the *alert* connection of the *serving* gate in the *simple* architecture in Listing 6 (line 6). This connection is mapped in Listing 7 as `simple_serving_alert` (line 7). The *compose* declaration of the abstract architecture in Listing 6 (lines 11-13) is mapped in Listing 7 by the constraints in lines 11-13. Accordingly, the *bindings* declaration is mapped by the constraints in lines 16-31. In SosADE, the transformation of SosADL models into Alloy can be automatically performed by the SoSy tool, which is running in the background. Therefore, instance modules of TASoS are dynamically created by the environment as the architect creates SosADL models.

5.4 Step 3: Solve the Constraint Satisfaction Problem

The third step of Ark comprises the automated analysis of the generated instance model by the constraint solver. In particular, the instance module instructs the solver to find for a concrete architecture that satisfies the scenario described in Listing 8 (lines 38-44), i.e., it is a coalition composed of at least one element of the sensor type, one element of the transmitter type, and one element of the gateway type. To perform this task, the architect can manually execute the generated Java class using the default analysis scope defined for the instance module in line 45.

Listing 8: Scenario instance module

```
38 fact instanceOfsimple{
39     some a: simple {
40         sensors in a.contain
41         transmitters in a.contain
42         gateway1 in a.contain
43     }
44 }
45 run case0 {some sensor} for 3 but 1
       Architecture, 4 System, 4 Mediator, 1
        Sos, 7 Port, 16 Connection, 16
       Unification, 1 Relay
```

5.5 Step 4: Evaluate Candidate Concrete Architectures for the URM

If the solver finds a solution for the previous problem within this analyzed scope, the architect can automatically obtain the corresponding SosADL models, such as the one presented in Listing 9 and illustrated in Figure 6. Afterwards, the architect can proceed with the evaluation of this concrete architecture in regards to one's original intent when designing the abstract coalition type in the first place. Moreover, one can use generated concrete architectures as input for simulations of the SoS. As a result of this activity, the architect may decide to modify the policies defined in the abstract architecture to prevent the formation of undesirable coalitions, to define mechanisms (i.e., mediators) that support new interactions between constituents and/or with the environment, or accept this abstract architecture, hence proceeding to the next stages in the SoS development process.

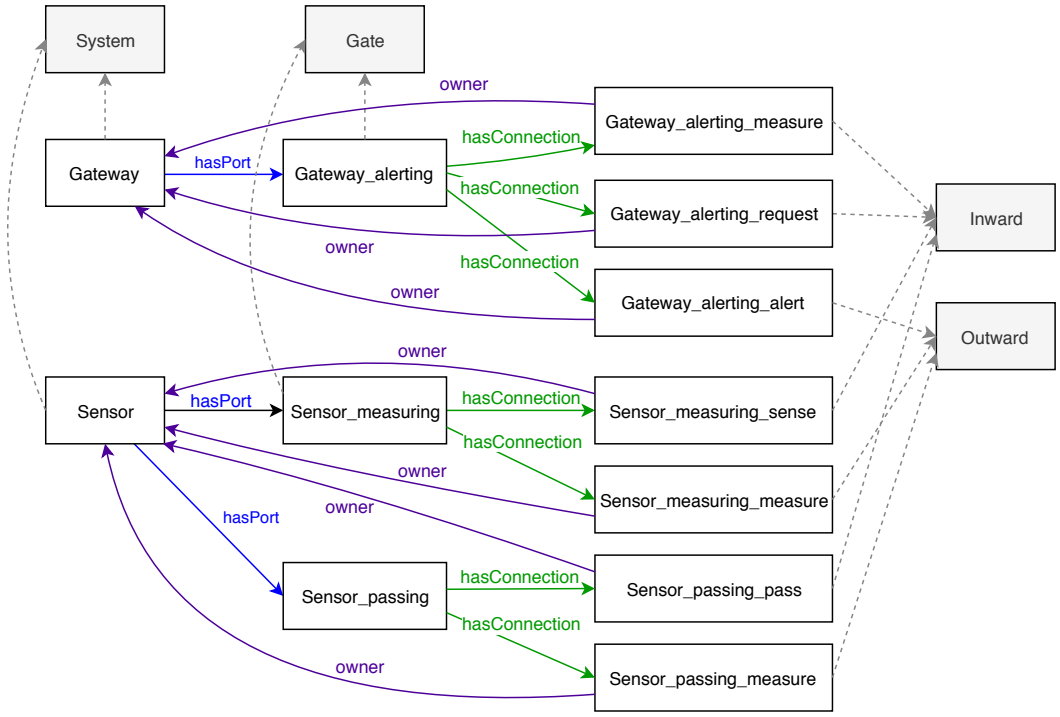
5.6 Performance of the Method

We can evaluate the performance of Ark in terms of elapsed time for the execution of steps 2 and 3, which comprise the use of a constraint solver to find a well-formed concrete architecture for the Urban River Monitoring. Nonetheless, the outcome of this evaluation depends on a careful selection of the analysis scope, which must be sufficiently large to hold at least one concrete architecture of the SoS yet small enough so as not to cause state explosion. Aiming to understand the role played by the selected analysis scope in the performance of Ark, we performed a quasi-experiment [69].

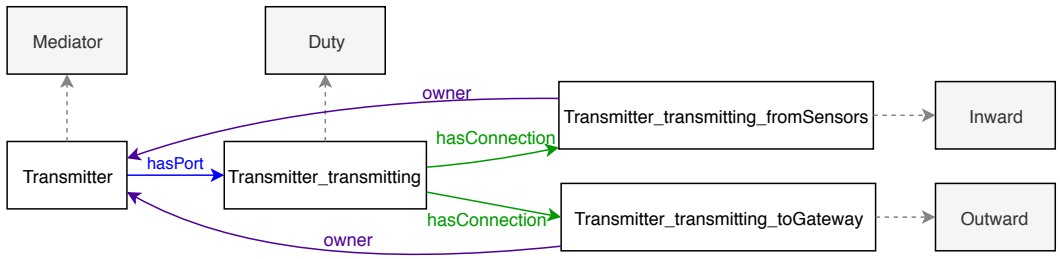
To collect data for this evaluation, we instrumented the generated Java class to repeatedly run the constraint solver with a different scope at each time. In particular, we tested all possible combinations within the interval defined in Table 1, totaling 400 unique test cases. It is important to highlight that the scope assigned to signature *ArchitecturalElement* actually covers all elements of this subtype, including *System*, *Sos*, and *Mediator*.

Table 1: Boundary for analysis scope

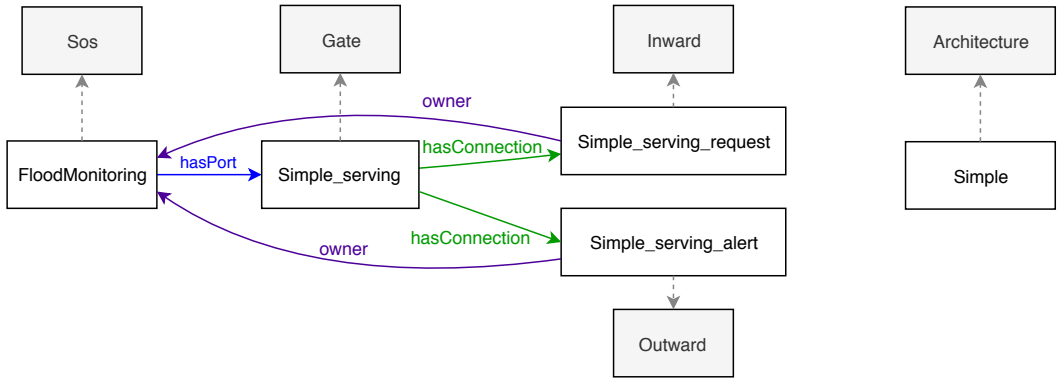
Signature	Min	Max
Architectural Element	3	7
Port	4	8
Connection	8	11
Unification	8	11



(a) Abstract system types



(b) Abstract mediator type



(c) Abstract coalition type

Fig. 5: Model of the URM abstract architecture in Alloy. TASoS elements (colored) are extended with the elements extracted from the SosADL description

Listing 7: Architecture instance module of TASoS for the URM

```
open tasos
open util/ordering[tasos/Architecture] as A0
open urmLibrary
/** Architecture(s) Declaration(s)*/
sig FloodMonitoring extends Sos{}
...
sig simple_serving_alert extends Outward{}
-- constraints about architectural elements
...
-- constraints about elements in coalitions
sig sensors extends sensor{}
sig transmitters extends transmitter{}
one sig gateway1 extends gateway{}
-- definition of abstract unifications in architectures
sig simple extends Architecture{}{
all t: transmitters |
 all s1: sensors |
   all s2: sensors |
    ((
    unify[sensor_measuring_measure&(s1.~owner),transmitter_transmitting_fromSensors&(t.~
    owner)] and //u1
    unify[transmitter_transmitting_toGateway&(t.~owner),gateway_alerting_measure&(
    gateway1.~owner)] //u3
    ) or (
    unify[sensor_measuring_measure&(s1.~owner),transmitter_transmitting_fromSensors&(t.~
    owner)] and //u1
    unify[transmitter_transmitting_toGateway&(t.~owner),sensor_passing_pass&(s2.~owner)]
     //u4
    ) or (
    unify[sensor_passing_measure&(s1.~owner),transmitter_transmitting_fromSensors&(t.~
    owner)] and //u2
    unify[transmitter_transmitting_toGateway&(t.~owner),gateway_alerting_measure&(
    gateway1.~owner)] //u3
    ) or (
    unify[sensor_passing_measure&(s1.~owner),transmitter_transmitting_fromSensors&(t.~
    owner)] and //u2
    unify[transmitter_transmitting_toGateway&(t.~owner),sensor_passing_pass&(s2.~owner)]
     //u4
    ))
}
```

We take into account the following aspects in each test case: (i) analysis scope, given by the quantity of elements assigned to each signature in the execution of the Alloy model for the abstract architecture; (ii) solver running time, given by the running time of the constraint solver; and (iii) satisfiability of the abstract architecture, which can be considered as satisfiable or unsatisfiable if the analysis scope is too small. For the sake of completeness, we also collected the running time of the `Solution2SosADLGenerator` when the solver produced a concrete architecture for the problem.

All test cases were executed sequentially in a machine macOS High Sierra v.10.13.6, with 16GB RAM, and Intel Core i7. All 400 test cases were successful, yielding a solution for the problem under the predefined time limit of five minutes. Table 2 shows descriptive statistics for this experiment. The running time of the constraint solver shows great variation between test cases even though none of them has taken longer than 5 seconds to terminate. Overall, the transformation back from the solver solution into SosADL is efficient, being concluded under 1 second.

Table 2: Descriptive statistics of dependent variables

	Solver	Transformation
Quantity	400	400
Min (ms)	624	14
Max (ms)	4793	49
Mean (ms)	2008	19
Median (ms)	1915	17

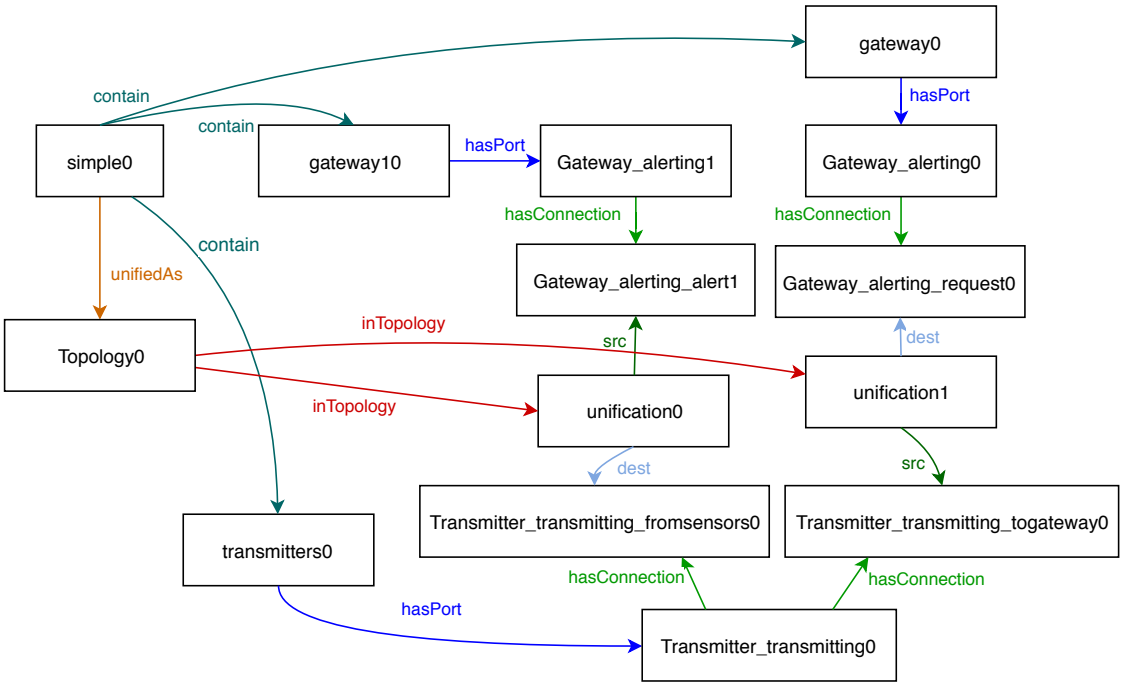


Fig. 6: A concrete architecture of the URM in SosADL

Listing 9: Excerpt of a concrete architecture for the URM that is compliant with the abstract coalition type

```
1  with urm_library
2  sos FloodMonitoring0 is {
3   architecture simple0() is {
4    gate serving0 is {
5     connection request0 is in{RangeType0}
6     connection alert0 is out{RangeType0}
7    } guarantee {
8     ...
9    }
10   behavior main is compose {
11    gateway10 is gateway10
12    gateway0 is gateway0
13    transmitters0 is transmitters0
14   } binding {
15    unify
16     one{transmitters0::transmitting0::
       togateway0} to
17     one{gateway0::alerting0::request0}
18    and
19    unify
20     one{gateway10::alerting1::alert1} to
21     one{transmitters0::transmitting0::
       fromsensors0}
22   }
23 }}
```

5.7 Threats to Validity

Four levels of validity threats were identified for this quasi-experiment, which are discussed in detail.

Internal validity is an inherent risk of this quasi-experiment since treatments are not assigned to subjects and objects by chance. Furthermore, there is only one object and one subject in this experiment due to stage of development of the tool set (SosADE) and lack of subjects with required skills in SosADL and Alloy. Still, this risk is ameliorated by focusing the evaluation on the performance of activities automated by the method, making the role played by the subject not significant in the outcome of this experiment.

External Validity is also a concern for this experiment since its outcome reflects only one object case. This risk is mitigated by selecting as object a system that has the fundamental characteristics of an SoS, as previously discussed. Nonetheless, it is possible to expand this experiment aiming to compare the performance of different constraint solvers and/or coalition types.

Construct validity is related to the threat that measurements are not appropriate for selected entities. To mitigate this threat, we defined several combinations of the analysis scope that do not depend on context.

Conclusion Validity can be also a threat because we have set a minimum and top analysis scope for our experiment, hence not taking into account test cases that would yield longer running times and would not terminate as they cause the state explosion problem. Since our approach purposefully seeks for the smallest solution (i.e., concrete architecture) within the analysis

scope, we conclude that a larger scope would not necessarily bring new information to the SoS analysis. Therefore, we focus our investigation on the lower boundary in our analysis scope provided that we selected a feasible object case for which the solver can produce a solution. We have detailed the protocol and materials used in this experiment aiming to further mitigate this threat.

5.8 Discussion

The translation of abstract architectures into a constraint satisfaction problem is not trivial, specially since this task requires one to be familiar with declarative programming paradigm. The tool support provided with Ark enables one to automatically obtain these Alloy models from a SosADL description and also to read the Alloy solution back as a SosADL model, hence concealing the use of a constraint solving tool in the method. Therefore, Ark enables one to use this formal method in the design of the SoS architectures without the additional burden that follows mastering a new notation.

In this scenario, the purpose of our case study has been twofold: (i) to demonstrate the method Ark for the synthesis of SoS concrete architectures, and (ii) to serve as a guide for practitioners and researchers who want to apply SosADL in the design of smart systems. The expected outcome of the method is a minimal concrete architecture that satisfies the abstract architecture description. Therefore, if this is not accomplished, one can either investigate if the constraints that govern the abstract architecture are correct or if the analysis scope is sufficient to cover at least one solution. Because the Alloy Analyzer will look for the minimal concrete architecture within such an analysis scope, we can work with a smaller instance of the concrete architecture to validate the suitability of the constraints that have been defined to govern the interactions within a coalition, which can also be more efficient since the constraint solver will perform an exhaustive search.

In practice, however, there can be multiple (e.g., dozens or hundreds) of elements in a coalition such as the one in our case study. Nonetheless, the interactions between these elements would still be governed by the same constraints. By focusing on the minimal concrete architecture of a SoS, we can more easily detect errs within the specification of these interactions. For instance, we can investigate if undesirable interactions are also feasible from the abstract architecture description so that we can refine our model to prevent these interactions at run-time. In this scenario, we can investigate complementary approaches for Ark to animate concrete architectures, hence enabling to simulate and/or compare emergent behaviors that can be expected from different coalitions at run-time. To do so, additional mechanisms are needed to scale out these minimal concrete architectures to represent real systems.

6 Related Work

Due to SoS evolutionary development, it is important to guarantee that changes to the abstract architecture are performed within a predefined time frame, specially given that SoS can perform safety-critical missions. Therefore, we presented in this article a constraint-based method to automatically determine the feasibility of an abstract architecture expressed with SosADL. The main advantages of our method can be summarized as follows: (i) it formally defines the rules governing the formation of coalitions at run-time; (ii) it limits the need for human intervention in the verification of such abstract architectures; and (iii) it automatically presents the output of a constraint solver as a concrete architecture expressed in SosADL, which can be more easily understood by SoS architects.

In this scenario, Ark can be compared to works that use constraint programming for modeling dynamic software systems. For instance, Sawyer et. al [64] transform a goal model that captures the variability in context demands (e.g. environment conditions and quality of service) in terms of a constraint satisfaction problem that can be automatically analyzed by a tool. Even though their approach is closer to requirements modeling than abstract architectures, one could further explore the relation that exists between the elements that have been selected to compose an abstract architecture and the goals that the SoS is able to accomplish, as discussed by Silva et. al [65]. Kogekar et. al [37] presents an approach to support the reconfiguration of explicit models of a system by formalizing the relationship between subsequent configurations in terms of constraints. In this respect, their approach is closer to the task of reconfiguring concrete architectures of the SoS, supporting the investigation of subsequent stages of SoS development that do not require changes to the abstract architecture. Given the complex nature of designing autonomous systems, Nafz et. al [48] introduce a design guideline for modeling the behavior of agent-based systems that is expressed in KodKod [66], an alternative to the Alloy Analyzer that accepts a subset of the Alloy language. Their approach instructs the solver to look for a reconfiguration of the system when elements of the solution have already been fixed by the problem. However, their formalization lacks some of the architectural elements

that would be needed to represent SoS concrete architectures in terms of ADLs.

We can also compare our work to research on model checking for software architectures [70]. Heyman et. al [23] propose a metamodel of software architectures in Alloy, focusing on security aspects of these software architectures, such as integrity issues, availability issues, and timing-related issues. The authors introduce an Alloy metamodel of software architectures that cover a subset of UML constructs, including signatures for components, connectors, and interfaces. Our metamodel is based on a novel notation for the description of SoS architectures that supports the definition of interconnections (i.e., unifications) in terms of constraints and the composition of systems and mediators within an architecture, offering additional elements for the description of complex models for smart systems architectures. In contrast, the Alloy model proposed by Kezniki et al. [35] is focused on the synthesis of connectors, which can be determined from the components and the communication patterns that are defined for the architecture. The outcome of their method is a connector instance configuration that establishes how these components are composed together in the architecture. Their approach takes into account non-functional properties, which describe structured/enumerated features, and roles for these connectors. In contrast, our method focuses on bridging the gap between abstract and concrete architectures by investigating topologies that can be created from constraints defined over the interconnections between constituents and mediators.

There are also other initiatives for designing and analyzing SoS based on ADLs, such as DANSE[6] (Designing for Adaptability and evolutioN in System of systems Engineering) and COMPASS[7] (Comprehensive Modelling for Advanced Systems of Systems). DANSE employs SysML to the description of executable architectures that can be analyzed against interface contracts. This approach is aimed at supporting the generation of new architectures as well as generation of subsequent ones by means of transformation steps between evolution. On the other hand, COMPASS develops a formal approach and applies CML for enriching the specification of systems and interfaces with contracts. One disadvantage of the latter is that an automatic transformation of SysML into CML can produce large, unreadable descriptions.

Kenley et. al [34] define a process for synthesizing SoS architectures based on three different models of the SoS, namely: a functional architecture, defining a sequence for sequentially executing actions that allow to accomplish a mission; a physical architecture, defining a set of physical capabilities (e.g., sensors, databases, and communication links); and, an allocated architecture, which assigns functional capabilities to physical components. In this scenario, abstract architectures are related to the objectives of functional and physical architectures whilst an allocated architecture is closer to concrete architectures. A dynamics mode is used in their work to express the dynamic behavior of allocated architectures as input to an executable model implemented in Discrete Agent Framework (DAF)[47], which is based on MATLAB. The task of creating allocated architectures is delegated to a model builder developed in DAF that replaces explicit definitions of arrangements and interconnections by a physical network, defining available point-to-point links, and an agent data path, selecting constituents that can be connected. Assumptions and the link allocation algorithm are employed to tailor architectures according to architects preferences, e.g., by choosing faster physical links instead of shortest path. Then, different techniques can be used, including UML activity diagrams for modeling the dynamics model and transforming then in Petri nets to create an executable model. In this regard, the main difference with our work is that SoSy, built-in in the SosADE development environment for SosADL, focuses on bridging the gap between abstract and concrete architectures, raising the abstraction level with which architects can design coalitions. Therefore, additional work is needed in order to support the simulation of SoS concrete architectures expressed in SosADL as it is proposed by Graciano Neto et. al [50].

Our work can also be compared with approaches that use formal architectural models in the description of SoS. Baldwin et. al [2] use set theory for mathematically representing SoS characteristics by means of systems, goals, and actions. Their approach models: (i) autonomy, as the cardinality of the set of actions that each system contributes for achieving the SoS goals; (ii) belonging, as the ratio between actions that a system contributes for the SoS goal and its autonomy (e.g., systems can only participate in the SoS if their belonging is greater than a threshold, which is inversely proportional to their contributed value); and (iii) dynamic connectivity, which is enabled when two systems share at least one connector and disabled when any system contribution stand below their belonging threshold. Using agent-based modeling, they simulated if it is possible to dynamically create coalitions given constituents' autonomy, belonging, and connectivity properties. In comparison to our approach, their model still lacks abstractions for the description of abstract types

[6]DANSE, `www.danse-ip.eu`

[7]COMPASS, `www.compass-research.eu`

and coalitions, which would enable to specify correctness properties about interconnections.

Khlif et. al [36] focus on the decomposition and refinement of an SoS architecture expressed in SysML. In particular, their approach does not cover the topology of the architecture, i.e., how constituents and mediators are connected together, and the modifications that could be applied to the software architecture. In comparison, Ark is based on a novel formal language, SosADL, that is semantically and syntactically well-defined for expressing SoS. Finally, empirical evidence for cost-benefit of a new method is needed for its dissemination in industry [44]. Even though we have demonstrated the potential of our method in an illustrative scenario of urban river monitoring, additional cases are needed. The existence of tool support for our method is expected to ameliorate this issue, since it automatically creates models that are required by Ark, thereby reducing the investment for applying this method in practice.

7 Conclusions

The evolutionary development of SoSs requires software architectures that can support changes in constituent systems for the realization of emergent behaviors. The main contribution of our method to the state-of-the-art is to provide mechanisms that enable to automatically verify the feasibility of a SoS abstract architecture at run-time based on correctness properties specified at design time. In particular, this method conceals the use of constraint solvers, which automatically synthesize concrete architectures that adhere to an abstract description of the SoS. Custom tool support, named SoSy, is also provided, automating the transformation of SoS abstract architectures in terms of a Constraint Satisfaction Problem (CSP) and the subsequent translation of the output for this problem as a concrete architecture of the SoS. We demonstrated this method and its accompanying tool support in a case study where we confirm the feasibility of an abstract architecture for a urban river monitoring SoS.

As future research, we will investigate means to ameliorate the efficiency of our method, which is sensitive to the selection of the analysis scope. We also intend to investigate the impact of particular architectural styles to the sustainability of abstract architectures and expand this method to support dynamic changes to abstract architectures, which directly impact concrete architectures. Examples of such changes encompass: (i) the addition of a new system to the coalition; (ii) unexpected self-termination of a communication link between systems due to internal conditions (e.g., low battery power or malfunctioning); or (iii) the optimization of internal and/or external resources. To achieve this goal, we will develop a mechanism that can be triggered by mediators and constituents for "sensing" the environment, e.g. discovering nearby constituents and voluntarily sharing information. Finally, a qualitative evaluation about the usability of our method and tool is also important for tailoring a design and development environment for the SosADL language based on the needs of researchers and practitioners. For instance, it can help identify the main challenges for the design of abstract architectures and offer additional guidance for these tasks. Therefore, such an investigation is certainly useful for minimizing human errs during the analysis of abstract and concrete architectures.

Acknowledgements The authors would like to thank the reviewers who helped to improve this paper. The authors also thank the researchers at the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) who have contributed to the SosADL language.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Avgeriou, P., Stal, M., Hilliard, R.: Architecture Sustainability. IEEE Software pp. 41–44 (2013). DOI 10.1109/MS.2013.120
2. Baldwin, W.C., Sauser, B.: Modeling the characteristics of system of system. In: IEEE International Conference on System of Systems Engineering (SoSE), pp. 1–6. Albuquerque, United States (2009)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3 edn. Addison-Wesley (2012)
4. Boardman, J., Sauser, B.: System of Systems - the meaning of of. In: IEEE/SMC International Conference on System of Systems Engineering (SoSE), pp. 1–6. IEEE, Los Angeles, United States (2006). DOI 10.1109/sysose.2006.1652284
5. Boehm, B.: A view of 20th and 21st century software engineering. In: International Conference on Software Engineering (ICSE), pp. 12–29. ACM Press, Shanghai, China (2006). DOI 10.1145/1134285.1134288
6. Boehm, B.: Some Future Software Engineering Opportunities and Challenges. In: S. Nanz (ed.) The Future of Software Engineering, pp. 1–32. Springer (2011)
7. Boehm, B., Brown, W., Basili, V., Turner, R.: Spiral Acquisition of Software-Intensive Systems-of-Systems. Crosstalk pp. 4–9 (2004)
8. Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. Information and Software Technology **54**(1), 16–40 (2012). DOI 10.1016/j.infsof.2011.06.002
9. Broy, M.: Seamless Method- and Model-based Software and Systems Engineering. In: S. Nanz (ed.) The Future of Software Engineering, pp. 33–47. Springer (2011)

10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented Software Architecture: A System of Patterns, vol. 1. John Wiley & Sons (1996)
11. Chattopadhyay, D., Ross, A.M., Rhodes, D.H.: A Framework for Tradespace Exploration of Systems of Systems. In: Conference on Systems Engineering Research (CSER), pp. 1–13. Los Angeles, United States (2008)
12. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, 2 edn. Addison-Wesley (2011)
13. Cook, S.A.: The Complexity of Theorem Proving Procedures. In: Annual ACM Symposium on Theory of computing (STOC). ACM Press (1971). DOI 10.1145/800157.805047
14. Dagli, C.H., Kilicay-Ergin, N.: System of Systems Architecting. In: M. Jamshidi (ed.) System of Systems Engineering, pp. 77–100. John Wiley & Sons, Inc. (2009). DOI 10.1002/9780470403501.ch4
15. Dahmann, J., Rebovich, G., Lane, J.A., Lowry, R.: System Engineering Artifacts for SoS. IEEE Aerospace and Electronic Systems Magazine **26**(1), 22–28 (2011). DOI 10.1109/MAES.2011.5719652
16. Dahmann, J., Rebovich, G., Lowry, R., Lane, J.A., Baldwin, K.: An Implementers' View of Systems for Systems of Systems. In: IEEE International Systems Conference (SysCon), pp. 212–217 (2011). DOI 10.1109/SYSCON.2011.5929039
17. Gagliardi, M., Bergey, J., Wood, B.: System of Systems (SoS) Architecture Centric Acquisition. [*On-line*], *World Wide Web* (2010). URL `https://resources.sei.cmu.edu/asset_files/Presentation/2010_017_001_53032.pdf`
18. Guessi, M., Cavalcante, E., Oliveira, L.B.R.: Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems. In: IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems (SESoS), pp. 12–18. IEEE, Florence, Italy (2015). DOI 10.1109/sesos.2015.10
19. Guessi, M., Neto, V.V.G., Bianchi, T., Felizardo, K.R., Oquendo, F., Nakagawa, E.Y.: A systematic literature review on the description of software architectures for systems of systems. In: Annual ACM Symposium on Applied Computing (SAC), pp. 1442–1449. ACM Press, Salamanca, Spain (2015). DOI 10.1145/2695664.2695795
20. Guessi, M., Oquendo, F., Nakagawa, E.Y.: Checking the Architectural Feasibility of Systems-of-Systems using Formal Descriptions. In: System of Systems Engineering Conference (SoSE), pp. 1–6. IEEE, Kongsberg, Norway (2016). DOI 10.1109/sysose.2016.7542939
21. Guessi Margarido, M.: Synthesis of software architectures for systems-of-systems: an automated method by constraint solving. Ph.D. thesis, Institute of Mathematics and Computer Science, University of São Paulo, Available at `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06022018-105449/` (2017)
22. Harmon, R.R., Corno, F., Castro-Leon, E.G.: Smart systems. IT Pro pp. 14–17 (2015). DOI 10.1109/mitp.2015.115
23. Heyman, T., Scandariato, R., Joosen, W.: Security in context: analysis and refinement of software architectures. In: IEEE Annual Computer Software and Applications Conference (COMPSAC), pp. 161–170. IEEE, Seoul, South Korea (2010). DOI 10.1109/compsac.2010.23
24. Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A., America, P.: A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software **80**(1), 106–126 (2007). DOI 10.1016/j.jss.2006.05.024
25. Hughes, D., Thoelen, K., Horré, W., Matthys, N., Del Cid, J., Michiels, S., Huygens, C., Joosen, W.: LooCI: A loosely-coupled component infrastructure for networked embedded systems. In: International Conference on Advances in Mobile Computing and Multimedia (MoMM), pp. 195–203. ACM, Kuala Lumpur, Malaysia (2009). DOI 10.1145/1821748.1821787
26. Hughes, D., Ueyama, J., Mendiondo, E., Matthys, N., Horré, W., Michiels, S., Huygens, C., Joosen, W., Man, K.L., Guan, S.U.: A middleware platform to support river monitoring using wireless sensor networks. Journal of the Brazilian Computer Society **17**(2), 85–102 (2011). DOI 10.1007/s13173-011-0029-3
27. ISO/IEC/IEEE 42010: International Standard for Systems and Software Engineering – Architectural description (2011)
28. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology **11**(2), 256–290 (2002). DOI 10.1145/505145.505149
29. Jackson, D.: Dependable Software by Design. Scientific American pp. 69–75 (2006). DOI 10.1038/scientificamerican0606-68
30. Jackson, D.: Software Abstractions, rev. edn. MIT University Press Group Ltd (2012)
31. Jamshidi, M. (ed.): System of Systems Engineering: Innovations for the Twenty-First Century. John Wiley & Sons (2008)
32. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 1–10. IEEE, Pittsburgh, USA (2005). DOI 10.1109/wicsa.2005.61
33. Jaspan, C., Keeling, M., Maccherone, L., Zenarosa, G.L., Shaw, M.: Software Mythbusters Explore Formal Methods. IEEE Software pp. 60–63 (2009). DOI 10.1109/ms.2009.188
34. Kenley, C.R., Dannenhoffer, T.M., Wood, P.C., DeLaurentis, D.A.: Synthesizing and Specifying Architectures for System of Systems. In: INCOSE International Symposium, vol. 24, pp. 94–107 (2014)
35. Keznikl, J., Bureš, T., Plášil, F., Hnětynka, P.: Automated resolution of connector architectures using constraint solving (ARCAS method). Software & Systems Modeling **13**(2), 843–872 (2014). DOI 10.1007/s10270-012-0274-8
36. Khlif, I., Kacem, M.H., Kacem, A.H., Drira, K.: A multi-scale modelling perspective for SoS architectures. In: European Conference on Software Architecture Workshops (ECSAW), pp. 1–5. ACM Press, Vienna, Austria (2014). DOI 10.1145/2642803.2642833
37. Kogekar, S., Neema, S., Eames, B., Koutsoukos, X., Ledeczi, A., Maroti, M.: Constraint-guided dynamic reconfiguration in sensor networks. In: International Symposium on Information Processing in Sensor Networks (IPSN), pp. 379–387 (2004). DOI 10.1109/IPSN.2004.239229
38. Koontz, R.J., Nord, R.L.: Architecting for Sustainable Software Delivery. CrossTalk pp. 14–19 (2012)
39. Kramer, J., Magee, J.: A Rigorous Architectural Approach to Adaptive Software Engineering. Journal

of Computer Science and Technology **24**(2), 183–188 (2009). DOI 10.1007/s11390-009-9216-5
40. Kruchten, P.: Documentation of Software Architecture from a Knowledge Management Perspective – Design Representation. In: M.A. Babar, T. Dingsøyr, P. Lago, H. van Vliet (eds.) Software Architecture Knowledge Management Theory and Practice, pp. 39–57. Springer (2009)
41. Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Software Engineering for Self-Adaptive Systems II, pp. 1–32 (LNCS 7475) (2013). DOI 10.1007/978-3-642-35813-5_1
42. hua Lu, H., Guo, F., Huang, F., de Chen, R.: The Construction of Smart City Based on SoS. In: International Conference on Advanced Computer Science and Electronics Information ICACSIS, pp. 34–37. Atlantis Press, Beijing, China (2013). DOI 10.2991/icacsei.2013.9
43. Maier, M.W.: Architecting principles for systems-of-systems. In: INCOSE International Symposium, vol. 6, pp. 565–573 (1996). URL `http://dx.doi.org/10.1002/j.2334-5837.1996.tb02054.x`
44. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering **39**(6), 869–891 (2013). DOI 10.1109/tse.2012.74
45. Mandrioli, D.: On the Heroism of really Pursuing Formal Methods. In: IEEE/ACM FME Workshop on Formal Methods in Software Engineering, pp. 1–5. IEEE, Florence, IT (2015). DOI 10.1109/formalise.2015.8
46. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1), 70–93 (2000). DOI 10.1109/32.825767
47. Mour, A., Kenley, C.R., Davendralingam, N., DeLaurentis, D.: Agent-based Modeling for Systems of Systems. In: INCOSE International Symposium, vol. 23, pp. 973–987. Wiley (2013). DOI 10.1002/j.2334-5837.2013.tb03067.x
48. Nafz, F., Ortmeier, F., Seebach, H., Steghfer, J.P., Reif, W.: Universal Self-Organization Mechanism for Role-Based Organic Computing Systems. In: Autonomic and Trusted Computing (ATC), pp. 17–31 (LNCS v. 5586). Springer, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-02704-8_3
49. Nakagawa, E.Y., Gonçalves, M., Guessi, M., Oliveira, L.B.R., Oquendo, F.: The state of the art and future perspectives in systems of systems software architectures. In: International Workshop on Software Engineering for Systems-of-Systems (SESoS), pp. 13–20. ACM Press, Montpellier, France (2013). DOI 10.1145/2489850.2489853
50. Neto, V.V.G., Paes, C.E.B., Garcés, L., Guessi, M., Manzano, W., Oquendo, F., Nakagawa, E.Y.: Stimuli-SoS: a model-based approach to derive stimuli generators for simulations of systems-of-systems software architectures. Journal of the Brazilian Computer Society **23**(1), 1–22 (2017). DOI 10.1186/s13173-017-0062-y
51. Nielsen, C.B., Larsen, P.G., Fitzgerald, J., Woodcock, J., Peleska, J.: Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. ACM Comput. Surv. **48**(2), 1–41 (2015). DOI 10.1145/2794381
52. Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering: Systems Engineering Guide for Systems of Systems . [*On-line*], *World Wide Web* (2008). URL `http://www.acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf`
53. OMG: Software and Systems Process Engineering Meta-Model Specification v2.0. [On-line] (2008). `http://www.omg.org/spec/SPEM/2.0/`
54. OMG: Unified Modeling Language v2.4.1. [On-line] (2011). `http://www.omg.org/spec/UML/2.4.1/`
55. OMG: Systems Modeling Language v1.3. [On-line] (2012). `http://www.omgsysml.org/`
56. Oquendo, F.: Formally describing the software architecture of systems-of-systems with SosADL. In: System of Systems Engineering Conference (SoSE), pp. 1–6. IEEE, Kongsberg, Norway (2016). DOI 10.1109/SYSOSE.2016.7542926
57. Oquendo, F.: π-calculus for SoS: A foundation for formally describing software-intensive systems-of-systems. In: System of Systems Engineering Conference (SoSE), pp. 1–6. Institute of Electrical and Electronics Engineers (IEEE), Kongsberg, Norway (2016). DOI 10.1109/sysose.2016.7542925
58. Oquendo, F.: Software Architecture Challenges and Emerging Research in Software-intensive Systems-of-Systems. In: European Conference on Software Architecture (ECSA), pp. 3–21. Springer, Copenhagen, Denmark (2016). DOI 10.1007/978-3-319-48992-6
59. Oquendo, F., Buisson, J., Leroux, E., Mogurou, G., Quilbeuf, J.: SoS ADL for Formal Architecture Description and Analysis of Software-intensive Systems-of-Systems (2016). Presentation at the Colloquium on Software-intensive Systems-of-Systems at ECSA
60. Pérez, J., Díaz, J., Garbajosa, J., Yagüe, A., Gonzalez, E., Lopez-Perea, M.: Large-scale smart grids as system of systems. In: International Workshop on Software Engineering for Systems-of-Systems (SESoS), pp. 38–42. ACM Press, Montpellier, FR (2013). DOI 10.1145/2489850.2489858
61. Rhodes, D.: Evolving Systems Engineering for Innovative Product and Systems Development. In: Massachusetts Institute of Technology (MIT) Systems Design and Management Alumni Conference (2004)
62. Roure, D.D.: Floodnet: A new flood warning system. Ingenia **23**, 50–51 (2005)
63. Sassone, A., Grosso, M., Poncino, M., Macii, E.: Smart Electronic Systems: an Overview. In: N. Bombieri, M. Poncino, G. Pravadelli (eds.) Smart Systems Integration and Simulation, pp. 5–21. Springer (2016). DOI 10.1007/978-3-319-27392-1
64. Sawyer, P., Mazo, R., Diaz, D., Salinesi, C., Hughes, D.: Using constraint programming to manage configurations in self-adaptive systems. Computer **45**(10), 56–63 (2012). DOI 10.1109/MC.2012.286
65. Silva, E., Batista, T., Oquendo, F.: A mission-oriented approach for designing system-of-systems. In: 10th System of Systems Engineering Conference (SoSE). IEEE (2015). DOI 10.1109/sysose.2015.7151951
66. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 632–647. Springer Berlin Heidelberg, Braga, Portugal (2007). DOI 10.1007/978-3-540-71209-1_49
67. Ulieru, M., Doursat, R.: Emergent engineering: a radical paradigm shift. Int. J. Autonomous and Adaptive Communications Systems **4**(1), 39–60 (2011). DOI 10.1504/ijaacs.2011.037748
68. Valerdi, R., Ross, A.M., Rhodes, D.H.: A Framework for Evolving System of Systems Engineering. Crosstalk pp. 28–30 (2007)

69. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer (2012)
70. Zhang, P., Muccini, H., Li, B.: A classification and comparison of model checking software architecture techniques. Journal of Systems and Software **83**(5), 723–744 (2010). DOI 10.1016/j.jss.2009.11.709

Milena Guessi is a Postdoctoral Research Fellow at the University of Toronto, Canada, and the University of São Paulo (USP), Brazil, in the field of Health Informatics. She obtained her B.Sc. in Computer Science and her M.Sc. in Software Engineering from USP. She received her PhD in Computer Science from USP and the University of South Brittany, France, in 2017. She has experience with model-based tools and methods for the analysis and description of software architectures of complex systems and systems-of-systems (SoS). She has served as a referee for journals and conferences on information systems and software architectures, including JSS, IST, ECSA, ICIS, and SoSE. Her research interests encompass software engineering for eHealth, specially in regards to addressing human values in the analysis and design of software architectures.

Flavio Oquendo is a Full Professor of Computing at the University of South Brittany, acting as Research Director on Software Architecture at the IRISA (UMR CNRS 6074), France. He received his BEng from ITA, Sao Jose dos Campos, SP, Brazil, and his MSc, PhD and HDR from the University of Grenoble, France. He has been a recipient of the Research Excellence Award from the French Ministry of Research and Higher Education. He has supervised more than 30 PhD theses over the years. Prof. Oquendo has published over 200 refereed papers in international journals and international conference proceedings and has been editor of over 25 journal special issues and research books. He has served on program committees of over 200 international conferences, including the key ones in his field of research, e.g. ACM/IEEE ICSE, ESEC/FSE, ICSA/WICSA, ECSA, SoSE, ICECCS, ICSP, has chaired many of them, in particular the French, European, and IEEE/IFIP International Conferences on Software Architecture (CAL, ECSA, WICSA). His research interests are centered on formal languages, processes and tools to support the efficient architecture of complex software-intensive systems and systems-of-systems. His web page is http://people.irisa.fr/Flavio.Oquendo/

Elisa Yumi Nakagawa is an associate professor in the Department of Computer Systems at the University of São Paulo - USP, Brazil. She conducted her post-doctoral research at Fraunhofer IESE, Germany, in 20112012 and at the University of South Brittany, France, in 2014-2015. She received her PhD degree from USP in 2006. She coordinates international/national research projects in software architecture, reference architectures, and systems-of-systems, and also supervises PhD/Masters students and post-doctoral researchers. She has organized international/national conferences and has served as a program committee member at many conferences and as a reviewer of various journals. She is a member of IEEE and SBC (Brazilian Computer Society).