Chapter 11

Assessing the Resilience of Self-Organizing Systems: A Quantitative Approach

Matteo Risoldi, Jose Luis Fernandez Marquez, and Giovanna Di Marzo Serugendo

Contents

As described in the previous chapter, self-organizing systems exhibit adaptation and resilience features, but the assessment and measurement of these features is not trivial, even if it would be very useful in order to quantify the adaptation and the resilience of different approaches and to compare systems. In this chapter, Matteo Risoldi, Jose Luis Fernandez Marquez, and Giovanna Di Marzo Serugendo propose a framework, called DREF, which aims to support the assessment and the measurement of these features in a quantitative manner. They use a case study to show the applicability of their proposal.

11.1 Introduction

Resilience, intended as the persistence of dependability when facing changes [1], is one of the central features of self-organizing (SO) systems due to their ability to adapt their behavior following changes and faults. The assessment of resilience is generally achieved with experiments and simulations. Robustness and adaptation to some changes is obtained through specific SO mechanisms, which have their limits and do not help overcoming any possible type of faults or change [2]. For instance, digital pheromone in ant-based systems helps overcome the appearance of obstacles in the environment or the disappearance of food but is of limited help in case of faults (malicious or not) in the agent behavior (e.g., not properly following the pheromone). Therefore, in the process of development of a SO system, a developer will often want to achieve better resilience by adding, removing, or modifying the system's behavior, then assessing the system to see whether and how it has improved. Due to the complex behaviors of SO systems, however, it is not easy to quantify how a new version of a SO system compares to the one it replaces. Informal methods of comparison are generally effective only for relatively simple and small-scale systems. As SO systems are often used to model large, complex behaviors, a structured, systematic, and repeatable way to compare the resilient properties of different versions of a system is necessary.

In this chapter, we show how the evolution process taking place during the development of a SO system can benefit from a quantification of the satisfaction of resilience-related properties by different versions of the system. To this end, we will assist the classical "trial and error" development process (i.e., varying parameters and performing simulations) with a formal framework for the quantification of resilience called DREF (*dependability and resilience engineering framework*) [3]. DREF provides a generic framework to organize resilience measures in a coherent, formally defined way that is flexible and customizable. It supports a systematic practice for assessing resilience and is applicable to any system of interest.

The main goal of this chapter is to show how a precise, quantitative definition of resilience measures helps the developer in the choice of a particular version of a system. We used DREF on a dynamic gradient case study with 20 nodes. To this end, we selected specific measurable requirements: (1) probability of message delivery,

(2) accuracy of the gradient, and (3) the number of messages created. We then measured quantitatively those requirements in a mobile scenario and assessed how modifying key parameters in subsequent versions of the system impact (positively or negatively) the resilience to this type of perturbation. Resilience is directly linked to the measures above assessed across the different executions. For instance, the higher the accuracy of the gradient, the higher the resilience of that version of the system to the mobility of nodes.

The chapter is organized as follows. Section 11.2 discusses how in SO systems literature there is a growing interest in formal methods. Section 11.3 describes the SAPERE framework used to create the case study described in Section 11.4. The latter section also discusses the simulation environment used to assess identified properties. Section 11.5 briefly introduces the notions of the DREF framework essential to this chapter and how they apply to the case study. In Section 11.6, we describe the resilience assessment results. Finally, Section 11.7 draws conclusions and outlines the perspectives of this work.

11.2 Formal Methods in SO Systems

The development of SO systems, and multi-agent systems in general, is a software-engineering activity like any other. The different paradigm made it so that, during a certain time, development practices were still not as mature as in other software-engineering fields. In 2000, Jennings argued [4] that one of the pitfalls of agent-oriented software engineering (AOSE) is that "You forget you are developing software: [...] the development of any agent system [is] a process of experimentation. [S]oftware engineering good practice [is] ignored."

Since then, there has been progress on this subject. In 2005, Bernon et al. [5] reviewed several AOSE methodologies and proposed a unified meta-model in an effort to standardize the conceptual framework behind multi-agent systems (MAS). More advanced techniques, such as model-driven engineering, and formal methods slowly but surely made their way into AOSE. A 2009 survey [6], catalogued about 50 works from the previous 10 years that are concerned with the application of formal methods to AOSE. A large number among them are concerned with validation and verification of MAS. Several of them use a model-driven engineering approach. In addition to novel approaches to simulation-based verification [7], there have been more rigorous approaches to verify MAS using model checking with an accent on modeling [8], model transformation [9], and requirements enrichment with business constraints [10].

Current methodologies specifically meant for developing SO systems [11] follow the typical phases of software-engineering methodologies: requirements, analysis, design, implementation, verification, and test. Those methodologies all focus on the design phase and, in some cases, on a series of design phases [12], in which the self-organizing behavior is progressively modeled and, in some methodologies, also simulated. Some of them consider analysis of faults [13], but none of them specifically addresses assessment or quantification of resilience.

The cited works are a strong contribution to bringing AOSE and SO development methods on par with other software-engineering domains but are mostly oriented to the requirements and design stages of software development. In the domain of SO systems, however, validation and verification methods should deal with the dynamic changes in modes and contexts and address runtime assurance as stated in [14]. The authors of [14] also agree that requirements for self-adapting systems should be expressed in a way that accounts for the uncertainty characterizing this kind of systems. They argue that the very same nature of self-adapting systems calls for models and development practices that blur the line between design-time and runtime with models being "built and maintained at runtime" with evolutions of the system being reflected at the model level. The DREF framework was built, in part, with the goal of integrating requirements, assurance, and system evolutions in the model itself. DREF is thus a suitable candidate to consider for assessing resilience of SO systems as it considers both runtime aspects and design evolutions.

11.3 The SAPERE Framework

In this section, we present the SAPERE framework, used for implementing the case study presented in this chapter. The SAPERE framework has been developed under the SAPERE European Project.* SAPERE provides a theoretical and practical framework for decentralized deployment and execution of self-aware and adaptive services for future and emerging pervasive network scenarios.

The SAPERE framework combines two different technologies: (1) tuple space systems, providing a shared space in which agent and services are advertised, and (2) chemical-inspired systems, providing chemical rules (called *eco-laws*) that govern the behavior of the SAPERE system [15,16].

An innovative architecture is built on top of these two technologies. The main motivation is to allow applications to rely on core services (e.g., spreading, gradient, or evaporation) provided by the environment or by specific libraries, building spatial structures across distributed (possibly mobile) nodes [17]. This architecture is organized into different layers; it provides abstractions that ease the design and implementation of new decentralized applications by relaying and reusing functionalities provided by those core services. Chapter 9 discusses the corresponding architectural style [18].

In the SAPERE implementation the main concepts are the following:

- "Live Semantic Annotation" (LSA): A tuple that represents any information about an agent or service. LSAs are injected in the LSA's space representing the (updated) state of their associated component.
- LSA's space: A distributed, shared space in which context and information are provided by a set of LSAs stored at given locations.

* http://www.sapere-project.ev/.

- Eco-laws: Rules acting as chemical reactions that implement core services. These rules act on the LSAs stored in each LSA's space by deleting, updating, or moving LSAs between LSA's spaces. In the same way that a biological system's processes obey nature's laws, a SAPERE application is subject to eco-laws that implement its active environment.
- LSA bonding: An LSA bond acts as a reference to another LSA and provides fine-tuned control of what is visible or modifiable to each agent and what is not. If an LSA of a given agent includes a bond to an LSA of another agent, the former agent can inspect the state and interface of that other agent (through the bond) and act accordingly. Bonds are a specific type of eco-laws.
- Agents: Execute in hosts and are able to locally access the LSA's space available in that host.

Eco-laws reside in the LSA's space and are implicitly and seamlessly triggered by tags or properties appearing in the LSAs. An LSA can be subject to different eco-laws depending on the specified properties or tags of the tuple. Core services provided by the eco-laws include spread, decay, aggregate, and bond. Higher-level services, built on top of those core services, are provided either by libraries or specialized agents. They include services for building or exploiting spatial structures, such as Chemotaxis or dynamic gradient, and application-level services, such as crowd steering.

The SAPERE infrastructure encompasses a middleware [19] running on stationary and mobile platforms, situation-awareness [20], semantic resource discovery [21], and crowd-steering applications [16].

11.4 Case Study: Dynamic Gradient

This section describes the dynamic gradient service, designed and implemented using the SAPERE framework.

Gradients are spatial structures spread across nodes that provide information about distance from and direction to the gradient source. Thus, when a piece of information is propagated among nodes under the form of a gradient, additional information about the sender distance and direction is progressively computed at each node.

Figure 11.1 shows an example of gradient spread over a network of 12 nodes. The nodes represent devices with computational and connectivity capabilities (e.g., laptops, smartphones, or tablets). The edges represent the connections between nodes; two nodes are connected if they are in communication range. In this example, the node with value 0 (source of the gradient) creates a gradient that spreads across the network. The value inside each node increases at each hop. When two or more gradient paths reach a node, the lowest gradient value is kept. The number of hops provides information about the distance. The node within communication range with the lowest hop count provides the direction to the gradient source.

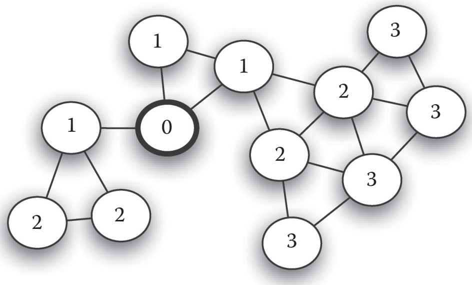


Figure 11.1 Gradient example.

This section focuses on *dynamic gradients*. A dynamic gradient provides information about the sender's (i.e., gradient source) distance and direction that is *periodically updated*. A dynamic gradient allows the maintainance (i.e., to update) of the gradient structure even when nodes are moving.

Dynamic gradients may be used in a wide range of applications, such as routing information in mobile ad hoc networks, crowd steering, or coordination in multi-agent systems.

11.4.1 Dynamic Gradient Implementation

The dynamic gradient service is provided by the SAPERE framework as an external library and is available in each node. The dynamic gradient service makes use of the spread, aggregate, and decay eco-laws.

Spread allows the information to be spread across the network, increasing hop counts at each node. Aggregate is in charge of selecting the value with the minimum number of hops when information is coming through different gradient paths. The combination of spread and aggregate eco-laws allow the creation of a gradient across the network, but the gradient is not updated; thus, the information about the sender's distance and direction becomes outdated when the nodes are moving. To avoid this problem, the dynamic gradient relies on the decay eco-law to periodically decrease the lifetime of the gradient LSAs. When the lifetime value reaches 0, the SAPERE middleware removes the gradient information LSA from the node. If the gradient information LSA is removed from the node that created the dynamic gradient (i.e., the gradient source), the gradient LSA is injected again in the system by the dynamic gradient service (library agent). This actually restarts a new gradient propagation thus providing updated information about the sender's distance and direction.

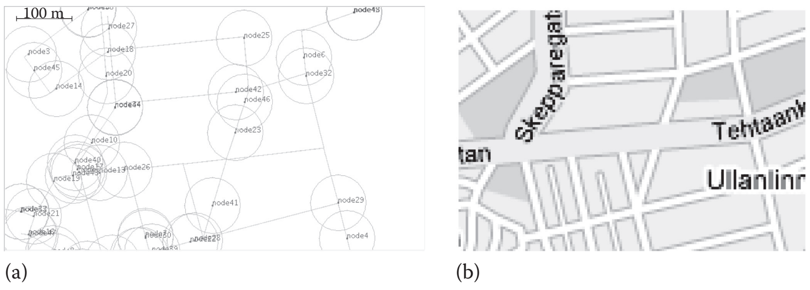
Eco-laws fire periodically, for example, sending information from one node to another (spreading) and aggregating them. The *frequency of eco-laws firing* is a key parameter that has an effect on the performance of the system. The decay eco-law is also involved in the dynamic gradient and fires at the same frequency as the other eco-laws.

A second key parameter in the case of the dynamic gradient service is the *decay value* (i.e., the lifetime) of the gradient information. Intuitively a short lifetime favors a fast adaptation but involves huge resource consumption (i.e., mainly bandwidth), and a long lifetime saves resources but does not cope with topological network changes. This parameter is exploited in the decay eco-law.

In this case study, we analyze the evolution process of setting these two parameters involved in the dynamic gradient service. The experimental values in the evolution process are assessed using the ONE simulator [22] introduced in the next section.

Figure 11.2a shows the map of Helsinki we have used for the case study with 20 SAPERE nodes. Figure 11.2b represents the real geographical map used in the simulations. We used a part of Helsinki city center (600 × 500 meters). We set the number of moving nodes to 20. Nodes simulate people (and their devices) walking on the streets. We used the mobility pattern called "shortest path based on map." Each node chooses a random destination, and it reaches the desired destination by following the shortest path on the map (following roads). The speed of the nodes is randomly chosen for each trajectory between 1–3 m/s. Each node has a wi-fi device with 200 m of communications range, and the transmission speed is 250 KBps. The simulation runs for 900 simulation steps, and each simulation step corresponds to 1 *s* of simulated time.

Figure 11.3a shows the simulation scenario before the gradient creation; black lines represent connections among nodes. Figure 11.3b depicts the gradient (black lines) corresponding to the initial configuration. The black circle node is the gradient source (i.e., the node that initially creates the gradient). We observe two nodes on the right that are disconnected. Figures 11.3c and 11.3d show subsequent updates of the gradient once the nodes start moving. Notice that for clarity reasons, Figures 11.3b, 11.3c, and 11.3d do not show the communication links representing the physical connections between the nodes and do not show the circles representing the communication range. They only show the gradient links.



(a) (b)

Figure 11.2 Scenario map: Helsinki area with SAPERE nodes (a), real geographical map (b).

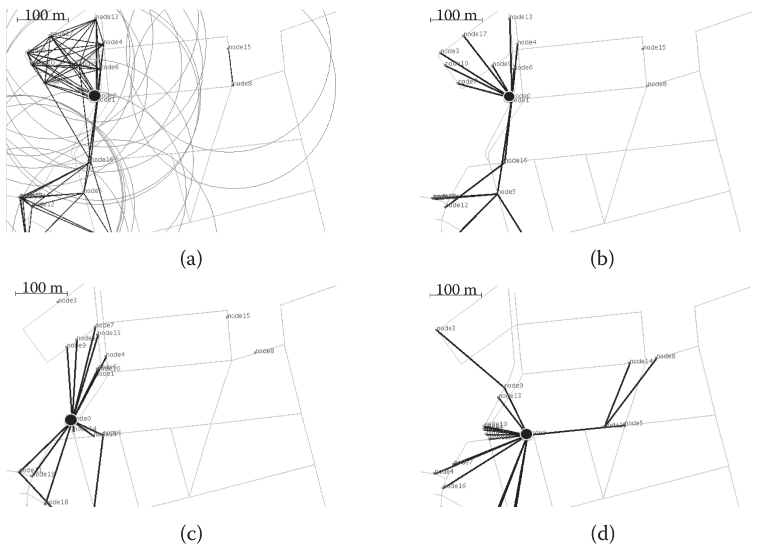


Figure 11.3 Simulation scenario before gradient creation (a), after gradient creation (b), once nodes have moved (c) and (d).

11.4.2 The ONE Simulator

The case study described above has been implemented using The ONE simulator [22], an open source network simulator providing a large range of realistic scenarios. Among others, it allows the investigation of simulations composed of many different entities, such as pedestrians, cars, buses, and trams, to import real mobility traces, to visualize the movement of nodes on Google maps, and to provide realistic network-related simulations of bandwidth consumptions, time to send information among nodes, or packet collisions.

We extended the ONE with the actual SAPERE middleware [19]. Consequently, the dynamic gradient is implemented using the SAPERE middleware (libraries, eco-laws, LSAs) running in the nodes, and the movement of nodes, visualization, performance metrics, and connectivity between the nodes are done by the ONE simulator.

Mainly, the ONE allows us to do the following:

- Simulate network connections, providing metrics about the network behavior (e.g., message drops, messages properly received or bandwidth consumption).
- Provide different network interfaces to the entities participating in the system, such as wi-fi or Bluetooth. This feature allows the creation of realistic heterogeneous networks.

- Use a library with many different mobility patterns.
- Execute the simulation with visualization or in batch mode, making it possible to use servers for very large-scale simulations.

11.5 The DREF Framework

DREF [3] is a framework that formalizes the fundamental concepts used to define dependability and resilience of ICT systems. It allows quantifying how the level of satisfiability of system properties varies over an evolution axis when facing changes. DREF formalizes a rather large set of core concepts related to the assessment of resilience. While the complete definitions can be found in [3], we will summarize here the ones that are useful in the context of this chapter:

Entities and properties: An *entity* is anything of interest that is considered. An entity could be, for example, a program, a database, a person, a hardware device, or a development process. In this chapter, we consider one entity *Dyn*, a dynamic gradient system:

$$Ent = \{Dyn\}. \tag{11.1}$$

A *property* is a basic concept used to characterize an entity. It can be, for example, an informal requirement or a mathematical property. In this chapter, we consider three relevant properties of the dynamic gradient system, namely, the probability of message delivery *dp*, the average accuracy of the gradient *ac* (i.e., the distance of the actual gradient from the ideal gradient that would exist if nodes were not moving), and the number of messages created *nm*:

$$Prop = \{dp,ac,nm\} \tag{11.2}$$

Evolution axis: An evolution axis is a set of values that is used to index a set of entities or a set of properties. In the context of this chapter, the development of *Dyn* went through different versions, which are indexed as Dyn_i^d, where *i* is the eco-law firing interval, and *d* is the decay value (i.e., lifetime of gradient information)—the two key parameters discussed in Section 11.4.1. We variated *i* between 1 and 3, and *d* between 2 and 12 in two-step intervals. Thus, there are two evolution axes indexed as

$$ev_i = \{1, 2, 3\}; \tag{11.3}$$

$$ev_d = \{2, 4, 6, 8, 10\} \tag{11.4}$$

Satisfiability: An entity will generally have to satisfy some property or requirement. This fact can be expressed with a *satisfiability function*, defined as follows. Let *Ent* be a set of entities and *Prop* a set of properties. The satisfiability of properties by entities is a function *sat* such that

$$sat : Prop \times Ent \rightarrow \mathbb{R} \cup \{\bot\} \tag{11.5}$$

Semantically, *sat* quantifies how much an entity satisfies (or not) a property and is defined depending on the application.

The satisfiability functions for the properties in our case study are defined as follows. The *dp* property is measured in terms of the ratio between the messages that are delivered and created:

$$sat(dp, e) = \frac{deliveredmessages}{createdmessages}. \tag{11.6}$$

The *ac* property is measured as the average accuracy (i.e., percentage of nodes with a correct value) of the gradient in the simulation:

$$sat(ac, e) = \frac{\sum_{i=1}^{n} ac_i}{n} \tag{11.7}$$

where $ac_i = \frac{k_i}{20}$ is the accuracy of the gradient at step i of the simulation, k_i is the number of nodes for which the gradient value is correct, 20 is the number of nodes, and n is the total number of simulation steps.

The *nm* property is measured as

$$sat(nm, e) = \frac{20{,}000}{createdmessages} \tag{11.8}$$

where 20,000 is an empirically established ideal value for the number of created messages.

Nominal satisfiability: An additional relevant concept defined by DREF is that of *nominal satisfiability*, denoted as *nsat*, such that

$$nsat : Prop \times Ent \rightarrow \mathbb{R} \tag{11.9}$$

This is a satisfiability function used to represent the *expected satisfiability*—what minimum satisfiability value is considered to be acceptable.

The nominal satisfactions for the properties in our case study are defined as follows:

$$nsat(dp,e) = 0.99 \tag{11.10}$$

$$nsat(ac,e) = 0.8 \tag{11.11}$$

$$nsat(nm,e) = 1 \tag{11.12}$$

(i.e., the *nm* property is exactly satisfied if a system creates 20,000 messages, over-satisfied if it creates less, and under-satisfied if it creates more).

Failure: Given *sat*, a satisfiability function, and *nsat*, a nominal satisfiability function, we say there is a failure for a tuple $(p, e) \in dom(sat)$ if $sat(p,e) < nsat(p,e)$. In other words, an entity fails a property if its satisfiability is lower than the nominal satisfiability.

Let *e* be an entity, *p* a property; a failure of property *p* by entity *e* is denoted *fail*(*p,e*). It is defined as follows:

$$fail : Prop \times Ent \rightarrow \mathbb{R}_0^+ \tag{11.13}$$

$$fail\,(p,e) = nsat(p,e) - Min\,(nsat\,(p,e), sat\,(p,e)) \tag{11.14}$$

and it quantifies how far the actual satisfiability is from satisfying the property.

Cumulative failure level: The cumulative failure level for an entity is the sum of failures for *all* of its properties. The cumulative failure level for an entity *e* is denoted $sfail_e$ and is defined as

$$sfail_e = \sum_{p \in Prop} fail(p,e)\ (0\ if\ \ Prop = \varnothing) \tag{11.15}$$

11.6 Experimental Results

We took the case study system through two rounds of evolution, which will explore some possible variations of the two parameters of the system. The parameters we chose are the eco-law firing interval and the decay value (i.e., the lifetime of the information in the nodes). The eco-law firing interval parameter is of major importance as it can cause network overload if its value is too high, and conversely, it can prevent the gradient from updating properly if the value is too low. The decay value is similarly important as a too-low value would prevent the information from surviving long enough to form a gradient, and a too-high value would hamper gradient updates.

As explained in the previous sections, we modeled the case study using the SAPERE framework, implemented it in the ONE simulator, and used DREF to process the results and extract useful quantitative measures about property satisfaction.

11.6.1 First Evolution: Fixing an Eco-Law Firing Interval

In the first evolution in the development of our dynamic gradient system, we want to search for an appropriate eco-law firing interval. What we are looking for is an interval that is sufficient to let the gradient establish itself while not overloading the network. In this iteration, we will stop at the first value that achieves good accuracy, leaving further optimization of the number of messages for a further iteration.

We fix the decay value at 10, which, given the size of the network, is an educated guess at a value high enough to let the information propagate through the network. We then start by considering an eco-law firing interval of 1 s, increasing it one further second for each new version (and simulation). Table 11.1 shows the satisfaction values for the first three versions of *Dyn* that we consider: Dyn_1^{10}, Dyn_2^{10} and Dyn_3^{10}. We see that for all three properties there is a constant improvement (i.e., a reduction of the failure level *fail*(*p,e*)) for each individual property. As a consequence, the cumulative failure level $sfail_e$ decreases with each new version, indicating that it is an improvement over the previous one. What really matters here is the *fail*(*p,e*) for properties *dp* and *ac*. As soon as we find a failure level of 0 for both (i.e, in version Dyn_3^{10}), we can stop this first evolution.

Table 11.1 First Evolution: Varying the Eco-Law Firing Interval

e	*p*	*sat(p,e)*	*nsat(p,e)*	*fail(p,e)*	$sfail_e$
Dyn_1^{10}	dp	0.8380	0.99	0.152	1.1201
	ac	0.5104	0.80	0.2896	
	nm	0.3215	1	0.6785	
Dyn_2^{10}	dp	0.9649	0.99	0.0251	0.61
	ac	0.7241	0.80	0.0759	
	nm	0.4910	1	0.5090	
Dyn_3^{10}	dp	0.9981	0.99	0	0.1462
	ac	0.8144	0.80	0	
	nm	0.8538	1	0.1462	

11.6.2 Second Evolution: Fixing a Decay Value

In this second evolution, we want to search for an appropriate decay value. Fixing the eco-law firing interval at 3 s, based on the results of the first iteration, we will simulate the system for decay values ranging from 2 to 12, in two-unit steps.

Table 11.2 shows the satisfaction values for the obtained versions of *Dyn*: Dyn_3^2, Dyn_3^4, Dyn_3^6, Dyn_3^8, Dyn_3^{10}, and Dyn_3^{12}. The measures here follow a slightly more interesting dynamic. The *dp* property is constantly satisfied by all versions, albeit with some oscillations in values. The *ac* property is deeply under-satisfied by Dyn_3^2, then improves with versions until reaching *nsat* with Dyn_3^6 (even staying above

Table 11.2 Second Evolution: Varying the Decay Value

e	*p*	*sat(p,e)*	*nsat(p,e)*	*fail(p,e)*	*sfail$_e$*
Dyn_3^2	dp	1	0.99	0	0.7473
	ac	0.0527	0.80	0.7473	
	nm	32.4	1	0	
Dyn_3^4	dp	0.9984	0.99	0	0.1287
	ac	0.6713	0.80	0.1287	
	nm	2.2629	1	0	
Dyn_3^6	dp	0.9985	0.99	0	0
	ac	0.8338	0.80	0	
	nm	1.2138	1	0	
Dyn_3^8	dp	0.9934	0.99	0	0.0575
	ac	0.8085	0.80	0	
	nm	0.9425	1	0.0575	
Dyn_3^{10}	dp	0.9981	0.99	0	0.1462
	ac	0.8144	0.80	0	
	nm	0.8538	1	0.1462	
Dyn_3^{12}	dp	0.9965	0.99	0	0.2093
	ac	0.8052v	0.80	0	
	nm	0.7907	1	0.2093	

nsat). The *nm* property instead follows a degradation pattern with the increase of decay values, going from satisfaction in the first three versions, to a gradually deeper under-satisfaction in the following ones. There is one version, Dyn_3^6, that satisfies all three properties. We can thus conclude that, in this iteration, we established an appropriate decay value to be 6.

11.6.3 Cumulative Failure Level as a Useful Overall Assessment of Versions

It is rather evident that just by using the satisfiability *sat*(*p,e*), shown in Figure 11.4,* it is not immediate to the eye of the developer whether a version is better than another at satisfying properties. Even by looking at the individual failure levels *fail*(*p,e*) for each property, shown in Figure 11.5, the assessment may not be simple as the failure values series may vary independently from one version to another.† This is the main reason for the existence of the cumulative failure level indicator $sfail_e$: It gives a concise quantification of how much a version is failing. The hypothetical developer behind our example would easily be able to plot the values for all the created versions and quickly spot the lower points that identify the best versions (shown in Figure 11.6).

Figure 11.7 shows how the gradient continuously recovers from the mobility of nodes for the case of Dyn_3^6. The red line shows the evolution of ac_i. The blue line shows the average of ac_i from time 1 to time *i* (note that at time 900, it is equal to $sat(ac, Dyn_3^6)$).

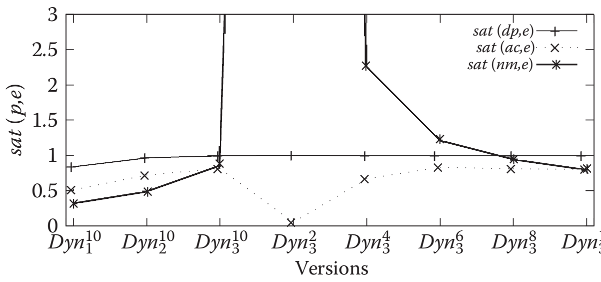


Figure 11.4 ***sat*(*p,e*) levels for each property versus evolution for versioned entity *Dyn* (n.b.: $sat\left(nm, Dyn_3^2\right) = 32.4$ is off scale for readability).**

* In this figure, as well as in Figures 11.5 and 11.6, the entity Dyn_3^{10} only appears once, although it is part of both evolutions.

† Admittedly, in this case study, there is a version that stands out for having all three failure levels at 0, that is, Dyn_3^6; however, this is far from being the general case.

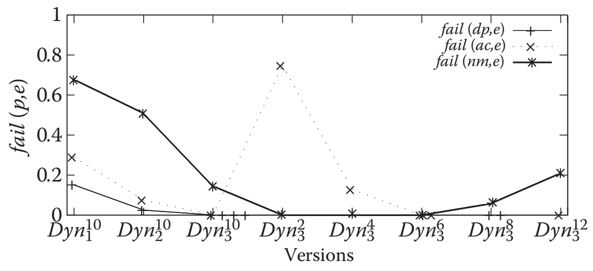


Figure 11.5 ***fail*(*p,e*) levels for each property versus evolution for versioned entity *Dyn*.**

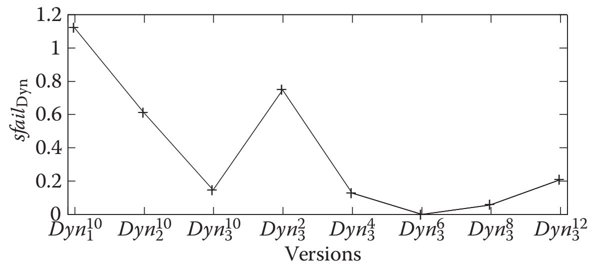


Figure 11.6 Cumulative failure level versus evolution for versioned entity *Dyn*.

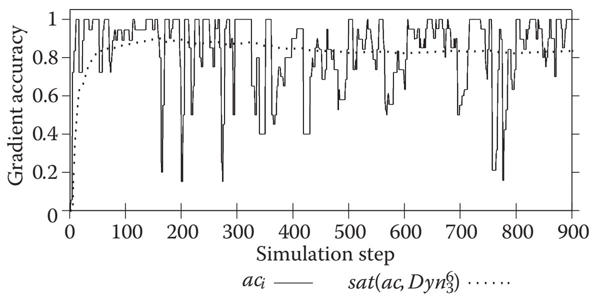


Figure 11.7 Gradient accuracy, Dyn_3^6.

11.6.4 Possible Further Evolutions

We stopped this example with the second evolution for the sake of brevity. However, it is easy to imagine that there could be a third evolution to answer the question of whether a further exploration of larger eco-law firing intervals could further

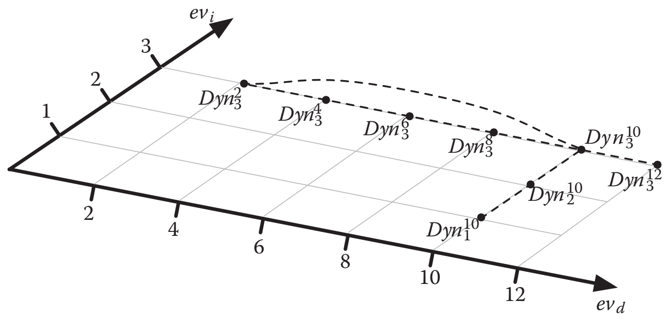


Figure 11.8 Evolution of our example over two coordinated axes.

improve the performance of the system. In this way, we could picture that the evolution of the system in our example, which is occurring over two coordinated axes (shown in Figure 11.8), could continue by adding further versions indexed on the axis (possibly adding more indexes to the axes or even additional axes). Also, instead of a manually defined evolution path like the one we followed, one could perform a complete exploration of all combinations of parameter values encompassed by given bounds if the available resources allowed for it.

11.6.5 Tolerance

In this example, we considered that a property is either satisfied (i.e., $sat(p,e) \geq nsat(p,e)$) or not. However, in DREF, it is possible to define a tolerance threshold for a property, that is, a level smaller than $nsat(p,e)$ that indicates a satisfaction value that does not quite satisfy a property but is still within acceptable limits. This is a particularly relevant concept for resilient systems as it can be used to describe tolerated failures (whether foreseen or out of the envelope). In our example, a tolerance threshold of 0.85 could be introduced for, for example, property *nm*, which would result in versions Dyn_3^8 and Dyn_3^{10} to be also considered satisfactory.

11.7 Conclusion and Perspectives

This chapter showed how the DREF framework can be used to quantitatively assess the resilience of SO systems during their development process. Through a case study, based on the SAPERE framework, we showed how the indicators defined in DREF can offer a useful synthesis of simulation results in order to compare and choose among different versions of a system.

DREF also defines other indicators, not discussed here but found in [3], that bring other useful information to the table. This chapter only focused on failures, allowing one to choose the version that "fails less." However, in other cases, it might be interesting to choose the version that "satisfies more" among several versions that are all correct. In this respect, DREF defines, for example, the concept of *global satisfiability*, which is a unique indicator measuring the cumulative level of satisfaction of *all* properties of an entity, possibly taking into account different property weights when applicable.

One of the main advantages of DREF is that we can use the cumulative failure level in order to easily compare different versions. Using the cumulative failure level, one could implement an evolutionary algorithm, such as a genetic algorithm, in order to automate the creation and selection of different versions. Thus, the system would start with random versions that would evolve through the execution of the evolutionary algorithm until the cumulative failure level of one of the versions is below a desired value (i.e., the cumulative failure level would be used as the evolutionary fitness function in case of using a genetic algorithm).

Our future perspectives for this study include the integration of DREF in the ONE simulator in order to systematically apply quantitative assessments of resilience in simulated systems. In a first instance, this will assist us in systematically investigating resilience of classified self-organizing mechanisms [23] against a range of faults and perturbations. In the long term, in conjunction with identified metrics, this will constitute a definite step toward the establishment of benchmarks and assessments for self-* systems.

Acknowledgments

M. Risoldi is supported by the Fonds National de la Recherche, Luxembourg, project MOVERE C09/IS/02. G. Di Marzo Serugendo and J. L. Fernandez Marquez are partially supported by the EU FP7 project "SAPERE - Self-Aware Pervasive Service Ecosystems" under contract No. 256873.

References

1. J.-C. Laprie, "From dependability to resilience," in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Anchorage (USA), 2008.
2. G. Di Marzo Serugendo, "Robustness and dependability of self-organising systems – A safety engineering perspective," in *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems(SSS)*, ser. LNCS, vol. 5873. Lyon, France: Springer, Berlin Heidelberg, 2009, pp. 254–268.
3. N. Guelfi, "A formal framework for dependability and resilience from a software engineering perspective," *Central European Journal of Computer Science*, vol. 1, pp. 294–328, 2011, 10.2478/s13537-011-0025-x.
4. N. R. Jennings and M. Wooldridge, "Agent-oriented software engineering," *Artificial Intelligence*, vol. 117, pp. 277–296, 2000.

5. C. Bernon, M. Cossentino, and J. Pavón, "Agent-oriented software engineering," *Knowl. Eng. Rev.*, vol. 20, no. 2, pp. 99–116, Jun. 2005.
6. A. E. Fallah-Seghrouchni, J. J. Gomez-Sanz, and M. P. Singh, "Formal methods in agent-oriented software engineering," in *Proceedings of the 10th International Conference on Agent-Oriented Software Engineering*, ser. AOSE '10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 213–228.
7. M. Niazi, A. Hussain, and M. Kolberg, "Verification & validation of agent based simulations using the VOMAS (virtual overlay multi-agent system) approach," in *MALLOW*, ser. CEUR Workshop Proceedings, M. Baldoni et al., Eds., vol. 494. CEUR-WS.org, 2009.
8. J. Barjis, I. Rychkova, and L. Yilmaz, "Modeling and simulation driven software development," in *Proceedings of the 2011 Emerging M&S Applications in Industry and Academia Symposium*, ser. EAIA '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 4–10.
9. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Verifying multi-agent programs by model checking," *Autonomous Agents and Multi-Agent Systems*, vol. 12, no. 2, pp. 239–256, Mar. 2006.
10. M. Montali, P. Torroni, N. Zannone, P. Mello, and V. Bryl, "Engineering and verifying agent-oriented requirements augmented by business constraints with b-tropos," *Autonomous Agents and Multi-Agent Systems*, vol. 23, pp. 193–223, 2011, 10.1007/s10458-010-9135-4.
11. M. Puviani, G. Di Marzo Serugendo, R. Frei, and G. Cabri, "A method fragments approach to methodologies for engineering self-organizing systems," *ACM Trans. Autonomous and Adaptive System*, vol. 7, no. 3, pp. 33:1–33:25, Oct. 2012.
12. M. Schut, "On model design for simulation of collective intelligence," *Information Sciences*, vol. 180, pp. 132–155, 2010.
13. G. Di Marzo Serugendo, J. Fitzgerald, and A. Romanovsky, "MetaSelf – An architecture and development method for dependable self-* systems," in *Symp. on Applied Computing (SAC)*, Sion, Switzerland, 2010, pp. 457–461.
14. B. Cheng et al., "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin/Heidelberg, 2009, vol. 5525, pp. 1–26, 10.1007/978-3-642-02161-9-1.
15. F. Zambonelli et al., "Self-aware pervasive service ecosystems," *Procedia Computer Science*, vol. 7, pp. 197–199, 2011.
16. M. Viroli, D. Pianini, S. Montagna, and G. Stevenson, "Pervasive ecosystems: A coordination model based on semantic chemistry," in *27th ACM Symp. on Applied Computing (SAC 2012)*, S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, Eds. Riva del Garda, TN, Italy: ACM, 26–30 March 2012.
17. J. L. Fernandez-Marquez, G. Di Marzo Serugendo, and S. Montagna, "Bio-core: Bio-inspired self-organising mechanisms core," in *Bio-Inspired Models of Networks, Information, and Computing Systems*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, E. Hart, J. Timmis, P. Mitchell, T. Nakamo, and F. Dabiri, Eds. Springer Berlin Heidelberg, 2012, vol. 103, pp. 59–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32711-7_5.
18. J. L. Fernandez-Marquez, G. Di Marzo Serugendo, P. Snyder, and G. Valetto, "Pattern-based architectural style for self-organizing software systems," in *Adaptive, Dynamic, and Resilient Systems*, G. Cabri and N. Suri, Eds. Taylor & Francis, 2013.

19. F. Zambonelli, G. Castelli, M. Mamei, and A. Rosi, "Integrating pervasive middleware with social networks in SAPERE," in *Mobile and Wireless Networking (iCOST), 2011 International Conference on Selected Topics in*, Oct. 2011, pp. 145–150.
20. G. Stevenson, J. L. Fernandez-Marquez, S. Montagna, A. Rosi, J. Ye, A.-E. Tchao, S. Dobson, G. Di Marzo Serugendo, and M. Viroli, "Towards situated awareness in urban networks: A bio-inspired approach," in *First International Workshop on Adaptive Service Ecosystems: Nature and Socially Inspired Solutions (ASENSIS) at Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO12)*. IEEE Computer Society, 2012.
21. J. Stevenson, G. Ye, S. Dobson, M. Viroli, and S. Montagna, "Self-organising semantic resource discovery for pervasive systems," in *First International Workshop on Adaptive Service Ecosystems: Nature and Socially Inspired Solutions (ASENSIS) at Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO12)*. IEEE Computer Society, 2012.
22. A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE Simulator for DTN Protocol Evaluation," in *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. New York: ICST, 2009.
23. J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: A complete overview," *Natural Computing*, pp. 1–25, 2012.