Coordination-aware Elasticity

Stefano Mariani*, Hong-Linh Truong◇, Georgiana Copil◇, Andrea Omicini*, Schahram Dustdar◇
*DISI, ALMA MATER STUDIORUM–Università di Bologna, 47521 Cesena, Italy
Email: {s.mariani, andrea.omicini}@unibo.it
◇Distributed Systems Group, Vienna University of Technology
Email: {truong, e.copil, dustdar}@dsg.tuwien.ac.at

***Abstract*—Enabling and controlling elasticity of cloud computing applications is a challenging issue. Elasticity programming directives have been introduced to *delegate* elasticity control to infrastructures and to *separate* elasticity control from application logic. Since *coordination models* provide a general approach to manage interaction and elasticity control entails interactions among cloud infrastructure components, we present a coordination-based approach to elasticity control, supporting delegation and separation of concerns at design and *run-time*, paving the way towards *coordination-aware elasticity*.**

I. INTRODUCTION

Elasticity is a fundamental concept in cloud computing, enabling different kinds of run-time changes w.r.t. cost, quality and resource dimensions associated with cloud applications while managing their *dependencies* [1]. To support elasticity, existing cloud infrastructures provide developers with low-level APIs to monitor and control system's properties/components/services (e.g., JClouds[1], SlipStream[2], Google Cloud Deployment Manager[3]). Implementing complex elasticity tradeoffs requires great effort in dealing with such APIs. To simplify the task, rules have been introduced, mostly to deal with innovations of elasticity APIs [2], [3]. Still, such rules are difficult to program and change at run-time. Thus, *elasticity programming directives* are introduced in [1] as a means to *(i) delegate* elasticity control to infrastructures and *(ii) separate* elasticity control from application logic.

Coordination is a fundamental concept in agent-oriented computing, enabling management of *dependencies* among agents' activities so as to ensure they all behave according to the goals of the system as a whole [4]. Coordination models and languages are provided to multi-agent systems engineers so as to enable and constraint interactions of agents [5]–[7]. Any coordination model is built out of three ingredients: a coordination medium, a set of coordination primitives and a set of *coordination laws* [8]. Coordination laws are meant to enable both *delegation* of interaction management to middleware and *separation* of interactive behaviour from computational one.

Although elasticity and coordination theories were born in different contexts and evolved independently, a fundamental similarity motivates our contribution. Both are concerned with *managing run-time dependencies*: the former, between elasticity dimensions of cloud applications, the latter, between activities of agents in a multi-agents system. In order to do so, the former relies on *elasticity enforcement mechanisms* to influence the way in which cloud application and infrastructure's components/services/resources behave and interact as well as their properties, while the latter exploits *coordination primitives* to influence the way in which agents interact as well as the outcome of such interactions. Furthermore, for composing, respectively, mechanisms and primitives into complex elasticity and coordination "procedures", elasticity and coordination exploit similar concepts: *elasticity programming directives* and *coordination laws*. In this paper, we take a first step toward the notion of *coordination-aware elasticity*, by integrating elasticity control with coordination support.

A. Motivation

Elasticity controllers, in charge of executing elasticity programming directives, are directly responsible for supporting and enforcing elasticity in the cloud, by carefully monitoring and controlling changes in cost, quality and resource dimensions of cloud applications, as well as their *dependencies*—through enforcement of elasticity mechanisms. Elasticity controllers usually rely on a plugin mechanism to support multi-cloud deployments and/or application/cloud-specific controls [9], [10], allowing stakeholders to develop their own customizations: thus, they have to efficiently and correctly manage *interactions* among such plugins.

In other words, elasticity controllers need to deal with *coordination*-related aspects to successfully support elasticity. Such coordination, if implemented by elasticity controllers, is a cumbersome process prone to errors–both for the developer and for the software component:

- enforcement mechanisms need parameters (waiting times, number of tries, targets of operations, etc.) often hard-coded—at most, configurable at design-time
- synchronization between enforcement mechanisms is poor (hard-coded or undefined waiting times, missing knowledge about operations success/failure, etc.)
- control flow "escapes" elasticity controller down to low-level cloud APIs, exposing to failures/bugs and hindering portability as well as code reuse

This way, not only flexibility, safety and encapsulation, but also separation and delegation of concerns are hindered, because elasticity controllers must deal with coordination issues. Thus integration of elasticity and coordination is needed, to be done at three levels of abstraction:

- the meta-model (conceptual) level: defining the abstractions to be used while thinking about the solution of the problem at hand—enforcing coordination-aware elasticity

This work was partially supported by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790)

[1]https://jclouds.apache.org/

[2]http://sixsq.com/products/slipstream.html

[3]http://goo.gl/vksPCP

- the model (language) level, defining how languages can be designed to express the solution, given meta-model abstractions—through elastic directives and coordination laws
- the technology (infrastructure) level: defining how both the abstractions and the language can be supported by a suitable architecture and runtime—by the elasticity runtime integrated with the coordination infrastructure

B. Contribution and paper structure

To address the challenges discussed above, in this paper, we (i) study relationships between elasticity and coordination, (ii) present patterns in elasticity control and coordination supporting coordination-aware elasticity, and (iii) provide a prototype based on rSYBL[4] elasticity controller and ReSpecT[5] coordination framework, showing how different stakeholders can use the coordination-driven control mechanisms for elasticity control. Accordingly, in Section III we incrementally narrow the context in which integration is performed, starting from the more abstract concepts behind elasticity and coordination, down to the concrete languages and runtimes used, respectively, for coding elasticity programming directive and coordination laws. Besides solving the aforementioned issues, benefits of integrating elasticity with coordination are:

- support *run-time* delegation and separation of concerns, by letting independent infrastructural components handle one aspect of computation (elasticity or coordination), while delegating the other to whom is responsible for it
- guarantee *safety* of interactions between elasticity controllers and cloud components/services/plugins, by relying on well-defined coordination primitives and laws
- improve *availability* of elasticity controllers, by distributing coordination-related computations to dedicated components, thus reducing computational load on the formers
- ease *development* process, by enabling and supporting separation of duties and responsibilities between "elasticity developers" and "coordination developers"

While Section II overviews background for our work, Section III presents the core concepts for integrating coordination aware elasticity. Section IV shows a case study depicting coordination-aware elasticity control for cloud services, Section V describes related work, and finally Section VI concludes the paper.

II. Background

A. Elasticity & Coordination: Concepts

Elasticity Control: *Elasticity programming directives* [1] are special statements (e.g., Java annotations in the case of SYBL [3]) enriching cloud applications' code to define how application's and infrastructure's elastic components should behave in response to elasticity-related events. Elasticity directives must enable different stakeholders to specify elasticity requirements at different levels of abstraction (e.g., for components, or for groups of components), describing their goals in terms of desirable application state, or strategies/rules to be triggered under specific conditions.

Coordination: *Coordination laws* describe how agents (or, *coordinables*) coordinate through a given *coordination medium* using given *coordination primitives* [8]: a coordination medium is the component enacting coordination laws ruling interaction among coordinables—which are, e.g., processes, application services, agents, human users. Examples of coordination media are concurrency-related abstractions such as semaphores and monitors, distributed systems abstractions such as channels and pipelines, and more advanced abstractions such as tuple spaces [11]. Examples of coordination primitives are thus acquiring/releasing a lock, sending/receiving messages, and LINDA primitives such as `out`, `rd`, `in` [11].

B. Elasticity & Coordination: Languages

Elasticity Control: Following the directive programming model, the SYBL language is introduced in [3] as the concrete language to program elastic directives with. SYBL directives begin with keyword `#SYBL`, followed by specification of the category they belong to – either `MONITORING`, `CONSTRAINT` or `STRATEGY` – and by clauses, composing runtime functions, user-defined functions and variables to express the specific directive. Elasticity programming directives are composed of *elasticity primitives*, whose main types are:

- *Monitoring* primitives — enable gathering of information regarding the three main dimensions of elasticity – QoS, resources and costs – both from application services and from the underlying infrastructure.
- *Constraint* primitives — enable definition of conditions over system state, which are continuously tested using monitoring primitives and whose violation/compliance triggers execution of strategies.
- *Strategy* primitives — enable definition of actions to be taken to modify system state in response to monitored constraints violation/compliance or elasticity-related events happening.

Coordination: Once a coordination model has been defined in terms of coordinables, coordination media and laws, it has to be reified as a language, an architecture, or both [8]. The ReSpecT coordination language was originally introduced in [12] and later extended to support a growing number of application scenarios [13]. ReSpecT supports: *(i)* implementation of new primitives, *(ii)* modification of existing primitive semantics and *(iii)* change of coordination laws—everything at run-time. A ReSpecT program is a collection of *reaction specification tuples* of the form `reaction(Event,Guards,Body)`:

- `Event`: the representation of anything could happen within the coordinated system, e.g., coordinable interactions, flow of time, motion in space, environment change.
- `Guards`: the conditions about the `Event` that should hold when the reaction is triggered to actually schedule it for execution, e.g., the event is due to an agent, it has been raised after time `T`, it refers to a coordination operation failure, etc.
- `Body`: the computations to undertake in response to `Event` if and only if `Guards` hold, e.g., remove that tuple, replace this coordination law, spawn that process, perform this computation, etc.

[4] Available at https://github.com/tuwiendsg/rSYBL
[5] Available within TuCSoN: http://tucson.unibo.it

C. Elasticity & Coordination: Runtimes

Elasticity Control: Most of runtime features are mapped to functions provided by the underlying cloud systems. They are used within clauses to observe and control properties belonging to quality, resources and costs dimensions of elastic systems, e.g.: `balance([time])` may be used to check a service's cost balance at a given time; `set/get_env([prop])` may be used to (respectively) control/observe service-specific and infrastructure-specific properties—e.g., to get the bid price for a resource we could write `r_bid = get_env(''R_BID'');` `runscript([script_file])` executes user-defined functions and procedures.

Coordination: The execution model of coordination laws varies depending on the coordination model adopted, and according to its specific implementation too. Anyway, to give the reader a flavour of it, we briefly examine the case of ReSpecT—refer to [12]. ReSpecT reactions are "triggered" by events happening within the coordinated system (matching their `Event` description) and executed *atomically*, in a *non-deterministic order*, according to a *transactional semantics*. Thus: *(i)* nothing can interfere with a reaction execution, *(ii)* no assumptions can be made upon execution order if multiple reactions are triggered, *(iii)* if any computation within a reaction body fails, the whole reaction is aborted and any change made reverted. This ensures fundamental *safety* and *liveness* properties for a coordination model [12].

III. Integrating Elasticity and Coordination

A. Concepts: System View

In order to enforce elasticity, cloud systems are likely to be: *(i)* composed by services subjects of elasticity control, *(ii)* supported by an infrastructure having monitoring and control capabilities over both the hosted services and the computational resources, *(iii)* equipped with a suitable language to program the elasticity directives to be enforced at run-time. A coordinated system, in turn, to enforce coordination is likely to be: *(i)* composed by agents subject of coordination policies, *(ii)* supported by an infrastructure hosting the chosen coordination media, *(iii)* equipped with a language able to specify the coordination laws to be enforced at runtime.

Therefore, the following conceptual mapping between elasticity and coordination abstractions can be established:

Elasticity		Coordination
Elastic Service/Resource	$\Longleftrightarrow$	Agent (Coordinable)
Elastic Infrastructure	$\Longleftrightarrow$	Coordination Media
Programming Directives	$\Longleftrightarrow$	Coordination Laws

In particular, an elastic system may be interpreted as a coordinated system within which:

- elastic services and computational resources are the entities subject of the coordination process (the coordinables)
- such a process is supported by a suitable *elastic coordination infrastructure* composed by a network of distributed coordination media and elastic components
- such components are responsible for the run-time enforcement and programmability of the desired *elastic coordination directives*

Notice this does not mean we should always consider elastic systems in terms of coordination abstractions: it means elastic systems have the necessary traits to include coordination-related techniques to better support elasticity control.

B. Languages: Interoperability

Once our conceptual mapping is accepted, a linguistic mapping between elastic programming directives and coordination laws becomes feasible and natural. In fact, elasticity programming directives are composed of *monitoring* primitives, *constraints* and *strategies*, describing how services and resources should behave within the system according to its observable state and dynamics (events happening), through the usage of well-defined elastic primitives [1]. Coordination laws, in turn, describe how coordinables should behave according to the state of the interaction space and their observable interactions (generating events), through the usage of well-defined coordination primitives [8]. It is thus necessary for coordination laws to be capable of *observing* the interaction space and *computing* over it—to change its state and dynamics.

Accordingly, the following mapping can be established:

	Elasticity Directives		Coordination Laws
(1)	Directive	$\Longleftrightarrow$	Law
(2)	Runtime Functions	$\Longleftrightarrow$	Primitives
(3)	Monitoring	$\Longleftrightarrow$	Events
(4)	Constraint	$\Longleftrightarrow$	Observation
(5)	Strategy	$\Longleftrightarrow$	Computation

Whereas mapping 1 and 2 are quite straightforward, the others need further discussion.

Monitoring: Monitoring primitives are meant to perceive the *state* and *dynamics* of an elastic system, based on properties made observable by the supporting infrastructure, the hosted services, and the computational resources available. Thus, they are likely to assume a continuous monitoring process undertook by a dedicated monitoring component, either belonging to the infrastructure or to the application level [3]. On the contrary, coordination events are singular, point-wise occurrences in system dynamics and/or changes in system state, which are captured by the coordination medium and therein stored to be matched against laws [12]—thus, the monitor is the coordination medium itself. However, goal of both monitoring primitives and coordination events is the same: enabling the system to *react* to changes in its state.

Constraint: Constraint primitives are meant to check whether some given conditions about state of the elastic system hold. Such conditions are based on properties observed by the monitoring primitives and exploited by strategy primitives to control their own execution [3]. Their semantics is thus quite similar to that of coordination observation capabilities, which are meant to control whether triggered laws can be scheduled, mostly based on properties belonging to the triggering event or to system state [12]. Therefore, even if their execution model can be different – e.g., as in the case of SYBL and ReSpecT, where elasticity constraints are used within strategies and continuously monitored, whereas guards are checked upon triggering of the reaction and outside its body – their purpose is exactly the same: controlling execution of (elastic/coordination) computations in response to (elastic/coordination) events.

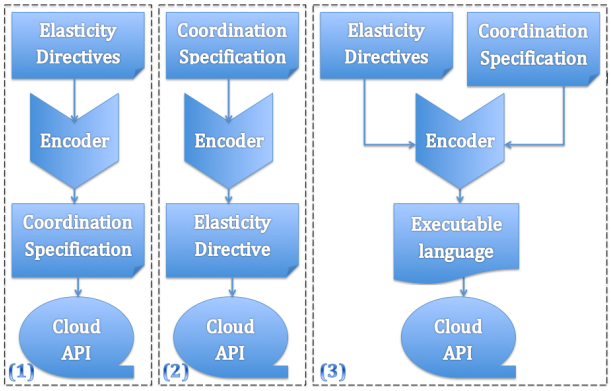

Fig. 1: Translational approaches to coordination-aware elasticity.

Strategy: Strategy primitives are meant to undertake computational actions modifying state and dynamics of the elastic system at hand, influencing service execution, resource properties and infrastructure behaviour/configuration [3]. Such actions are triggered in response to events regarding violation/-compliance of monitored conditions. Once again, a semantics similarity with computations inside coordination laws can be identified: within a law in fact, are put coordination and observation primitives, enabling the coordination medium to perform computations over the interaction space [12].

For the sake of clarity, in the case of SYBL and ReSpecT, the linguistic mapping can be specialized as follows:

	SYBL		ReSpecT
(1)	Directive	$\Longleftrightarrow$	Specification tuple
(2)	runtime functions	$\Longleftrightarrow$	Primitives/Predicates
(3)	`MONITORING`	$\Longleftrightarrow$	`Event`
(4)	`CONSTRAINT`	$\Longleftrightarrow$	`Guards`
(5)	`STRATEGY`	$\Longleftrightarrow$	`Body`

Summing up, thanks to the conceptual mapping between elasticity and coordination abstractions, it is always possible to find a meaningful match between elasticity programming directives and coordination laws language constructs—despite differences in their execution semantics. While virtually enabling either coordination or elasticity languages to be exploited in dealing with both issues, thus promoting interoperability, Subsection III-D highlights the importance of keeping each language focussed on its most natural duties—that is, although in synergy, coordination laws should deal with coordination-related issues whereas elasticity programming directives should take care of elasticity enforcement.

C. Runtimes: Separation of Concerns

Integrating languages is not enough to achieve full support to coordination-aware elasticity. It might be so by choosing to follow a *translational* approach to integration, that is, one of the following alternatives—depicted in Fig. 1: encoding any elasticity programming directive into coordination laws (Fig. 1, 1); encoding any coordination law into elastic directives (Fig. 1, 2); encoding programming directives and coordination laws independently, then translating them in a common executable language integrating their API and interfacing with cloud API (Fig. 1, 3).

Nevertheless, the above solutions share issues that make their adoption problematic: first of all, regardless of the translation chosen, the *separation of concerns* principle is disregarded, ending up with either a language taking charge of issues conceptually belonging to the other (translations 1 and 2) or a language mixing different responsibilities (as in the case of translation 3); furthermore, enabling *dynamic* change of elasticity and coordination policies *at run-time* requires encoders to be capable of performing on-the-fly translations, which is likely to become cumbersome. Thus, a different approach should be chosen, supporting both design and run-time separation of concerns by *delegation* of responsibilities among distributed components.

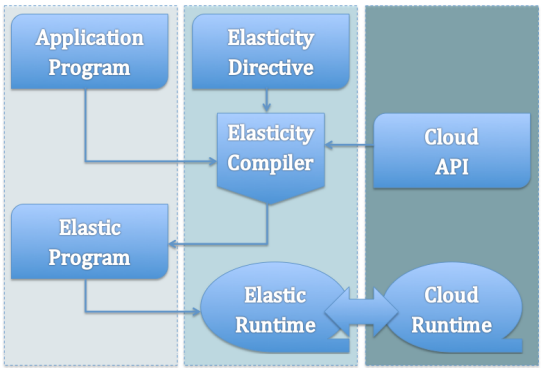

Fig. 2: Typical development and execution "flow" of an elastic application exploiting elasticity programming directives

Fig. 2 summarizes the main lifecycle stages of an application exploiting elasticity programming directives (e.g., encoded in SYBL)—based on [1]: application developers can write their code while elasticity developers code (e.g., in SYBL) directives independently; then, an "elasticity compiler" takes application code and elasticity directives integrating them in an executable elastic program, also considering API provided by the underlying cloud infrastructure as they are exploited within directives; finally, the program is executed on the cloud infrastructure by the cloud runtime and the elastic runtime (e.g., rSYBL) working in synergy.

Our goal is extending such a lifecycle to integrate coordination services, in particular a coordination language along with its runtime, in the development process as well as in the execution process. To this end, a suitable integration at the infrastructural level is required. Since the aim is not translating languages one into the other, there is no need to worry about how any possible elasticity directives statement can be translated into coordination laws statements, or vice versa: just an *integration API* is required, enabling elasticity directives to interact with coordination laws (and vice versa), therefore enabling distribution of computations between elasticity runtime (e.g., rSYBL) and coordination runtime (e.g., ReSpecT)—as depicted in Fig. 3. By design, this ensures that our integration solution will be able to both *(i)* make elasticity and coordination controllers work in tight synergy – thanks to the interoperability enabled by our linguistic mapping – and *(ii)* make the whole coordination-aware elasticity enforcement process capable of dynamically delegate responsibilities to the right component—supporting separation of concerns.

D. Runtimes Integration Guidelines

Monitoring API: Monitoring tasks can be carried out either with usual elasticity monitoring primitives, likely to directly

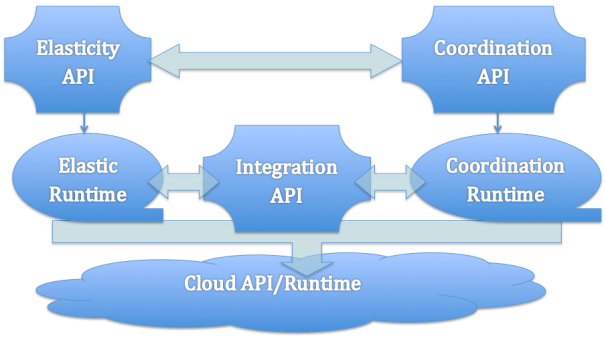

Fig. 3: Integration API and architecture supporting *coordination-aware elasticity*.

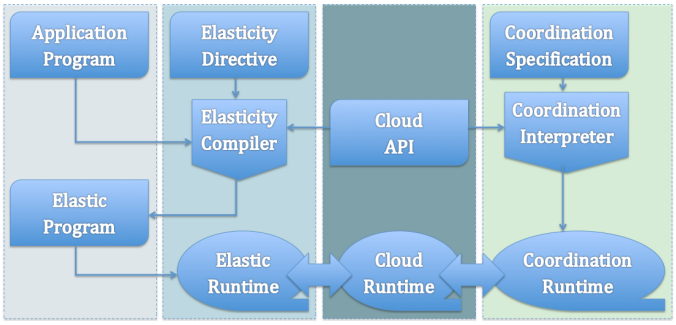

Fig. 4: Development and execution "flow" of a *coordination-aware* elastic application.

exploit underlying cloud platform API, or by using coordination laws. In principle, monitoring should always be done using monitoring primitives, being them explicitly conceived to do so. Nevertheless, whenever it is necessary to monitor measures or events not strictly related to elasticity (e.g., to manage out-cloud sensors or in-cloud synchronization of resources), coordination laws are preferred. Monitoring responsibilities are then split between elasticity and coordination runtimes, according to the principle of separation of concerns. If and when they need to exchange information or delegate computations, mapping (2) does the job: to communicate from elasticity to coordination, e.g., is sufficient to reify delegation requests into well-defined tuples, e.g., for ReSpecT to communicate with SYBL it is sufficient to call SYBL primitives within ReSpecT reactions using the `spawn` primitive, which runs a parallel process[6]. Definition of tuples and activities to be spawned shapes our *monitoring integration API*.

Constraint API: Constraints are constructs triggering events when some conditions over monitoring tasks are met [1]. Thus, the only issue to deal with is: if some constraints trigger events that have to be managed by coordination laws, such events should be suitably translated into well-defined tuples triggering execution of coordination laws. If monitoring is done with coordination laws, this is naturally achieved; if monitoring is carried out by elasticity directives, the translation has to be done within such directives. Accordingly, the only *constraints integration API* needed is the capability of elasticity controllers to request coordination services.

Strategy API: Execution of strategies is the process where coordination-aware elasticity mostly manifests. Thus, it is where most of the run-time *delegation* between elasticity (SYBL) and coordination (ReSpecT) mechanisms and components actually takes place. In fact, even the most basic and common elasticity-related task may imply non-trivial coordination issues, which are likely to become cumbersome to deal without the proper abstractions. Strategies are executed within elasticity directives, in response to elasticity constraints being violated/met, according to elasticity monitoring tasks, thus, on a "per-need" basis, they may delegate coordination tasks to suitably engineered coordination laws, which in turn, may rely on elasticity control API to query elasticity-related properties, or to ask for execution of elastic computations. Likewise monitoring, thus, strategies application responsibility is split between elasticity runtime and coordination runtime, according to the principle of separation of concerns. When elasticity programming directives (e.g., SYBL) and coordination laws (e.g., ReSpecT) need to exchange information, mapping (2) comes in hand again:

- whenever needed, elasticity controllers reify coordination requests as tuples putting them in the coordination medium working as the coordination manager; there, coordination laws are triggered to enact the desired coordination process, returning a completion result – in the form of a another tuple – indicating success or failure.
- whenever needed, the coordination medium interacts with elasticity controllers (e.g., rSYBL components) by calling elasticity primitives within `spawn`-ed processes; then, elasticity controllers serve such elasticity-related tasks returning a completion result—again, as a tuple.

Defining structure and content of tuples triggering coordination laws (ReSpecT reactions), the spawned processes calling runtime functions (SYBL primitives) and the components enabling elasticity control (rSYBL) to interact with coordination media (ReSpecT tuple centres), actually defines our *strategy integration API*.

Summing up, monitoring, constraint and strategy API altogether shape the *integration API* enabling the novel lifecycle stages for coordination-aware, elastic application development depicted in Fig. 4.

IV. Case Study: Integrating SYBL & ReSpecT

Following discussion in Section III, especially suggestions given in Subsection III-D, we present our integration proposal, focussing on the SYBL language and runtime (rSYBL) for elasticity control and the ReSpecT language and runtime (TuCSoN) for coordination support. To test feasibility of such approach to *coordination-aware elasticity*, then its effectiveness in terms of developers' efforts required as well as benefits provided (see Section I), we developed a prototype integration, which is available on GitHub[7].

A. Integration API

1) Monitoring API: Our proposed integration between SYBL and ReSpecT features:

- a spawned activity waiting to receive monitoring requests as ReSpecT tuples to enact the monitoring process calling rSYBL API—not reported due to the lack of space, but available on rSYBL repository. In particular:

[6] http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide [7] https://github.com/tuwiendsg/rSYBL/tree/coordination

- `sampleMetrics(·)` starts monitoring on the given SYBL node
- `metricToCheck(·,·)` filters metrics to monitor
- `monitoringTime(·)` configures monitoring timing

- a ReSpecT specification enabling run-time configuration of monitoring through such tuples

```
1 reaction(
2   out(sampleMetrics(NodeId)),
3   (operation, invocation),
4   ( out(metricToCheck(NodeId, 'cpuUsage')),
5     out(metricToCheck(NodeId, 'latency')),
6     out(monitoringTime(10000)) ) ).
```

In this way, run-time adaptation of monitoring configuration is possible by changing the above ReSpecT specification, using ReSpecT specification primitives such as `out_s` and `in_s`. Also, if monitoring logic has to be changed, this can be done both at design-time, by replacing, modifying or extending the spawned activity Java class, and at run-time, by adding a new spawned activity then telling TuCSoN where to find it.

2) Constraint API: The dedicated "bridge" component sketched in Fig. 5 has been developed—not all methods are shown. It enables rSYBL elasticity controllers to call ReSpecT primitives – e.g. coordination primitives `out`, `in` and specification primitives `out_s`, `in_s` – and TuCSoN services—e.g. to start and stop a coordination service medium. In this way, regardless whether a constraint regards elasticity properties or coordination state, it is always possible to check it from SYBL code. Such a bridge is also used to support integration with monitoring API as well as strategy API.

3) Strategy API: Tuples used in this API are:

- `doScaleOut(·)` starting scaling out
- `scaleOutTries(·)` setting attempts to make, in case of errors, before quitting
- `scaleOutTime(·)` configuring maximum waiting time, at each attempt, for the new node to become "active"

ReSpecT specification enabling for run-time adaptation is:

```
reaction(
  out(doScaleOut(NodeId)),
  (operation, invocation),
  ( out(scaleOutTries(10)),
    out(scaleOutTime(10000)) ) ).
```

The spawned process is not reported due to the lack of space. As for monitoring, design and run-time adaptation is possible by working on the spawned process or on ReSpecT reaction.

B. Developers' Overhead

To measure the "overhead" required to developers for adopting our coordination-aware approach to elasticity, it should be clear what "know-how" is necessary for them to acquire. This depends on whether they are satisfied with "built-in", already provided scaling and monitoring "logics" – that is, *how* scaling is performed, using *which* cloud API and how, e.g., doing synchronous or asynchronous calls –, or if they want more customization opportunities.

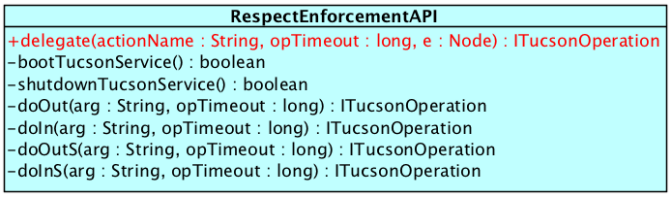

Fig. 5: Enabling SYBL to call ReSpecT primitives.

In the former case, developers should—e.g., in case of a "scale-out" primitive:

- either inform the runtime about where the file storing ReSpecT specification is in the filesystem – can be done in a configuration stage or even on the fly, as an argument of the coordination-aware primitive invoked, e.g., `scaleout('path/to/spec.rsp')` – or add such specification to ReSpecT runtime on the fly, using TuCSoN operations—e.g. `out_s(reaction)`.
- write such specification, using the ReSpecT language—either on a file or on the fly, within the body of an `out_s` operation (`doOutS` in Fig. 5)
- within SYBL directives, comply with the "actions syntax" specifically conceived for rSYBL-ReSpecT integration, e.g. `scaleout(args)`, `scalein(args)`, `monitor(args)`
- within ReSpecT specifications, comply with tuples' syntax as defined by the built-in "agents" (or spawned activities) provided by the integrated runtime—as shown in ReSpecT listings in Subsection IV-A

Summing up, only overhead is adding a bit of very basic declarative programming (similar to what reported in Subsection IV-A) to developers workflow, being careful to comply with given syntax. This is enough to gain all the benefits of coordination-aware elasticity, as described in Section I.

In the latter case, developers should also be aware of the following Java classes, to be customized as needed:

- `SyblScaleOutAgent` is the class responsible to interface ReSpecT with rSYBL in the specific case of a `scaleout` request—`SyblScaleOutSpawnActivity` and `SyblScaleInAgent` are similar. It exploits ReSpecT coordination services to retrieve scaling out configurations – as given by ReSpecT specifications similar to those in Subsection IV-A –, then delegates rSYBL runtime elasticity-related operations, and finally informs elasticity controllers whether scaling out was successful or not. E.g., Fig. 6 sketches how configurations are retrieved through ReSpecT (lines 3-13) and how to synchronize with elasticity controllers (lines 17-25).
- `EnforcementAPI` is the class responsible for interfacing rSYBL with ReSpecT. As sketched in Fig. 7, it "parses" coordination-aware elastic operations in SYBL directives, then delegates them to ReSpecT runtime, and finally coordinates with it to collect operations outcome.
- `RespectEnforcementAPI` is the class defining the basic mechanisms provided to rSYBL to interact with ReSpecT coordination services. As depicted in Fig. 5 on page 6, method `delegate` is the only public method, responsible to parse the coordination operation requested and to act accordingly.

In summary, developers are required to know basics about TuCSoN and ReSpecT APIs as well as about APIs provided by our prototype integration. Nevertheless, they gain complete control over the coordination-aware elasticity enforcement

```
// how many scaling out tries?
ITucsonOperation op = this.acc.in(
    this.tid,
    new LogicTuple("scaleOutTries", new Var("Tries")),
    Long.MAX_VALUE);
tries = op.getLogicTupleResult().getArg(0).intValue();
// what time step between re-tries?
op = this.acc.in(
    this.tid,
    new LogicTuple("scaleOutTime", new Var("Time")),
    Long.MAX_VALUE);
step = op.getLogicTupleResult().getArg(0).intValue();
... // delegate rSYBL scaling then check success/failure
// put outcome in shared ReSpecT tuple centre
this.acc.out(
    this.tid,
    new LogicTuple("scaleOut",
        new Value(
            this.node.getId(),
            new Value("done", new Value(res))
        )
    ),
    Long.MAX_VALUE);
```

Fig. 6: Run-time adaptation of elasticity control through coordination.

```
if (actionName.contains("scaleout")) {
  // coordinated scale out requested
  ... // parse arguments
  this.doCoordinatedScaleOut(args, e);
} else if (actionName.startsWith("scalein")) {
  // coordinated scale in requested
  ... // parse arguments
  this.doCoordinatedScaleIn(args, e);
} else if (actionName.startsWith("monitorMetrics")) {
  // coordinated monitoring requested
  ... // parse arguments
  this.doCoordinatedMonitorMetrics(args, e);
} else {
  this.respect.delegate( // basic ReSpecT operation
    actionName,
    RespectEnforcementAPI.OP_TIMEOUT,
    e);
}
// [scaleout case]
// read outcome from shared ReSpecT tuple centre
op = this.respect.delegate(
    "in(scaleOut(" + node.getId() + "),
    done(B))",
    RespectEnforcementAPI.OP_TIMEOUT,
    node);
this.executingControlAction = false;
if (op.isResultSuccess()) {
  // scale out successful
} else {
  // scale out failed
}
```

Fig. 7: Coordination-aware elasticity primitives. The bridge component is responsible for decoupling elasticity controllers from controlled resources. Notice the synchronization point with listing Fig. 6: in lines 24-28 we consume the tuple put in lines 17-25 of Fig. 6.

process, being able to adjust the way in which elasticity mechanisms interact with coordination services.

C. Runtime Benefits

We now compare a coordination-aware elasticity controller (Pseudocode 2) to a coordination-unaware one (Pseudocode 1) in the case of a scale out process. Pseudocode 1 exhibits issues mentioned in Subsection I-A:

- line 3: elasticity controllers invoke the underlying cloud API through a number of chained calls (abstracted away to ease understanding). Thus, they are "locked-in" to such API: in case their syntax changes, it is sufficient to update calls accordingly, but in case of a semantics change, the whole elasticity enforcement process may require a fix.
- lines 4-8: if cloud API call is synchronous, elasticity controllers lose their control flow, being obliged to wait until the call returns. Thus: what if cloud API waits indefinitely for scale out to succeed? What if multiple tries are done in case of failure, preventing elasticity controllers to reply to new requests in the meanwhile? If cloud API call is asynchronous: how do elasticity controllers know if scale out was successful? Are they obliged to perform busy-waits, continuously monitoring the process? Safety, availability and efficiency are anyway compromised.
- line 12: in case something bad happens, elasticity controllers are responsible for recovery, fail-over, retry, etc., whichever is the failure-handling mechanism chosen. E.g., in case it retries, time in which control flow remains blocked, keeping elasticity controllers insensitive to new requests, keeps increasing.

Pseudocode 1 Coordination-unaware Elasticity Control

```
1:  procedure SCALEOUT(node1)                    ▷ elasticity control
2:      node2 ← cloudAPI.doScaleOut(node1)       ▷ API lock-in
3:      if [was synch call] then                 ▷ loss of control flow
4:      else if [was asynch call] then           ▷ undefined/busy wait
5:      end if
6:      if checkHealth(node2) then               ▷ scale out successful?
7:          startMonitoring(node2)
8:      else
9:          SCALEOUT(node1)                      ▷ retry
10:     end if
11: end procedure
```

In Pseudocode 2, above issues are solved:

- lines 3-4: scaling out is delegated to coordination services. Control flow is retained by elasticity controllers, now free from cloud API issues (e.g. invocation semantics) and no longer tied to the specific cloud provider. Coordination services are now locked-in, but this is fine, since in case of semantics change to cloud API the elasticity control process should not be affected, whereas lower-level mechanisms (e.g. coordination services) may be—e.g., synchronous vs. asynchronous calls should not influence elasticity control but coordination only.
- lines 6-14: scaling out call is now asynchronous *by default*. Furthermore, coordination guarantees replies in finite time (which can be set even at run-time). Availability as well as efficiency are increased, since elasticity control is free to process other incoming requests while waiting previous ones results, and safety is improved thanks to failures being confined to coordination services.
- lines 16-18: run-time configuration of parameters (e.g. waiting time, number of re-tries, etc.), some cloud API calls, coordination with monitoring services, synchronization issues and handling of operations outcome are all delegate to coordination services—because they are coordination issues.

As stated in Section I, delegation of duties between elasticity and coordination supports separation of concerns while enabling better design and management of elasticity control.

Pseudocode 2 Coordination-aware Elasticity Control

```
1: procedure SCALEOUT(n1)                          ▷ elasticity control
2:     coordServ ← getCoordinationService()
3:     coordServ.doScaleout(n1, tOut)                          ▷ asynch
4: end procedure
5: procedure ONSCALEOUTSUCCESS(n2)
6:     startMonitoring(n2)
7: end procedure
8: procedure ONSCALEOUTFAILURE(n1)
9:     SCALEOUT(n1)
10: end procedure
11: procedure ONSCALEOUTTIMEOUT(n1)
12:     SCALEOUT(n1)
13: end procedure
14: procedure DOSCALEOUT(n1, tOut)                          ▷ coordination
15:     ...          ▷ Read params, call cloud API, coordinate monitoring, wait completion, share result
16: end procedure
```

V. Related Work

Rajana et al. [14] coordinate message packets to prevent network congestion in cloud environments. Wang et al. [15] coordinate cloud providers and cloud consumers through a policy-based enforcement system. Wei et al. [16] coordinate resources provisioning and exploitation across multiple workflows, proposing an agent-based, message-based decentralized algorithm matching resource supply and demand. Besides goals of the works, differently from our approach no well-defined coordination model in the sense of [8] is adopted, thus solutions are ad-hoc tailored to the specific domain. Distler et al. [17] propose extendable coordination services, which enable users to dynamically register composite operations exploiting low-level coordination services API, as the means to achieve run-time adaptation of cloud-related coordination issues—e.g., storage quota management. Based on CometCloud tuple space -based coordination mechanism, Beach et al. [18] propose a broker-based matchmaking engine for integrating heterogeneous devices within a single (cloud) computing environment. Besides goals being different again, our focus is more on supporting service-level, general purpose, coordination-aware elasticity, providing service stakeholders (e.g., application and cloud-based infrastructures developers) with better experience in controlling their cloud services.

VI. Conclusions & Ongoing Work

While elasticity primitives simplify developers task in dealing with elasticity tradeoffs, elasticity languages and runtime systems should leverage coordination capabilities of cloud infrastructures to deal with complex scenarios by means of delegation and separation of concerns. In this paper, we analyzed elasticity and coordination models to propose a novel approach we called *coordination-aware elasticity*. We have shown how elasticity and coordination could be integrated w.r.t. concepts, languages and runtime systems. We also provided a prototype to demonstrate feasibility and benefits of our approach, based on SYBL and ReSpecT. Currently, we are working on the prototype to both expand its features set and carry out novel experiments across multiple cloud environments.

References

[1] S. Dustdar, Y. Guo, R. Han, B. Satzger, and H.-L. Truong, "Programming directives for elastic computing," *IEEE Internet Computing*, vol. 16, no. 6, 2012.

[2] D. Hassan and R. Hill, "A language based security approach for securing map-reduce computations in the cloud," in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, Dec 2013, pp. 307–308.

[3] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Sybl: An extensible language for controlling elasticity in cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 112–119.

[4] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994.

[5] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," in *The Engineering of Large Systems*, ser. Advances in Computers, M. V. Zelkowitz, Ed. Academic Press, 1998, vol. 46, pp. 329–400.

[6] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992.

[7] M. Schumacher, *Objective Coordination in Multi-Agent System Engineering. Design and Implementation*, ser. LNCS. Springer, Apr. 2001, vol. 2039.

[8] P. Ciancarini, "Coordination models and languages as software integrators," *ACM Computing Surveys*, vol. 28, no. 2, pp. 300–302, 1996.

[9] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. S. I. University), and N. Koziris, "Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, 2013, pp. 34–41.

[10] A. Almeida, F. Dantas, E. Cavalcante, and T. Batista, "A branch-and-bound algorithm for autonomic adaptation of multi-cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 315–323.

[11] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[12] A. Omicini and E. Denti, "Formal ReSpecT," *Electronic Notes in Theoretical Computer Science*, vol. 48, pp. 179–196, Jun. 2001, Declarative Programming – Selected Papers from AGP 2000.

[13] S. Mariani and A. Omicini, "Space-aware coordination in ReSpecT," in *From Objects to Agents*, ser. CEUR Workshop Proceedings, M. Baldoni, C. Baroglio, F. Bergenti, and A. Garro, Eds., vol. 1099. Turin, Italy: Sun SITE Central Europe, RWTH Aachen University, 2–3 Dec. 2013, pp. 1–7, xIV Workshop (WOA 2013). Workshop Notes.

[14] V. S. Rajanna, S. Shah, A. Jahagirdar, C. Lemoine, and K. Gopalan, "Xco: Explicit coordination to prevent network fabric congestion in cloud computing cluster platforms," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 252–263.

[15] M.-X. Wang, L. Xu, and C. Pahl, "A coordination protocol for user-customisable cloud policy monitoring," in *CLOSER 2013 - Proceedings of the 3rd International conference on Cloud Computing and Services Science*. SciTePress – Science and Technology Publications, 2013.

[16] Y. Wei and M. B. Blake, "Decentralized resource coordination across service workflows in a cloud environment," in *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '13)*. Washington, DC, USA: IEEE CS, 2013, pp. 15–20.

[17] T. Distler, F. Fischer, R. Kapitza, and S. Ling, "Enhancing coordination in cloud infrastructures with an extendable coordination service," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, ser. SDMCMM '12. New York, NY, USA: ACM, 2012, pp. 1:1–1:6.

[18] T. H. Beach, O. F. Rana, and N. J. Avis, "Integrating acceleration devices using CometCloud," in *Proceedings of the 1st ACM Workshop on Optimization Techniques for Resources Management in Clouds (ORMaCloud '13)*. New York, NY, USA: ACM, 2013, pp. 17–24.