# An Approach for Isolated Testing of Self-Organization Algorithms

Benedikt Eberhardinger, Gerrit Anders, Hella Seebach, Florian Siefert, Alexander Knapp, and Wolfgang Reif

Institute for Software & Systems Engineering, University of Augsburg, Germany
{eberhardinger, anders, seebach, siefert, knapp, reif}@isse.de

**Abstract.** We provide a systematic approach for testing self-organization (SO) algorithms. The main challenges for such a testing domain are the strongly ramified state space, the possible error masking, the interleaving of mechanisms, and the oracle problem resulting from the main characteristics of SO algorithms: their inherent non-deterministic behavior on the one hand, and their dynamic environment on the other. A key to success for our SO algorithm testing framework is automation, since it is rarely possible to cope with the ramified state space manually. The test automation is based on a model-based testing approach where probabilistic environment profiles are used to derive test cases that are performed and evaluated on isolated SO algorithms. Besides isolation, we are able to achieve representative test results with respect to a specific application. For illustration purposes, we apply the concepts of our framework to partitioning-based SO algorithms and provide an evaluation in the context of an existing smart-grid application.

**Keywords:** Self-organizing Systems, Adaptive Systems, Self-organization Algorithms, Software Engineering, Quality Assurance, Software Test

## 1 Introduction

The established quality assurance measure of testing aims at systematically covering possible paths through the system with the goal of finding failures. In this paper we provide a systematic approach for testing self-organization (SO) algorithms. This contribution is an elaboration and extension of the first vision of a framework for testing SO algorithms shown in [12] and is integrated into our overall research road map aiming at testing self-organizing, adaptive systems (SOAS); the vision of this overall approach is outlined in [14].

The properties of the SO algorithms like inherent non-deterministic behavior, an ever-changing environment, a high number of interacting components, and interleaving operations make it hard to achieve a systematic testing approach for SO algorithms. A key to success is automation, since manually it is rarely possible to cope with the high number of demanded test cases (that are necessary as a result of the huge state space). However, the automation of testing SO algorithms is faced by the complexity of the system class requiring techniques to cope with the following key challenges:

***C-ErrorMask*** SO algorithms—and in general SOAS—are designed to be robust and flexible under ever-changing environmental conditions. As a side-effect of these properties an SO algorithm itself, other SO algorithms, or adaptation mechanisms might cover the tracks of possible failures that should be revealed during testing the SO algorithms. Thus, faulty behavior of one SO algorithm could be compensated by another mechanism masking the failure.
For instance, assume an erroneous SO algorithm that returns wrong or inappropriate system structures as a result to the controlled components. This fault then could be masked by an adaption mechanism of the components that compensates the wrong system structure by a high and costly amount of adaptation. The SO algorithm would encompass a fault, but no failure is visible at first glance since the adaptation mechanism masks it by keeping the system alive.

***C-Isolate*** SO algorithms are based on interaction with the system's components. In the majority of cases, several different SO algorithms are incorporated; their overlap of interaction with the components leads to so-called interleaved feedback loops. These are challenging in testing, because it is hard to get dedicated results for a single SO algorithm. To address this challenge, there is a need for isolated testing of single SO algorithms.
As an example of this challenge, assume an SO algorithm that forms organizational structures, e.g., by partitioning the system's components. A further algorithm, however, performs its calculations for parametrization of the components based on this structure, e.g., for forming an evenly distributed output for each partition. These two algorithms are interleaved and a dedicated test result for the algorithm performing its calculation on the structure is only possible if they are isolated, since the results depend on the results of the first and vice versa. However, this isolation is hard to achieve due to the high dependencies between the two algorithms.

***C-Oracle*** The oracle problem is a well-known challenge for all testing endeavors [9,23]. However, the properties of SO algorithms increase this problem: For classical testing the conditions of execution for the system under test (SuT) as well as the concrete requirements are known. Let us call these facts the "known-knowns".[1] For SO algorithms under test (SOuT) we know that there are unknown conditions of execution where we can hardly decide a priori, i.e., at design-time, whether a state is correct or not; we call these conditions the "known-unknowns". Moreover, for the SOuT there might even be situations we are not aware of at all, which we call the "unknown-unknowns". An oracle that is capable of evaluating the test results of SO algorithm at least has to be able to handle the "known-unknowns".
For an instance of "known-unknowns" consider a smart energy grid setting (which is outlined in more detail in Sect. 2) where different power plants are self-organized in different so-called autonomous virtual power plants. If weather-dependent power plant are include, the SO here will depend on the weather con-

[1] The classification of known-knowns, known-unknowns, and unknown-unknowns is borrowed from United States Secretary of Defense Donald Rumsfeld's response given to a question at a U.S. Department of Defense news briefing on February 12, 2002.

ditions. We know that there are different conditions like sunny, rainy, and windy, but we also know that we do not know all different possible combinations and the according correct organizational structures at design-time of the test (or at least we cannot compute all). However, an oracle has to cope with that and has to decide whether a result is accepted as correct or rejected as incorrect.

***C-BranchingStateSpace*** A huge state space is a common challenge for software testing [9], but, as for the oracle problem, SO algorithms add a further dimension. Most of the approaches in software testing coping with a huge state space make use of the structure of the state space to reduce the amount of test cases needed to be executed. For instance, an infinite loop in a code fragment means an infinite state space, but its ramification degree is rather small. A mechanism applied here is boundary-interior-testing [33] that cuts deep branches at certain lengths. SO algorithms, however, are mostly based on heuristics for coping with the ever-changing environment, making the result non-deterministic; these lead to a wide and rather flat branching structure of the state space and make most of the classical techniques hardly applicable directly.

We address the challenges for testing SO algorithms in our framework *Isolated Testing of Self-organization Algorithms* (IsoTeSO) encompassing test case generation, execution, and evaluation. It provides techniques and concepts for methodically decomposing and isolating the SOuT and executing them in a controlled environment. The isolation can take place on several levels, e.g., isolation from other SO algorithms or from the environment controlled by the SO algorithm itself. Addressing C-Isolate further enables to cope with C-ErrorMask since we can reduce disturbances with the testing environment. However, it turns out that this measure is not enough to address C-ErrorMask, since for SO algorithms operating in several phases the error masking may occur within the SO algorithm itself. We hence introduce a gray-box access to the SOuT by the oracle to be able to detect failures hidden in the depth of the SO algorithms. This is embedded in the automated oracle of IsoTeSO whose implementation addresses C-Oracle based on the concept of what we call the corridor of correct behavior. This corridor allows to distinguish between correct and incorrect states even for the "know-unknowns", making it capable for evaluating the test results of the SO algorithm. Test case generation within IsoTeSO is based on a static and dynamic test model concept that is capable of handling the complex environment of SO algorithms, addressing C-BranchingStateSpace by systematically selecting test cases based on their occurrence probability within the application domain.

The remainder of this paper is structured as follows. Section 2 introduces the smart-grid case study used for evaluation and Sect. 3 presents our main ideas for testing SOAS. In Sect. 4, we detail the building blocks of the IsoTeSO framework including the test models. The framework is then used for testing two different SO algorithms introduced in Sect. 5. The evaluation in Sect. 6 confirms that our approach encompasses an efficient combination of model-based and random generation techniques based on our test models for finding different kinds of (injected) faults in the examined SO algorithms. Section 7 discusses related work. We conclude with a summary and some ideas for future work in Sect. 8.

## 2 Case Study: Self-organized Creation of Virtual Power Plants in Smart Grids

The wide-spread installation of weather-dependent power plants as well as the advent of new consumer types like electric vehicles put a lot of strain on power grids. Additionally, small dispatchable power plants (e.g., biogas plants) owned by individuals or cooperatives feed in power without external control. To save expenses, gain more flexibility, and deal with uncertainties, future *autonomous* power management systems have to take advantage of the full potential of dispatchable prosumers[2] by incorporating them into the scheduling scheme. Further, uncertainties have to be anticipated when creating schedules and compensated for locally to prevent their propagation through the system.

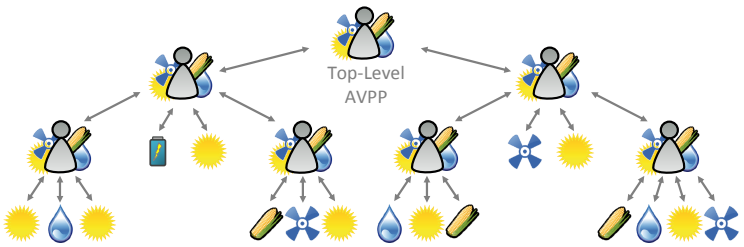


**Fig. 1.** Hierarchical system structure of a future autonomous and decentralized power management system: Power plants are structured into systems of systems represented by AVPPs that act as intermediaries to decrease the complexity of control and scheduling. AVPPs can be part of other AVPPs. The left child of the top-level AVPP, for instance, controls a solar power plant, a storage battery, and two subordinate AVPPs.

To meet the challenges of future power management systems, Steghöfer et al. presented the concept of *Autonomous Virtual Power Plants* (AVPPs) in [44] (similar visions of virtual power plants are discussed in [37]). AVPPs represent self-organizing groups of two or more power plants of various types (cf. Fig. 1). The organizational structure represents a *partitioning*, i.e., every power plant is a member of exactly one AVPP, which is established and maintained by a (partitioning-based) SO algorithm. Constraints that specify valid partitionings, e.g., a maximum number of power plants that may belong to one AVPP or that every power plant has to belong to exactly one AVPP, among others, induce a *corridor of correct behavior* over the space of all partitionings. In this setting, each AVPP has to satisfy a fraction of the overall demand. To accomplish this task, each AVPP autonomously and periodically calculates schedules for directly subordinate dispatchable power plants. Further, each AVPP's dispatchable power plants have to reactively compensate for deviations resulting from local output or load fluctuations (i.e., uncertainties) to avoid affecting other parts of the system.

[2] We use the term "prosumer" to refer to producers as well as consumers.

AVPPs autonomously adapt their structure to changing internal or environmental conditions, they are able to live up to the responsibility of maintaining an organizational structure enabling the system to hold the balance between energy supply and demand. In particular, if an AVPP repeatedly cannot satisfy its assigned fraction of the overall demand or compensate for its local uncertainties, it triggers a reorganization of the partitioning. The goal is to form homogeneous partitionings in the sense of a structure of similar AVPPs that are likely to feature a heterogeneous composition: On the one hand, by distributing unreliable power plants among AVPPs, the chance of fluctuations is reduced and the system's robustness increases. On the other hand, by balancing the AVPPs' degrees of freedom (e.g., by mixing different generator types), their ability to locally deal with uncertainties, i.e., fluctuations, is promoted. To cope with the vast number of dispatchable power plants, the concept of AVPPs proposes a scalable, hierarchical structure in which AVPPs act as intermediaries. This system decomposition reduces the number of dispatchable power plants (including directly subordinate AVPPs) each AVPP controls resulting in shorter scheduling times for each AVPP and the overall system.

In the smart-grid application, we have to cope with C-Isolate as well as C-ErrorMask since the partitioning algorithm and the mechanism that balances supply and demand in each AVPP influence each other significantly. Error masking can also occur in the partitioning algorithms if, e.g., the result of the SO algorithm is faulty but the system state after the reorganization process is valid. The problem of state space explosion (C-BranchingStateSpace) has to be tackled in this case study due to the stochastic nature of the partitioning algorithms in order to deal with the large search spaces, the non-deterministic behavior of single agents, and the stochastic environment (e.g., weather conditions, market prices). Finally, C-Oracle occurs in this context since it is hardly decidable at design-time which partitioning for which power plants is correct as this depends on the unknown environmental setting of the controlled power plants as well as on the unknown states of the autonomous power plants themselves.

## 3 The Corridor Enforcing Infrastructure (CEI) for Testing Self-organizing, Adaptive Systems

Our approach for testing SOAS—and consequently for testing SO algorithms—is based on the *Corridor Enforcing Infrastructure* (CEI) [14]. The CEI is an architectural pattern for SOAS using decentralized feedback-loops to monitor and control single agents or small groups of agents in order to ensure that the system's requirements are fulfilled at run-time. Within the CEI the concepts and fundamentals of the *Restore Invariant Approach* (RIA) [22] are applied. RIA defines the *Corridor of Correct Behavior* (CCB), which is described by requirements concerning the system's structure, formalized as constraints. Concerning the smart-grid scenario (introduced in Sect. 2) the CCB is formed by the constraints describing valid partitionings for the AVPPs, e.g., a maximum and minimum number of power plants that are allowed in each AVPP. The con-

junction of all these constraints is called the *invariant* (*INV*). An exemplary corridor is shown in Fig. 2. If *INV* is satisfied, the system is inside the corridor; otherwise, the system leaves the corridor, indicated by the flash. At this point the system has to be reorganized in order to return into the corridor, as shown by the transition with a check mark. A failure occurs in this context if a transition outside of the corridor is taken, indicated by a cross.

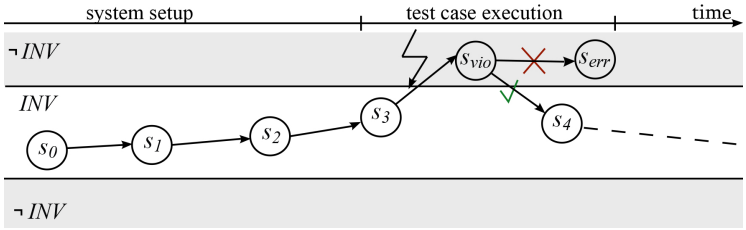


**Fig. 2.** Schematic state-based view of the corridor of correct behavior and the different phases of testing SO algorithms. *INV* is the conjunction of all constraints of the system controlled by the CEI.

The CEI implements the RIA with decentralized pairs of a monitor and a controller, similar to the MAPE cycle [25] or the Observer/Controller (O/C) architecture [41]. The schematic view in Fig. 3 shows an implementation of the CEI based on the O/C architecture where the essential parts are the system under observation and control (SuOC, i.e., single agents or groups of agents controlled), the observer (O, i.e., the component monitoring the state of the SuOC and providing information to the controller) and the controller (C, i.e., the self-x-algorithms controlling the SuOC, e.g., an SO algorithm). Note that the CEI consists of sets of nested feedback-loops controlling the entire system. Figure 3 shows further the different layers of testing applied to cope with the complexity of the system: agent, interaction, and system layer. The IsoTeSO framework is located in this concept on the *Interaction Layer* where the SO algorithms are incorporated into the *Controller*.

In some cases the observer for single agents or small groups of agents can be generated (semi-)automatically from the requirements documents for the considered system, as shown by Eberhardinger et al. [15]. The small groups controlled in the smart-grid scenario are the power plants partitioned into AVPPS. Furthermore, single power plants are in control loops to adjust, in particular, their energy production. The reorganization by the controller is performed by one or more SO algorithms resulting in a new system configuration. Such a system configuration has to satisfy the constraints describing valid organizational structures, e.g., a maximum number of controlled components within each organization. With regard to the partitioning problem—applied for the AVPPs—this means that each power plant belongs to exactly one AVPP. For other application scenarios one might require a structure in which each component belongs to at least one

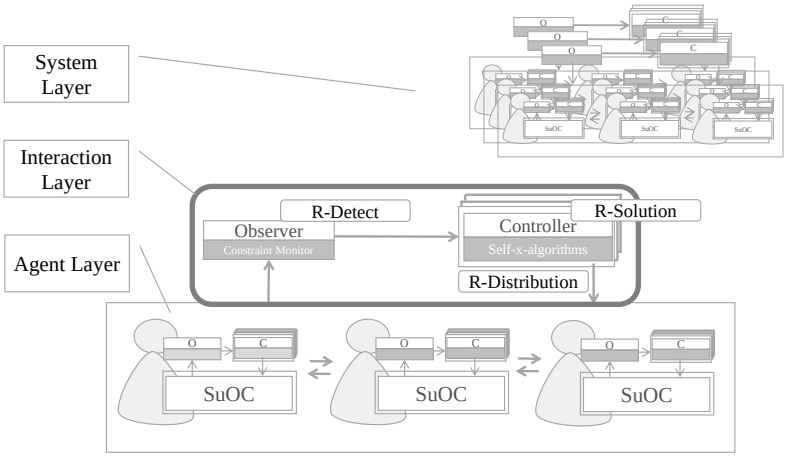


**Fig. 3.** Schematic view of the *Corridor Enforcing Infrastructure* (CEI) and its different level of testing (namely, agent, interaction, and system layer). The IsoTeSO Framework is located in this concept on the *Interaction Layer* where the SO algorithms are incorporated into the *Controller*. Besides the *Controller (C)* the CEI main components are the *System under Observation and Control (SuOC)*, and the *Controller (C)* that form together distributed, decentralized feedback-loops on different levels, i.e., the CEI consists sets of nested feedback-loops (symbolized by the nested set of different system parts in the upper part of the figure) controlling the entire system.

organization, which corresponds to a set covering [7]. The concrete choice of the organization algorithms and their constraints has no impact on our approach, both set covering and partitioning SO algorithms could be implemented within the concept of the CEI.

In order to grasp SO algorithms for testing, we need techniques to examine the CEI and its mechanisms, covering the following responsibilities of the CEI (marked with a rounded rectangle in Fig. 3): correct initiation of a reorganization if and only if a constraint is violated (monitoring infrastructure, R-Detect); calculation of correct system configurations in case of violations (R-Solution); and correct distribution of these configurations within the single agents or small groups of agents controlled by the CEI (R-Distribution). At the moment, IsoTeSO focuses on revealing the kinds of SO algorithms failures related to (R-Solution) and (R-Distribution). Section 8 gives an outlook on the possible extensions of the framework for testing the monitoring infrastructure (R-Detect).

On the other hand, the CCB immediately affords for our testing efforts a check whether a system configuration produced by an SO algorithm for reorganization is correct, i.e., whether all constraints of the CCB hold. This integrated check consequently forms the basis of a test oracle in the IsoTeSO framework. However, in order to use the CCB correctness check for result validation, a consistent snapshot of the system state is needed. Therefore, in a first step, IsoTeSO is built upon a stepwise execution model that naturally allows to synchronize

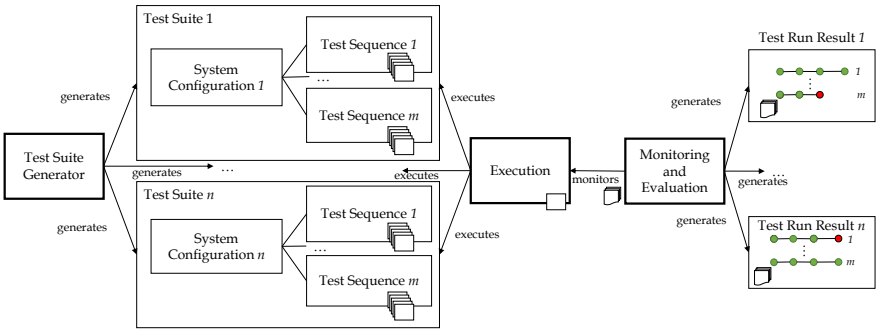


**Fig. 4.** IsoTeSO generates $n$ test suites, each consisting of an initial system configuration for the test setup and $m$ test sequences. Each test sequence contains a number of test cases (indicated by the overlapping rectangles). Each test sequence is executed individually (indicated by the single rectangle in the execution component). The monitoring and evaluation component monitors the system and samples the reorganization results as well as the system structure for each test case (indicated by the forms at the arrow between the monitoring and evaluation component and the execution component). For each test suite, the results of all test cases are evaluated in a test run result, leading to $n$ results for $m$ evaluated test sequences.

all system components at distinct points in time. The necessary synchronization may reduce the possible interleavings of component behaviors, but this may be mitigated by letting components decide for making an idle local step in every synchronous round. Still, the concept of the CCB makes it possible to create a fully automated oracle that allows to tackle C-Oracle, since it is possible to decide even under unknown conditions of execution whether a state is inside the CCB or not, addressing an evaluation of the "known-unknowns".

## 4 A Framework for Isolated Testing of Self-organization Algorithms (IsoTeSO)

As common for testing frameworks, we organize IsoTeSO in a process-oriented manner into: test case generation[3] (cf. Sect. 4.2), test execution (cf. Sect. 4.3), and, finally, test evaluation as a part of output processing (cf. Sect. 4.4). An overview of the framework is shown in Fig. 4.

To derive appropriate test suites within the test suite generator component, we use the test model presented in Sect. 4.1 that enables sufficient abstraction to tackle C-BranchingStateSpace. IsoTeSO allows for isolating SO algorithms from

[3] Due to the fully automated evaluation of test cases by the oracle component of IsoTeSO, test case generation reduces to test input generation as no expected output is needed. This concept builds up on Artho et al. [6] also combining run-time verification and test input generation for creating test cases. In the remainder of this paper we use test case generation in the sense of test input generation.

other parts of the system, including other SO algorithms (C-Isolate). For this purpose, IsoTeSO provides a fully controlled test environment, i.e., the environment of the SOuT is fully mocked. To enable mocking, the execution component, described in Sect. 4.3, provides a system simulator and a gray-box interface for the SOuT. The gray-box interface enables the evaluation of the internal state of the SOuT, e.g., for checking interim results, and thus also allows us to address C-ErrorMask. The evaluation of the validity of reconfigured system structures is performed by the test oracle that checks the violation of constraints describing valid solutions (C-Oracle). The responsible component for the test evaluation—the monitoring and evaluation component—is described in item 2.

Overall, we assume that the test system is embedded into an agent-based execution system. To allow a consistent snapshot and therefore coherent synchronization points after each executed test case, the system is based on a stepwise execution model.

### 4.1 Test Model of the Framework IsoTeSO

The test model of the framework is the source for generating test suites (cf. Sect. 4.2) and is composed of three parts:

1. Model of the system under test (SuT)
2. Model of the SO algorithm under test (SOuT)
3. Model of environmental changes and influences

The first one provides the necessary information for the domain in which the SOuT is applied, the second one defines suitable configurations for the SOuT which are highly influenced by domain knowledge. The model of environmental changes and influences describes the dynamics of the environment and is mainly used for creating test sequences. In the following, these three parts of the test model are described with respect to their intended purpose within the test suite generation component.

*Model of the system under test.* The model of the SuT specifies all important information from the domain the SOuT belongs to and is used for generating the system configuration within the test suites. For this purpose, a domain description (in the form of a UML class diagram) is provided that is enriched by constraints describing the CCB (cf. Sect. 3). With regard to our case study, the constraints define valid partitionings, e.g., the maximum and minimum number of power plants for each AVPP as well as the constraint that each power plant must be contained in exactly one AVPP. Further, the behavior of each agent type (e.g., wind turbines, solar panels, or biogas power plants) is defined by standard design documents. In our smart-grid case study we additionally have to define *groups of agents* which are influenced by similar environmental changes, such as wind turbines with a certain locality. Accordingly, a group of agents share one *environment profile*, a stochastic model abstracting possible environmental changes (more details are given at the end of this subsection). The fact that an

agent is in a certain group, is not known to the SOuT and has no direct influence on the partitioning decisions of the SOuT. The members of a group of agents can be of different agent types.

We define how the environment influences certain types of agents so that a reduction of the test cases to realistic scenarios is possible. Indeed, describing the influences relation between the environment and types of agents is rather complex. At this step we rely on simplification and abstraction within the model to keep our approach scalable: We assume that the mapping of environment influences to agent types (in its abstracted form) can be determined and defined a priori, i.e., we neglect that the influence might change over time or is indeterministic. The consequences of this assumption and simplification are on the one hand that the model is scalable within the approach and can be handled by the test engineer, but, on the other hand, that it might neglect some situations for testing. Our evaluation results (cf. Sect. 6.3), however, showed that it is still possible to find different kind of failures.

To recap, the model of the system under test must contain at least:

- Types of agents
- Definition of possible initial states for the agents
- Suitable ranges for the minimum and maximum number of agents of a specific agent type
- Constraints concerning groups of agents (minimum and maximum number as well as size)
- Mapping of environmental influences to agent types
- Constraints concerning valid system structures, i.e., the relevant part of the CCB

Since the model of the system under test depends on the application domain the description for the model itself is quite generic and coarse. However, the detailed information of the application—given in Sect. 6—shows how to transfer this generic description into a specific model of the system under test.

*Model of the SO algorithm under test.* This model specifies valid configurations for the SOuT. It is used to derive valid ranges for all relevant parameters, such as the algorithm's maximum run-time. The selection of relevant parameters highly depends on the concrete algorithm (e.g., some algorithms allow to specify a maximum execution time and some do not), the situations that should be covered by the test runs (e.g., some failures only occur if we give the algorithm enough time), and the domain knowledge (e.g., in some domains, the maximum run-time is naturally bounded). Clearly, despite the random testing approach, a suitable parametrization can be used for directed testing. This means that we can push the algorithm into interesting directions, e.g., to use specific functionality, which increases the code coverage.

The parameters of the SOuT and dependencies between the parameters are specified as constraints of the permitted setting of the parameters of the SOuT. The resulting constraint satisfaction problem (CSP) is solved by the input component of the framework for forming a system configuration within a test run.

For a SOuT that is based on a particle swarm optimization algorithm to form organizational structures (like the PSOPP algorithm described in Sect. 5.2) a parameter of the algorithm is the number of particles to use, constrained in the model of the SOuT, for instance, by an upper and lower bound. Further, the number of particles are related to the constraints concerning the number of partitions to be formed. This relationship is incorporated into the CSP. The described CSP is the foundation for automatically generating valid settings for the SOuT. A more detailed example for the specification is given in Sect. 6.

*Model of environmental changes and influences.* Recalling our idea of isolated testing of SO algorithms, the third component of the test model is the model of environmental changes and influences. This is because of possible interferences with other SO algorithms due to interleaved feedback loops (C-Isolate) as well as the huge state space induced by the different possible states of the environment and the algorithm's non-deterministic behavior (C-BranchingStateSpace). We address C-BranchingStateSpace by providing stochastic models of the environment, called *environment profiles* (EPs), and C-Isolate by *functions describing the environment's influence* on the system that enables to decouple the SO algorithm.[4]

As we do not assume that all agents share the same environment (e.g., because of their geographical distribution) and are equally influenced by environmental conditions, we define these EPs and influence functions with regard to a specific group of agents $\mathcal{G}$.[5] This allows us to deal with large state spaces even better (C-BranchingStateSpace). For example, it is not necessary to consider the complete set of possible states of the environment $\mathcal{E} = \{(\mathsf{cloudy}, \mathsf{high\ price}), (\mathsf{rainy}, \mathsf{high\ price}), (\mathsf{sunny}, \mathsf{high\ price}), (\mathsf{cloudy}, \mathsf{low\ price}), (\mathsf{rainy}, \mathsf{low\ price}), (\mathsf{sunny}, \mathsf{low\ price})\}$ if we regard a group of solar power plants whose output mainly depends on the current weather conditions and is more or less independent of the current market price (a property that is defined in the mapping of environmental influences to agent types). Instead, for each group of agents $\mathcal{G}$, we map one or more states of the environment from the set $\mathcal{E}$ to a single so-called *relevant state* that describes the relevant parts of the environment's state for $\mathcal{G}$, i.e., those that have an influence on $\mathcal{G}$'s behavior. By gathering all relevant states, we obtain the entire set of relevant states $\mathcal{R}_\mathcal{G}$ for $\mathcal{G}$.

In case of our group of solar power plants the states $(\mathsf{cloudy}, \mathsf{high\ price})$, $(\mathsf{cloudy}, \mathsf{low\ price}) \in \mathcal{E}$ are mapped to a state $\mathsf{cloudy}$ that becomes a member of $\mathcal{R}_\mathcal{G}$. In this example, $\mathcal{R}_\mathcal{G}$ is finally equivalent to the set of weather conditions $\{\mathsf{cloudy}, \mathsf{rainy}, \mathsf{sunny}\}$ considered in $\mathcal{E}$.

The identification of relevant states is supported by the mapping of environmental influences to agent types as described in the model of the system under test. Thus, if the environment state or a set of environment states corresponds to an environmental influence that is already mapped to an agent type of the con-

[4] Note that the environment also covers in this case other SO algorithms of the system.
[5] That technique of state reduction is performed according to the state abstraction principles that are well known in classical testing [33].

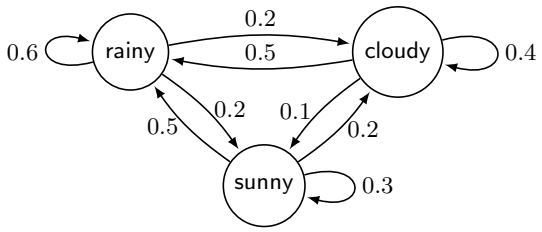


**Fig. 5.** A simplified EP for a group of solar power plants at a specific geographic location: Possible weather changes between rainy, sunny, and cloudy are indicated by directed edges. The numbers represent transition probabilities.

cerning agent group, this state has to be included into the set of relevant states for this agent group. However, mapping the environment's states to relevant states of an agent group is—as the mapping of environment influences to agent types in the model of the system under test—not generically solvable; in every application domain this classification of relevance has to be made specifically. Further, the relevant states are not limited to the mapping described for the environmental influences to agent types, since, among other things, the other SO algorithms are here also part of the environment (where as that is not considered by the model of the system under test).

The exemplified description above shows what the necessary steps are and how this mapping should be achieved. The mapping in general does not have to be disjoint but we expect it to be complete since only useful environment states should be included in $\mathcal{E}$, i.e., states that influence at least one of the agent groups.

With regard to a specific group of agents, an EP not only captures the relevant states $\mathcal{R}_\mathcal{G}$ of $\mathcal{G}$'s environment, but also probabilities for changes from one state to another. Assuming that the next state only depends on the current state, an EP represents a first-order Markov chain. Fig. 5 depicts a simplified example of an EP for a group of solar power plants in a specific region; as a matter of fact, the models used for testing are much more complex (cf. Sect. 6).Such an EP can either be created using domain knowledge, derived from statistical data gathered during the execution of the system under test, or a combination of both. In the literature of multi-agent systems, Markov chains are often used to simulate the environment for evaluation purposes (cf. [42,1]).

To model the way the environment influences the members of an agent group $\mathcal{G}$, we use a function $f_\mathcal{G} : \mathcal{R}_\mathcal{G} \times \mathcal{S}_\mathcal{G} \to \mathcal{S}_\mathcal{G}$, where $\mathcal{S}_\mathcal{G}$ represents all possible states of $\mathcal{G}$'s members. With regard to a member $a \in \mathcal{G}$, the function $f_\mathcal{G}$ maps the new state $\sigma'_{env} \in \mathcal{R}_\mathcal{G}$ of $\mathcal{G}$'s environment and $a$'s current state $\sigma_a \in \mathcal{S}_\mathcal{G}$ to a new state $\sigma'_a \in \mathcal{S}_\mathcal{G}$. For instance, the change of the current weather conditions from sunny to $\sigma'_{env} =$ rainy could impair a solar power plant's ability to make adequate predictions of its future output, which is reflected in the transition from $\sigma_a =$ good predictions to $\sigma'_a =$ bad predictions. Different influences of the weather on different types of weather-dependent power plants—represented as

different agent groups—can be formalized by group-specific functions $f_{\mathcal{G}}$. On the other hand, if an EP describes possible developments of the prices at an energy market, $f_{\mathcal{G}}$ can model the way a power plant or consumer behaves at the market, i.e., its strategy. For example, if the market price falls below a certain threshold, some consumers might change their strategy to "buy energy", whereas the producers might become more reluctant to sell their production. As is the case with the creation of EPs, such influence functions can be deduced from domain knowledge and statistical data. Note that, in some cases, the influence function might depend on random variables and thus becomes probabilistic. This reflects the fact that we cannot assume perfect knowledge about the influences of the environment on the agents due to the high complexity and the agents' autonomy. However, it is depends on the test designer's choice as well as on the application domain whether to use a probabilistic functions to map the environment influences to the states of agent groups. The drawback of modeling these functions on random variables is that the process of test case generation is less controllable, but it might reveal some interesting test cases. In our evaluation (cf. Sect. 6), we used deterministic functions which especially payed off in the process of getting more control over the test case generation procedure and gaining higher failure detection rates. Of course, there might be applications where detailing the model applies better in order to gain a higher failure detection rate due to the more accurate model that is for instance able to reveal border cases that are failure prone. The choice depends here on the test designer that adapts the presented approach to specific algorithms and application domains.

As we will explain in the next subsection, the EPs are used for the test case generation by simulating the Markov chains, yielding random but representative test sequences describing environmental changes for the different groups of agents. The probabilistic information of the EPs can be used to estimate the relevance of generated test cases and test sequences and their likelihood of occurrence in a realistic setting (C-BranchingStateSpace). As we will see in our evaluation in Sect. 6, this property makes EPs also an important tool for coverage analysis.

### 4.2 Test Suite Generator Component of IsoTeSO

IsoTeSO's test suite generator component generates different test suites each containing *one* initial system configuration on which basis it randomly creates $n$ test sequences by using the environment profiles (cf. Fig. 4).A randomly generated initial system configuration contains the following information:

- a set of agents, possibly of different types
- a set of initial agent states, one for each agent
- a set of agent groups so that each agent is contained in exactly one group
- an initial system structure
- the selected SOuT
- additional system parameters, including the parametrization of SOuT and a random seed to be used during the execution of the test sequences

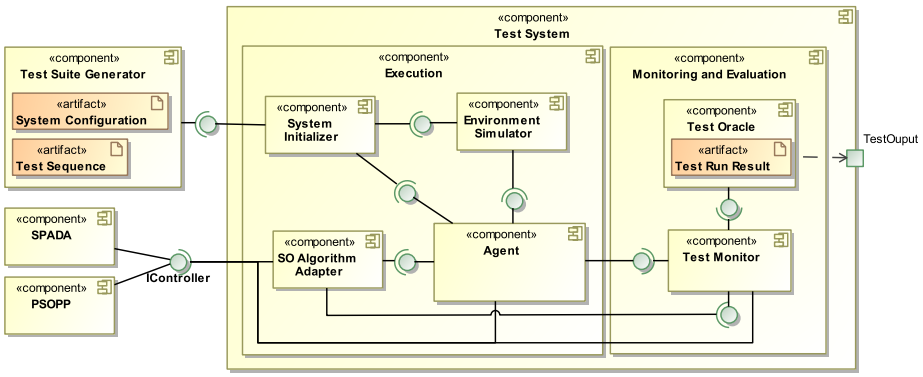


**Fig. 6.** UML component diagram of the detailed architecture of IsoTeSO having two different partitioning-based SO algorithms plugged in (SPADA, PSOPP). The *Test Suite Generator* component provides the artifacts of a test suite. The *System Initializer* component initializes the system using the generated test suites and sets up the *Environment Simulator* and the *Agents*. Within the *Execution* component, the test suites are executed via the *Environment Simulator* while the *Monitoring and Evaluation* component logs the execution (*Test Monitor*) and the *Test Oracle* evaluates it to *Test Run Results*.

The fact that all sequences within one test suite start with the same system configuration is important in the context of SOAS to tackle the already mentioned four challenges (cf. Sect. 1). Each test sequence is an ordered series of test cases whose length corresponds to the number of discrete (time) steps that should be simulated in the course of the test run. This is possible since IsoTeSO uses a stepwise execution model. To generate the test cases, a chosen set of EPs is simulated for the specified number of time steps to obtain the sequence of environmental changes.

IsoTeSO's test suite generator component can be used in two different modes concerning the execution of the framework:

- *Online:* This mode produces new test suites during the execution of the framework. It accordingly enables an "endless" execution of the framework. For this purpose a new test suite is generated and executed after every completed test run.
- *Offline:* This mode assumes a given set of test suites which is generated a priori and sequentially executed by the framwork.

In both modes, the test suites are executed by the execution component.

### 4.3 Execution Component of IsoTeSO

The execution component defines two phases: (1) setting up the system and (2) executing the test cases on the SOuT.

*Setting up the system.* First, the *System Initializer* sub-component processes the initial system configuration provided by the current test suite. For this purpose, it sets up the set of predefined agents, using the given initial agent states, and establishes the given initial system structure. Next the SOuT is incorporated via an interface (`IController`) which plugs in the SOuT via an adapter component. At this stage, we can use IsoTeSO only for black-box testing, since we have no access to interim results of the SOuT. For the isolated testing of our partitioning algorithms, we plugged in SPADA and PSO as shown in Fig. 6. The adapter component is an implementation of the adapter pattern that enables efficient testing of different SO algorithms. Therefore, the adapter fulfills the following tasks, when reorganizing the system structure: (i) it instantiates the specified SOuT, (ii) sends the current system structure to this SOuT, (iii) as soon as the SOuT terminates, the adapter informs the test monitor (as described in Sect. 4.4) about the solution, and (iv) after the monitor checked the solution, the adapter requests the SOuT to adopt the new system structure. Point (iii) yields a gray-box view for testing, since we are now able to check interim steps of the test case execution. This is needed for testing if the SOuT—as part of the controller of the CEI—fulfills (R-Solution) besides (R-Distribution).

*Executing test cases on the SOuT.* After the setup is completed, the environment simulator sequentially executes the given test cases. This execution is based on a stepwise execution model that enable consistent synchronization points after every test case. First, the agent components perceive the corresponding environmental changes and adapt their internal states according to their influence functions $f_{\mathcal{G}}$. After the changes are applied to every component, the system constraints are checked for a violation, i.e., whether the system leaves the CCB, if so the SOuT is activated in next step via the adapter component.[6] Based on the current system state, the SOuT has to reorganize the system structure in order to restore the system constraints, i.e., to bring the system back into the CCB. That last action completes one execution sequence that correspond to the executed test case.

The gray-box view on the SOuT allows us to tackle (R-Solution) as well as C-Oracle since we can evaluate the calculated system structure before it is distributed among the agents. In addition, the gray-box view consequently avoids error masking during testing—defined as challenge C-ErrorMask. The adapter continuously informs the test monitor about the current system state during the test case execution at specific points in time. Gaining the synchronized system state is possible due to the stepwise execution model. Based on this information, the test oracle can decide if the distribution of the solution leads to a correct system configuration, thereby addressing (R-Distribution).

[6] Note that not every test case execution leads directly to constraint violations and thus to an activation of the SOuT. To form a realistic system structure within the test system, it is necessary to allow the system to take transitions that do not violate the CCB.

### 4.4 Monitoring and Evaluation Component of IsoTeSO

The monitoring and evaluation component of IsoTeSO—that mainly tackles C-Oracle by applying the concepts of the CCB to monitoring and evaluation—is split into two sub-components, each being responsible for a specific task:

1. the test monitor observes and samples data of the executed SOuT and of the test system
2. the test oracle evaluates the sampled data

To live up to these responsibilities, the monitoring and evaluation component directly interacts with the execution component and the SOuT using the provided interfaces (cf. Fig. 6).

*Monitoring & sampling data.* In the course of the execution of a test suite, the monitoring and evaluation component monitors and samples data of the test system influenced by the SOuT as well as directly of the SOuT. This allows to observe the two steps of a SOuT: (1) computing a solution for the reorganization, and (2) distributing the solution to the agents. The result of these steps is stored in the current system state, and sent to the test oracle for further evaluation. A necessary prerequisite for monitoring and sampling data is the established stepwise execution model, allowing to synchronize all system components at distinct points in time for later evaluation. However, the synchronization of all system components does not imply prerequisites or restrictions to the SOuT's characteristic. Having a synchronization point and a global view within the test system does not interfere the decentralization of the SOuT since the SOuT itself makes no use that global view, it is only introduced for test oracle. Indeed, there are some restrictions concerning the possible tested situations. The stepwise execution model neglects test cases that address situations where components are in different states. Incorporating them into our test approach is a matter of future work outlined in Sect. 8.

*Evaluating the data by a test oracle.* Having received the sampled state for a test case from the test monitor, the constraint-based test oracle evaluates the result of the test case either to *passed* or *failed*. The evaluation of the results is based on constraints, which represent the requirements for the SOAS as well as the SOuT and form the CCB (as described in Sect. 3). Each constraint must be satisfied by the given system structure to pass the test. The evaluation of the constraints can be automatized by parsing them to checks whether or not a test succeeds. Besides this evaluation based on the constraints, IsoTeSO also supports smoke tests, e.g., observations whether the system throws exceptions during execution or simply crashes. This enables a completely automatized test process within the framework and addresses C-Oracle.

As shown in Fig. 4, the monitoring and evaluation component generates test run results for a specific test suite. The test run result contains evaluations of each test case result (failed, passed) as well as the logged information (system configuration, SOuT solution) during the test run for each executed test case.

# 5 Tested Self-organization Algorithms

In this section, we present two SO algorithms, a decentralized approach (Sect. 5.1) and a metaheuristic (Sect. 5.2). The aim of both algorithms is to partition a set of agents $\mathcal{A} = \{a_1, \ldots, a_n\}$ into pairwise disjoint subsets, i.e., partitions, that together constitute a *partitioning* at as minimal costs as possible. With regard to our case study, each AVPP represents a partition and the set of all AVPPs corresponds to a partitioning. As SPADA and PSOPP are part of the controller of the CEI, the oracle has to evaluate if their interim results (R-Solution) as well as the resulting system structures (R-Distribution) meet the functional requirements, i.e., satisfy the properties of partitionings (cf. Sect. 6).

## 5.1 A Decentralized Algorithm for Partitioning Multi-Agent Systems

SPADA [4], the ***S**et **P**artitioning **A**lgorithm for **D**istributed **A**gents*, solves the *complete set partitioning problem* (CSPP) in a general, decentralized manner. In the following, we give a short summary of SPADA's basic functionality and characteristics. A more detailed description can be found in [4].

In SPADA, the agents use an internal graph-based representation of the current partitioning, called *acquaintances graph*, to solve the CSPP. All operations the agents apply to establish a suitable partitioning can therefore be mapped to graph operations. The nodes of the acquaintances graph are the agents participating in the reorganization. Directed edges represent acquaintance relationships between agents. Together the acquaintances form an overlay network that restricts communication to acquainted agents, thereby lowering complexity in large systems. To indicate that an agent is not only acquainted with another but also in the same partition, edges can be marked. Partitions are thus defined by the transitive-reflexive closure of the binary relation given by the marked edges. Each partition has a designated leader that is responsible for optimizing its composition according to application-specific criteria. An example of such an acquaintances graph is depicted in Fig. 7 (more details concerning the acquaintances graph can be found in [4]).

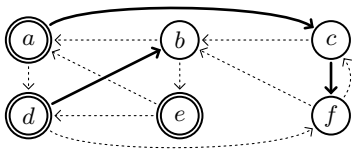


**Fig. 7.** An exemplary acquaintances graph for a system consisting of six agents (cf. [4]): Agents are represented as nodes and acquaintances as directed edges, e.g., $d$ is acquainted with $b$ and $f$. Marked edges (symbolized as solid arcs) indicate that their tail and head belong to the same partition. In this example, there are three partitions $\{a, c, f\}, \{b, d\}, \{e\}$ with leaders $a, d, e$.

To improve its partition, each leader periodically evaluates if it is beneficial to integrate new agents or to exclude some of its members (e.g., with regard to our case-study, to improve the equal distribution of unreliable power plants among AVPPs). The latter can be beneficial in case of reorganizations that require to create new partitions, e.g., if a partition's or an agent's properties have changed so that the partition's formation criteria no longer favor including the agent. The integration and exclusion of agents is implemented by modifying the edges in the acquaintances graph.

To decide about termination, leaders periodically evaluate application-specific termination criteria. These are formulated as constraints which can also be monitored at run-time to trigger reorganization. If the termination criteria are met, the leader marks its partition as terminated. As long as a partition is marked as terminated, its leader does not change its structure. However, the termination labeling is removed if the partition is changed from outside, i.e., if one of its members is integrated into another partition. This characteristic allows SPADA to make selective changes to an existing partitioning, which is very useful in dynamic environments. It has been shown empirically that SPADA's local decisions lead to a partitioning whose quality is within 10% of the optimum [4].

With regard to our case study, each leader instantiates a new AVPP agent as soon as all partitions terminated. The AVPP then assumes control of all power plants in the partition. In case of a reorganization, the acquaintances graph is created on the basis of the existing system structure.

### 5.2 A Particle Swarm Optimizer for Partitioning Multi-Agent Systems

*PSOPP* [3], the ***P**article **S**warm **O**ptimizer for the **P**artitioning **P**roblem*, is based on *Particle Swarm Optimization* (PSO) [24], a bio-inspired computational method and metaheuristic for optimization in large search spaces. In PSO, a number of particles concurrently explore the search space in search of better candidate solutions by modifying their current positions (at random or by approaching other candidate solutions) as long as a specific termination criterion is not met. During this process, each particle's current position represents a specific candidate solution. To be able to improve the quality of candidate solutions in a target-oriented manner, each particle $\Pi_i$ is aware of its best found solution $\mathcal{B}_i$ and the best found solution $\mathcal{B}_{\mathcal{N}_i}$ in its neighborhood $\mathcal{N}_i$. The algorithm's outcome is the global best found solution $\mathcal{B}$.

PSOPP solves a variant of the CSPP in the presence of *partitioning constraints* that constrain feasible partitions in terms of a minimum $s_{min}$ and a maximum $s_{max}$ size as well as a minimum $n_{min}$ and a maximum $n_{max}$ number of partitions. Therefore, the test oracle additionally has to check—without interfering the algorithm, by evaluating the logged data of the algorithm and not locking it during execution—if the interim results and the resulting system structure satisfy the partitioning constraints. In PSOPP, each particle represents a partitioning that satisfies the partitioning constraints. The central idea—which could also be applied to other metaheuristics—is to allow the particles to move

around the search space by using the basic set operations *join*, *split*, and *exchange*. The join operation creates the union of two partitions, the split operation divides an existing partition into two non-empty subsets, and the exchange operation exchanges elements between two partitions. Particles can apply these operations at random as well as in a target-oriented manner. The main purpose of the former case is to enable the particles to explore the search space by randomly modifying their represented partitioning, i.e., their position. The latter case, in contrast, allows particles to exploit existing candidate solutions by approaching other candidate solutions in promising regions of the search space. Since PSOPP's operations are defined in a way that their application always maintains solution correctness, it combs through a search space that only contains correct solutions. This is advantageous with regard to its performance. Because PSOPP is initialized with a correct candidate solution, it is an any time algorithm.

Similar to SPADA, PSOPP can be customized to a specific application by devising an appropriate fitness function that assesses the quality of solutions and thus steers the search for them. Due to these characteristics, PSOPP can be applied to many different applications in which solving the partitioning problem considered in this paper is relevant and global knowledge is available.

Having specified valid partitionings by means of $n_{min}, n_{max}, s_{min}, s_{max}$ as well as the particles' attitude towards exploration and exploitation by fixing some parameters that influence the probability that particles make a random move or approach other candidate solutions, PSOPP creates a predefined number of particles at random or predetermined positions in the search space (the set of particles does not change at run-time). The latter is suitable when a reorganization of an existing system structure has to take place: If the current structure does not contradict the partitioning constraints, it can be used as a starting point for the self-organization process. Mixing predefined and randomly generated initial partitionings allows to hold up diversity. When searching for an initial system structure, particles are created at random positions.

As long as a specific termination criterion is not met, a particle $\Pi_i$ performs the following actions in each iteration; these are also depicted in Fig. 8:

1. Evaluate the fitness $f(\mathcal{P})$ of the represented partitioning $\mathcal{P}$.
2. If the particle's fitness $f(\mathcal{P})$ is higher than the fitness $f(\mathcal{B}_i)$ of its best found solution $\mathcal{B}_i$, set $\mathcal{B}_i$ to $\mathcal{P}$. Further, inform all other particles $\Pi_j$ that contain $\Pi_i$ in their neighborhood $\mathcal{N}_j$ about the improvement so that they can update $\mathcal{B}_{\mathcal{N}_j}$, i.e., the best found solution in their neighborhood.
3. Update the best found solution $\mathcal{B}_{\mathcal{N}_i}$ in $\Pi_i$'s neighborhood $\mathcal{N}_i$.
4. Stop if the termination criterion is met.
5. Otherwise, randomly opt for the direction in which to move, i.e., choose whether a random move or an approach operation should be applied. In case of an approach operation, also determine the position (i.e., $\mathcal{B}_i$ or $\mathcal{B}_{\mathcal{N}_i}$) that should be approached.
6. Determine the new position $\mathcal{P}'$ by applying the selected move operation to $\mathcal{P}$.

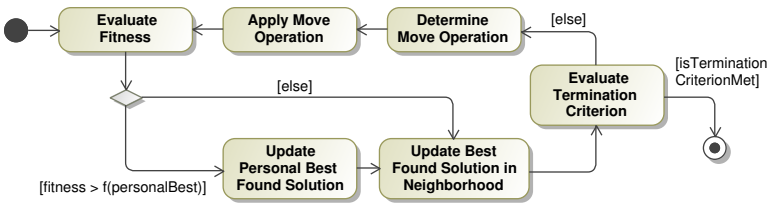


**Fig. 8.** Actions performed by particles in each iteration [3].

Once all particles terminated, PSOPP returns the best found solution $\mathcal{B}$. Possible termination criteria are, e.g., a predefined amount of time, a predefined number of iterations (i.e., moves through the search space), a predefined threshold for the minimum fitness value, or a combination of these criteria.

## 6 Evaluation

In our evaluation, we used a Java implementation of IsoTeSO that is based on a multi-agent system called TEMAS [2]. We opted for TEMAS because it supports a stepwise execution out of the box, which allows IsoTeSO to monitor consistent states of the system at specific points in time.

As foundation of the *model of the system under test*, we used the domain model of the AVPP application, which can be found in [5]. The model of the environmental changes and influences described the effect of different environmental conditions, such as weather conditions, on the power plants' ability to make adequate predictions of their future output. Clearly, this effect depends on the concrete type of agent, i.e., power plant. For this reason, we regarded four different agent types: solar panels, wind turbines, biogas power plants, and hydro power plants. Based on the assumption that the effect of changing environmental conditions, such as the global radiation, the wind speed, the available amount of biogas, or the water flow, is characteristic of a specific type of power plant, we generated different sets of agent groups. Further, the power plants' geographic location was taken into account. Considering an AVPP's prediction accuracy as a property resulting from the average prediction accuracy of its members, the system's goal was to maintain a structure of AVPPs that feature a similar prediction accuracy (cf. Sect. 2). As soon as the dissimilarity of the AVPPs' prediction accuracy exceeded a certain threshold (as a result of environmental changes), the power plants triggered a reorganization that should reestablish the similarity. We parametrized the *model of the system under test* as follows:

- number of agents $\#ag$: between 2 and 1000
- agent group size: between 2 and $\#ag$
- number of agent groups: between 1 and $\lfloor \frac{1}{2} \cdot \#ag \rfloor$
- partition size: $2 \leq s_{min} \leq s_{max} \leq \#ag$
- number of partitions: $1 \leq n_{min} \leq n_{max} \leq \lfloor \frac{1}{2} \cdot \#ag \rfloor$
- 10 test sequences per test suite

- number of test cases per test sequence: between 50 and 1000
- number of states per EP: between 3 and 25

As SOuT, we integrated a Java-based implementation of the partitioning algorithms SPADA and PSOPP (cf. Sect. 5). Both implement the IController interface that is used by the SO Algorithm Adapter to initiate the SOuT, request results, and ask the SOuT to adopt the new system structure (cf. Fig. 6). The latter was implemented by sequentially moving the power plants contained in the calculated partitioning from their current AVPP into the corresponding new AVPP. With regard to the system structure, this procedure assures that every power plant is always contained in exactly one AVPP. If the last power plant was removed from an AVPP, this AVPP was dissolved. The description of the algorithms provided in [4] and [3] as well as their implementation served as the *model of the SOuT*. As explained in Sect. 4.1, we used this information to identify relevant parameters and suitable valid ranges for their parametrization. For SPADA and PSOPP, we identified the following parameters:

- SPADA
  - number of acquaintances per agent: between 1 and 20
  - number of agents each leader evaluates for integration into its partition: between 1 and 10
  - maximum number of agents a leader can integrate into its partition within a single step: between 1 and 10
- PSOPP
  - number of particles $\#P$: between 1 and 4
  - number of particles starting at the current partitioning: between 0 and $\#P$
  - probabilities $c_{rdm}, c_{\mathcal{B}_i}, c_{\mathcal{B}} \in [0, 1]$ (with $c_{rdm} + c_{\mathcal{B}_i} + c_{\mathcal{B}} = 1$) to apply a random move operator, approach the particle's best found solution $\mathcal{B}_i$, and approach the global best found solution $\mathcal{B}$, respectively
  - max. run-time in seconds: between 1 and 10

After each reorganization, the oracle checks if the algorithm's result complies with the definition of partitionings (i.e., each power plant must be a member of exactly one AVPP). As explained in Sect. 4.3 and Sect. 4.4, these checks are performed once the algorithm indicates its termination (R-Solution) as well as after the result has been adopted (R-Distribution). Only in case of PSOPP, the oracle additionally evaluated the satisfaction of the partitioning constraints introduced in Sect. 5.2 since SPADA does not allow to restrict valid partitionings with regard to the size and number of partitions.

### 6.1 Fault Injection

To evaluate our approach, we injected four faults into the SPADA and five faults into the PSOPP implementation. According to Püschel et al. [36], all these faults can be assigned to the classes of "permanent and transient RECONF faults". Our injected faults have in common that they do not cause the algorithm to throw an exception that simply has to be caught by the oracle (i.e., smoke

tests), but that their application can result in an invalid reorganization result or invalid system structure. In a preliminary evaluation that ran for about one week, we tested the SPADA and PSOPP implementation without injecting any faults. We did not observe any failures in the course of these tests. So we can be confident that the failures the oracle reported during our subsequent evaluation can be attributed to a specific injected fault.

*SPADA: Injected Faults.* The first two types of SPADA faults (cf. SPADA-F1 and SPADA-F2) manifest in an incorrect transformation of the current system structure into SPADA's internal model of a partitioning, i.e., the acquaintances graph (cf. Sect. 5.1). This false mapping results in an invalid reorganization result.

*SPADA-F1/SPADA-F2*
- *Description*: When creating the acquaintances graph for a new reorganization on the basis of the current system structure, an arbitrary AVPP is not represented in the acquaintances graph if the number of AVPPs is above (in case of SPADA-F1) or below (in case of SPADA-F2) a certain threshold. We set these thresholds to 100 for SPADA-F1 and to 5 for SPADA-F2.
- *Effect*: The resulting partitioning does not contain the power plants that have been members of the "forgotten" AVPP.

The two other types of faults we integrated into SPADA concern a functionality that is used to transform the result, given in the form of an acquaintances graph, into a set of sets. This functionality is used to provide the result to the SO Algorithm Adapter (R-Solution) and as a preprocessing step to create the new AVPP structure (R-Distribution).

*SPADA-F3*
- *Description*: In case the size of a partition exceeds a predefined threshold, arbitrary power plants are deleted from this partition until its size equals this threshold. In our evaluation, we set this threshold to 100.
- *Effect*: Some power plants are not represented in the partitioning.

*SPADA-F4*
- *Description*: In case the size of a partition exceeds a predefined threshold, this partition is replaced by a partition that is randomly selected from the partitioning. In our evaluation, we set this threshold to 100.
- *Effect*: Some power plants are not represented in the partitioning, whereas others occur two or more times.

All SPADA faults can be detected using the gray-box interface (R-Solution), i.e., before the underlying system structure is changed. Given the way the result is transformed into a new system structure (it is ensured that every power plant is always a member of exactly one AVPP), these faults *cannot* be detected using the black-box view (R-Distribution). Note that the oracle does not check the system structure with respect to the number and the size of AVPPs in case of SPADA. For each type of injected fault, we are thus confronted with the problem of error masking (C-ErrorMask).

*PSOPP: Injected Faults.* Regarding PSOPP, we modified the implementation of the move operations "random split" (PSOPP-F1), "random join" (PSOPP-F2), "approach split" (PSOPP-F3), "approach join" (PSOPP-F4), and "approach exchange" (PSOPP-F5) as described in the following listing.

*PSOPP-F1*
- *Description*: If a partition $K$ is randomly split into two partitions $L$ and $M$, an arbitrary power plant of $L$ is replaced by another arbitrary power plant of $M$. This fault does only occur if the size of $L$ and $M$ is below a threshold $t_1$ or above a threshold $t_2$.
- *Effect*: With regard to the resulting partitioning, a specific power plant is missing and another occurs twice.

*PSOPP-F2*
- *Description:* If two partitions $K$ and $L$ are merged into a new partition $M$ when applying the random join operator, either $K$ or $L$ is not removed from the partitioning. This fault does only occur if the size of $K$ and $L$ is below $t_1$ or above $t_2$.
- *Effect:* In the resulting partitioning, the power plants of either $K$ or $L$ occur twice as they are also contained in $M$.

*PSOPP-F3*
- *Description:* If a partition $K$ is split into two partitions $L$ and $M$, the resulting partitioning does not contain either $L$ or $M$. This fault does only occur if the size of $L$ and $M$ is below $t_1$ or above $t_2$.
- *Effect:* In the resulting partitioning, the power plants of either partition $L$ or $M$ are missing.

*PSOPP-F4*
- *Description*: If two partitions $K$ and $L$ are merged into a new partition $M$ when applying the approach join operator, one element is removed from $M$. This fault does only occur if the size of $K$ and $L$ is below $t_1$ or above $t_2$.
- *Effect*: In the resulting partitioning, a single power plant is missing.

*PSOPP-F5*
- *Description*: If some power plants are exchanged between two partitions $K$ and $L$, one power plant of either $K$ or $L$ occurs in both resulting partitions $M$ and $N$. This fault does only occur if the size of $M$ and $N$ is below $t_1$ or above $t_2$.
- *Effect*: In the resulting partitioning, one power plants occurs twice.

In our experiments, we used $t_1 = 2$ and $t_2 = 100$ so that the failures do only occur in certain situations. Note that the application of an injected fault does not necessarily yield an invalid result because an invalid candidate solution must be rated better than all other (possibly valid) candidate solutions found by the particles (C-ErrorMask). Clearly, this characteristic together with PSOPP's non-deterministic behavior exacerbates the detection of an injected fault (C-BranchingStateSpace).

| Injected Fault | PSOPP F1 | PSOPP F2 | PSOPP F3 | PSOPP F4 | PSOPP F5 | PSOPP F5d | SPADA F1 | SPADA F2 | SPADA F3 | SPADA F4 |
|---|---|---|---|---|---|---|---|---|---|---|
| #EP States | 11.68 (5.39) | 15.77 (6.49) | 12.30 (5.96) | 13.74 (5.75) | 15.41 (6.16) | 14.25 (6.05) | 16.42 (5.71) | 14.72 (6.42) | 16.62 (6.19) | 15.44 (6.31) |
| #EP Transitions | 127.42 (99.20) | 222.06 (148.59) | 143.31 (124.31) | 169.78 (128.68) | 210.22 (142.27) | 183.28 (130.00) | 230.83 (134.75) | 197.09 (134.29) | 240.56 (141.24) | 212.72 (136.51) |
| %EP State Coverage | 99.51 (2.33) | 99.57 (2.08) | 99.45 (2.50) | 99.42 (2.71) | 99.47 (2.35) | 99.54 (2.31) | 99.66 (1.77) | 99.59 (2.15) | 99.54 (2.35) | 99.60 (1.98) |
| %EP Transition Coverage | 52.31 (15.80) | 45.37 (12.63) | 50.57 (14.32) | 47.53 (11.97) | 44.76 (11.47) | 47.57 (14.11) | 43.98 (11.34) | 47.46 (15.11) | 43.52 (11.60) | 45.34 (13.23) |

**Table 1.** Statistical data concerning the number of EP states, EP transitions, as well as the coverage of EP states and EP transitions. All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

In principle, all types of PSOPP faults could lead to a false result that can be detected using the gray-box interface (R-Solution) as well as the black-box view (R-Distribution). Note, however, that not all invalid results manifest themselves in an invalid system structure. Consider the following example illustrating error masking (C-ErrorMask): Assume that the power plants $a$ and $b$ are currently members of the same AVPP. If a reorganization causes an invalid result that does not contain these two power plants, the oracle detects a failure using the gray-box interface. However, the resulting system structure is *valid* in case the minimum size of an AVPP is $\leq 2$ and the maximum number of AVPPs is not exceeded. This is because $a$ and $b$ simply remain in their old AVPP if they are not contained in the provided result.

### 6.2 Test Execution

To be able to make a clear statement which types of faults can be found by IsoTeSO, only one specific type was injected during the execution of a single test sequence. All in all, we injected 10 different types of faults: SPADA-F1 to SPADA-F4, PSOPP-F1 to PSOPP-F5, and an additional variant of PSOPP-F5, called PSOPP-F5d, which we will explain in more detail in the course of Sect. 6.3. For each type, we generated 70 test suites, each containing 10 test sequences, resulting in 700 test sequences per fault type and a total number of executed test sequences of 7000. Overall, we generated 3,679,326 test cases, corresponding to an average of 367,932.60 per fault type. As shown in Tab. 1, this high number of test cases allowed us to obtain an EP state coverage of more than 99% and an EP transition coverage ranging between approximately 44% and 52% on average for all fault types. To illustrate the advantages of the gray-box view, we did not abort the execution of a test sequence until the oracle found a failure using the black-box view. In the course of the execution of a single test sequence, IsoTeSO could therefore register multiple failures using the gray-box interface. On the other hand, it is not possible to detect more than one failure per test sequence using the black-box view.

### 6.3 Evaluation Results

As Tab. 2 shows, IsoTeSO was able to detect every kind of injected fault and our evaluation results support our claims made in Sect. 6.1: (1) All failures detected using the black-box view are also detected using the gray-box view, and (2) the SPADA faults cannot be detected using the black-box view. The fact that not all PSOPP faults that were disclosed using the gray-box view were also registered using the black-box view (between 0.00% and 34.44% on average) also demonstrates the problem of error masking (C-ErrorMask) in SOAS. Except for PSOPP-F4, the percentage of detected failures using the gray-box view (between 50.00% and 82.31% on average) outmatches the percentage of detected failures using the black-box view in all cases (between 31.25% and 80.00% on average). These observations clearly indicate the need for gray-box interfaces in the context of testing SO algorithms.

The relatively high number of test sequences in which no failure was detected (between 70.22% and 97.86% on average) highlights the need for directed testing that is able to deal with the huge search space in a more efficient way (C-BranchingStateSpace). For PSOPP, the different numbers of applied test cases per test sequence reflect the difficulty of disclosing a specific type of fault using the black-box view. In case of SPADA, all test cases were applied since the injected faults cannot be detected using the black-box view. Another indicator for the difficulty of finding a specific fault type is the depth of the first detected failure (i.e., the index of the test case in which the first failure was detected). Here, we see significant differences among the different SPADA and PSOPP fault types. This is because the system not only has to be pushed into a faulty intermediate state but the reorganization also has to end in a faulty state that can be detected by the oracle. This challenge becomes clearer when taking a look at the percentage of undetected failures due to error masking, which ranges between 17.69% and 97.75% for PSOPP and between 0.00% and 99.84% for SPADA. Together with the average number of reorganizations per test sequence—that, except for SPADA-F3, significantly outmatches the average number of failures per test sequence—these observations illustrate the difficulty of testing SOAS.

We observed that especially PSOPP-F5 had a relatively high number of executed test sequences without detected failures (97.86% compared to an average of 88.67% over all other types of fault). We therefore decided to use PSOPP-F5 to investigate the influence of directed testing on IsoTeSO's ability to disclose a fault. To study this effect, we introduced an additional fault type PSOPP-F5d that is equivalent to PSOPP-F5 but uses a specific system configuration: To increase the chance of applying PSOPP's approach exchange operator (an operation that is usually only applied in rare cases), we set the allowed number of partitions in PSOPP-F5d to $n_{min} = n_{max}$. Using this parametrization, PSOPP has no choice but to apply the random exchange or approach exchange operator, because the split/join operators increase/decrease the number of partitions by one. Note that this measure does not increase the code coverage (PSOPP also applied the approach exchange operator in case of PSOPP-F5), but the chance that the fault can be disclosed given the algorithm's non-deterministic behavior

| | PSOPP | | | | | | SPADA | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Injected Fault | F1 | F2 | F3 | F4 | F5 | F5d | F1 | F2 | F3 | F4 |
| #Agents | 523.53 (310.53) | 529.66 (292.23) | 473.29 (269.35) | 511.83 (287.62) | 491.26 (300.28) | 477.64 (294.48) | 520.29 (286.29) | 509.63 (289.01) | 494.90 (297.69) | 447.40 (279.09) |
| #Agent groups | 39.80 (67.18) | 37.42 (48.83) | 20.09 (27.72) | 36.05 (64.05) | 30.2 (47.79) | 32.74 (48.77) | 31.86 (57.70) | 35.12 (47.42) | 31.30 (58.27) | 38.85 (45.35) |
| %Test sequences without failure | 79.97 (40.05) | 84.55 (36.17) | 91.42 (28.03) | 96.28 (18.94) | 97.86 (14.49) | 70.22 (45.76) | 97.13 (16.69) | 84.22 (36.48) | 85.69 (35.04) | 90.13 (29.85) |
| #Test cases per test sequence | 520.01 (281.50) | 523.71 (275.73) | 531.53 (273.61) | 521.19 (278.14) | 517.23 (272.25) | 518.54 (270.26) | 548.38 (278.18) | 535.62 (271.43) | 511.46 (277.29) | 518.51 (282.27) |
| %Applied test cases per test sequence | 81.37 (38.39) | 85.69 (34.44) | 92.74 (25.62) | 96.77 (17.11) | 98.62 (11.16) | 73.90 (42.75) | 100.00 (0.00) | 100.00 (0.00) | 100.00 (0.00) | 100.00 (0.00) |
| #Reorganizations per test sequence | 73.29 (64.40) | 77.68 (62.08) | 91.87 (59.90) | 91.87 (60.24) | 90.01 (56.36) | 53.35 (61.19) | 100.00 (58.20) | 87.64 (62.46) | 83.60 (60.35) | 87.75 (61.30) |
| #Failures per test sequence | 0.36 (1.89) | 0.19 (0.53) | 0.13 (0.52) | 2.29 (16.03) | 0.02 (0.15) | 0.54 (1.03) | 0.03 (0.17) | 0.16 (0.36) | 83.49 (60.39) | 0.12 (0.35) |
| #Failures | 252 | 130 | 91 | 1603 | 16 | 376 | 20 | 110 | 58443 | 84 |
| %Undetected failures | 44.44 | 17.69 | 34.07 | 97.75 | 50.00 | 40.96 | 0.00 | 0.00 | 99.84 | 9.52 |
| %Detected failures (gray box) | 55.56 | 82.31 | 65.93 | 2.25 | 50.00 | 59.04 | 100.00 | 100.00 | 0.16 | 90.48 |
| %Detected failures (black box) | 53.17 | 80.00 | 57.14 | 2.25 | 31.25 | 51.33 | 0.00 | 0.00 | 0.00 | 0.00 |
| %Test sequences with failures detected in gray box only | 4.29 (20.33) | 3.70 (18.97) | 13.33 (34.28) | 0.00 (0.00) | 34.44 (45.63) | 9.74 (27.63) | 100.00 (0.00) | 100.00 (0.00) | 100.00 (0.00) | 100.00 (0.00) |
| %Test sequences with failures detected in black box only | 0.00 (0.00) | 0.93 (9.62) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) | 0.00 (0.00) |
| Depth of first detected failure | 8.56 (22.82) | 17.50 (62.33) | 7.55 (9.12) | 44.04 (109.78) | 30.87 (74.51) | 24.28 (75.26) | 3.85 (0.49) | 4.66 (1.64) | 9.96 (14.38) | 13.20 (16.06) |
| Depth of first detected failure (gray box) | 8.56 (22.82) | 17.63 (62.61) | 7.55 (9.12) | 44.04 (109.78) | 30.87 (74.51) | 24.28 (75.26) | 3.85 (0.49) | 4.66 (1.64) | 9.96 (14.38) | 13.20 (16.06) |
| Depth of first detected failure (black box) | 8.77 (23.30) | 13.64 (46.08) | 6.08 (5.12) | 44.04 (109.78) | 90.27 (173.32) | 22.83 (71.90) | N/A | N/A | N/A | N/A |

**Table 2.** Statistical data concerning the number of agents, the number of agent groups, the number of test cases, the number of reorganizations, as well as the occurrence, detection, and depth of failures. All undetected failures (see "%Undetected failures per test sequence") can be attributed to error masking. "#Failures" and "#Failures per test sequence" refer to the number of faulty intermediate states the corresponding SO algorithm entered (note that this information is provided by our fault injection mechanism and not by the oracle that can only check for the validity of final states, i.e., reorganization results). All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

(C-BranchingStateSpace): In approximately the same number of executed test cases, PSOPP-F5d entered a faulty intermediate state about 23 times more often than PSOPP-F5. The benefit of directed testing is further reflected in an increase of the percentage of a faulty end state in case of a faulty intermediate state from 50.00% for PSOPP-F5 to 59.04% for PSOPP-F5d. Consequently, the number of applied test sequences without detected failures dropped from 97.86% to 70.22%.

Summarizing, although IsoTeSO was able to find every type of injected fault, the high number of needed test sequences for failure detection demonstrates that the automatic generation of test suites on the basis of EPs and influence functions is especially useful for testing SOAS. This is mainly because of the non-deterministic behavior of SO algorithms (C-BranchingStateSpace). Our evaluation results also show that a combination of model-based and random generation techniques is essential to efficiently and effectively detect specific types of fault. In particular, the gray-box interface is an important feature to mitigate the effect of error masking (C-ErrorMask). However, we expect an additional white-box view to certainly further increase the potential of finding faults in SO algorithms.

## 7 Related Work

The necessity of testing adaptive systems has been recognized both in the testing community [30,32,52,49] and in the community of adaptive systems [20,35,11,26]. Run-time as well as design-time approaches have identified non-determinism and the emergent behavior as main challenges for testing adaptive systems.

Run-time approaches for testing take up the paradigm of run-time verification [17,27,18]. They shift testing into run-time to be able to observe and test, e.g., the adaptation to new situations. Camara et al. [10] are using these concepts to consider fully integrated systems. Their testing approach focuses mainly on testing non-functional properties of the system or more precisely, the resilience of the adaptive system. The authors therefore investigate the system's adaptive capabilities by collecting and analyzing data in a simulated environment. The gained information is used as feedback for the running system. A similar approach is taken by Ramirez et al. [38], also focusing on non-functional requirements. The authors use the sampled data from a simulation to calculate a distance to expected values derived from the goal specification of the system. This information is used to adapt the system or its requirements proactively during run-time. Run-time approaches, however, are limited to tests of the fully integrated system and therefore are faced with problems like error masking which is very likely in such self-healing systems. In our layered testing approach—where IsoTeSO is placed on the interaction layer—we benefit from the piecemeal integration of the system for testing. Thus, it is possible to avoid error masking within IsoTeSO by testing the SO algorithms isolated.

An important difference to the mentioned work is that we use these techniques for finding failures instead of analyzing the current system state for generating feedback for the adaptation. Still, we also use the basic concepts of run-time testing. The CEI allows us to split the evaluation into the three responsibilities R-Detect, R-Solution, and R-Distribution which in turn enable us to evaluate the runs without the evaluation of complex system states on the system level. As our evaluation shows, the CEI-based testing approach is especially beneficial in the context of self-organization.

Design-time approaches like [45,30,32,52,28] test the systems in a classical manner during the development. All these approaches are considering some ded-

icated parts of the system. Consequently, it is not possible to give evidence about the correct functionality of the overall system. Zhang et al. [52] compose their tests towards a fully integrated system test, but they do not consider adaptivity or SO explicitly since they focus on testing the correct execution of plans within multi-agent systems. Nguyen et al. [30] promote an approach for a component test suite (where the components correspond to agents), but do not consider interaction or organization between the agents as it would be necessary for SO.

The evaluation of the test results, i.e., the application of a test oracle for adaptive behavior is only considered by Fredericks et al. [20,19] and Nguyen et al. [31]. Both approaches are relying on goals reflecting the requirements of the system that are somewhat loosened in order to reflect the ever-changing environment the agents have to adapt to: The approaches mitigate the goals with the *RELAXed* approach [48] or consider soft goals that do not need to hold at all time. Consequently, the decision of the test oracle is rather fuzzy. In our approach the definition of correct and in-correct behavior is given by the CCB that enables us to decide whether a failure occurs or not.

*Probabilistic Test Models of Environmental Changes and Influences.* There are several approaches to tackle uncertainties about the expected usage of a SuT within testing models. They can be summed up under the idea of *operational profiles* [43]. The information within these test models represents the user's behavior in a probabilistic model. For this purpose, different techniques for generating and using these profiles have been provided.

Sammodi et al. [40], e.g., generate usage profiles, as they call them, by monitoring the user's interaction with the system and deriving the profiles for the observed usage afterward. One of our possibilities to establish EPs also follows this monitoring and analysis process with the difference that we are not monitoring users of the system. Instead we monitor the whole system environment which includes all influences on the SOuT. Another approach to design models of the usage is presented by Samih et al. [39]. They enrich the models by introducing capabilities in order to model variants of specific features, i.e., product features, to form test models for product line engineering. The approach made by Ehlers et al. [16] focuses on using the usage profiles for detecting anomalies in adaptive systems in order to use the information about the anomalies during the adaptation process. Besides focusing on handling user behavior, there is also some work on representing the behavior of other system components in the test model, like Popovic et al. [34]. The authors use the models for protocol testing and therefore represent valid and invalid communication between components.

Operational profiles thus are an established technique for modeling uncertain behavior—mainly of the user—for designing test models and for evaluation purposes. In IsoTeSO, we use environment profiles which are based on a similar concept to deal with the complexity of the ever-changing environment of SOAS by reducing its state space using a probabilistic approach.

*Isolated Testing of Component-Based Systems.* Our presented concepts for a framework for isolated testing of SO algorithms are related to methods and

techniques in the research area of isolated testing of component-based systems. A recent approach in this area by Thillen et al. [46] promotes the "tester in the middle" idea that aims at improving testing of distributed components depending on other components in the system. Their application area is the testing of network components. For this purpose, they model dependencies within the network to be able to build mock-ups out of this models. Bauer et al. [8] propose a statistical strategy for isolated testing of component-based systems. Their approach is based on state-based models that are used to generate interaction test models. The goal is to test the interactions and functionalities within the system under test which is composed of several system components. Yao and Wang [51] present a framework for testing distributed software components that provides an environment to allow a client-side software component to define tests for a black-box component published on the server-side. The framework focuses on automatic test execution without considering explicitly the generation and evaluation of the tests. Wu et al. [50] propose JATA, a testing language for distributed components enabling the use JUNIT in the context of service-oriented systems and offering support for a message-oriented middleware; like Yao and Wang [51] it focuses on the execution of component tests on web services. In contrast to the approaches in [51,50], IsoTeSO offers a complete approach from test generation over execution to test evaluation with a focus on agent-based systems and functional testing of single SO algorithms.

Our approach for isolated testing of SO algorithms is an efficient combination of model-based techniques using the concepts of isolated testing and probabilistic modeling in order to tackle the challenges of testing SOAS. To our knowledge, there is no approach extending both of these techniques to SO algorithms.

## 8 Conclusion and Future Work

We motivated the necessity of testing SOAS and outlined an approach that copes with the complexity arising from the characteristics of these systems. We introduced the framework IsoTeSO for testing SO algorithms that is an important element coping up with this challenge. To be able to test SO algorithms in a systematic and automatic fashion, we identified four challenges that are addressed: error masking among the algorithms of SOAS, interleaved feedback loops which distort test results, the oracle problem in the context of SO, and the ramified state space spanned by SO algorithms. We were able to show that it is possible to meet these challenges by isolating the SO algorithms and applying a model-based testing approach. The resulting framework encompasses automatic test suite generation, execution, and evaluation within a controlled environment. Environment profiles are used to abstract from concrete environments and allow for generating large test sequences. This technique showed to be extremely valuable since a lot of test runs are necessary to find a trace through a SOAS that actually reveals a failure.

We evaluated IsoTeSO on the basis of two different partitioning-based SO algorithms in an existing smart-grid application. Our results demonstrate that an

efficient combination of model-based and random test case generation techniques allows to find different kinds of failures in an acceptable time limit. In particular, the gray-box view turned out to be very important to reduce the effect of error masking.

*Future work* focuses on three aspects: first, *extending the evaluation* of the test run results enabling *fault diagnosis*; second, gaining more *control on the execution* to increase *reproducibility* of the test results; and third, using IsoTeSO to investigate non-functional aspects of the system, i.e., to evaluate the performance of SO-algorithms.

For the extension toward fault diagnosis, we aim at combining the different test sequences of a specific test suite into a *test sequence evaluation tree*. For this purpose, we will try to identify equal prefixes of the test sequences. If two or more test sequences share a common prefix, they will share a common path in the tree. Thus, it is possible to unfold overlaps between executed and evaluated test sequences and map them into a tree model. This tree could be used for fault localization, optimization of later test generations, or visualization of the results for a deeper understanding.

To increase the *reproducibility* of the test results, the test system has to ensure the following properties: (1) a random seed has to be used in the whole test system; (2) the scheduling of the underlying execution platform has to be controlled in a reproducible way; and (3) the concurrent execution of a distributed test system has to be synchronized in a deterministic way. There are different techniques to address (1) to (3) in order to detect so-called "Mandelbugs"[7] (cf. the CHESS tool by Musuvathi et al. [29] or the empirical study on this topic by Thomson et al. [47]). These could be used to provide an adequate underlying platform for IsoTeSO. Although designing such a platform was not in the scope of this paper, we met the points (1) and partially (3). As we perform our tests within the limits of the underlying execution platform (Java), which does not affect the ability to detect faults but might affect the reproducibility (at least in some cases), we currently do not meet (2) and achieve (3) only on the basis of a stepwise execution model. We are going to extend our experiments with a continuous execution model to evaluate differences in the test results and its effects on the reproducibility as well as the ability of IsoTeSO to execute tests in this environment.

Besides investigating functional tests we already started to use IsoTeSO for the evaluation of performance criteria. In [13], we showed its ability to perform evaluations on non-functional aspects of SO algorithms and developed a set of five major requirements for metrics that evaluate the performance of SO algorithm within IsoTeSO. The development of the metrics and the integration

[7] According to Grottke and Trivedi [21] a Mandelbug is "[a] fault whose activation and/or error propagation are complex, where 'complexity' can take [the following form]: [...] The activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system-internal environment of the application [...]".

into the framework is a further topic of future work in testing self-organizing, adaptive systems.

*Acknowledgment.* This research is sponsored by the research project *Testing Self-Organizing, adaptive Systems (TeSOS)* of the German Research Foundation.

## References

1. Anders, G., Siefert, F., Mair, M., Reif, W.: Proactive Guidance for Dynamic and Cooperative Resource Allocation under Uncertainties. In: Proc. 8th IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO). IEEE Computer Society (2014)
2. Anders, G., Siefert, F., Msadek, N., Kiefhaber, R., Kosak, O., Reif, W., Ungerer, T.: TEMAS – A Trust-Enabling Multi-Agent System for Open Environments. Tech. Rep. 2013-04, Universität Augsburg (2013), http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/2311
3. Anders, G., Siefert, F., Reif, W.: A Particle Swarm Optimizer for Solving the Set Partitioning Problem in the Presence of Partitioning Constraints. In: Proc. 7th Int. Conf. Agents and Artificial Intelligence (ICAART). SciTePress (2015)
4. Anders, G., Siefert, F., Steghöfer, J.P., Reif, W.: A decentralized multi-agent algorithm for the set partitioning problem. In: Rahwan, I., Wobcke, W., Sen, S., Sugawara, T. (eds.) Proc. 15th Int. Conf. Principles and Practice of Multi-Agent Systems (PRIMA), Lect. Notes Comp. Sci., vol. 7455, pp. 107–121. Springer (2012)
5. Anders, G., Steghöfer, J.P., Klejnowski, L., Wissner, M., Hammer, S., Siefert, F., Seebach, H., Bernard, Y., Reif, W., Müller-Schloer, C., André, E.: Reference Architectures for Trustworthy Energy Management, Desktop Grid Computing Applications, and Ubiquitous Display Environments. Tech. Rep. 2013-05, Universität Augsburg (2013), http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/2303
6. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Roşu, G., Sen, K., Visser, W., et al.: Combining test case generation and runtime verification. Theoretical Computer Science 336(2), 209–234 (2005)
7. Balas, E., Padberg, M.W.: Set partitioning: A survey. SIAM Review 18(4), 710–760 (1976)
8. Bauer, T., Eschbach, R.: Enabling statistical testing for component-based systems. In: Fähnrich, K.P., Franczyk, B. (eds.) GI Jahrestagung, LNI, vol. 176, pp. 357–362. GI (2010)
9. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison Wesley (1999)
10. Cámara, J., de Lemos, R.: Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: Proc. 7th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 53–62 (2012)
11. Cheng, B., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, Lect. Notes Comp. Sci., vol. 5525, pp. 1–26. Springer (2009)
12. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Reif, W.: A framework for testing self-organisation algorithms. In: 37. Treffen der GI Fachgruppe TAV. vol. 35:1. Softwaretechnik-Trends der Gesellschaft für Informatik (2015)

13. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Reif, W.: A research overview and evaluation of performance metrics for self-organization algorithms. In: Proc. 9th IEEE Int. Conf. Self-Adaptive and Self-Organizing Systems Wsh. (SASOW). pp. 122–127. IEEE Computer Society (2015)
14. Eberhardinger, B., Seebach, H., Knapp, A., Reif, W.: Towards testing self-organizing, adaptive systems. In: Merayo, M., de Oca, E. (eds.) Proc. 26th IFIP WG 6.1 Int. Conf. Testing Software and Systems (ICTSS), Lect. Notes Comp. Sci., vol. 8763, pp. 180–185. Springer (2014)
15. Eberhardinger, B., Steghöfer, J.P., Nafz, F., Reif, W.: Model-driven synthesis of monitoring infrastructure for reliable adaptive multi-agent systems. In: Proc. 24th IEEE Int. Symp. Software Reliability Engineering (ISSRE). pp. 21–30. IEEE Computer Society (2013)
16. Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proc. 8th ACM Int. Conf. Autonomic Computing (ICAC). pp. 197–200. ACM (2011)
17. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Barthe, G., et al. (eds.) Proc. 9th Int. Conf. Software Engineering and Formal Methods (SEFM). Lect. Notes Comp. Sci., vol. 7041, pp. 204–220. Springer (2011)
18. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: Continuous assurance of non-functional requirements. Formal Asp. Comp. 24(2), 163–186 (2012)
19. Fredericks, E.M., DeVries, B., Cheng, B.H.C.: Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proc. 9th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 17–26. ACM (2014)
20. Fredericks, E.M., Ramirez, A.J., Cheng, B.H.C.: Towards run-time testing of dynamic adaptive systems. In: Proc. 8th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 169–174. IEEE (2013)
21. Grottke, M., Trivedi, K.S.: A classification of software faults. Journal of Reliability Engineering Association of Japan 27(7), 425–438 (2005)
22. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for organic computing systems. In: Brueckner, S.A., Robertson, P., Bellur, U. (eds.) Proc. 2nd IEEE Int. Conf. Self-Adaptive and Self-Organizing Systems. pp. 233–242. IEEE Computer Society (2008)
23. Hierons, R.M.: Oracle for distributed testing. IEEE Trans. Softw. Eng. 38(3), 629–641 (2012)
24. Kennedy, J., Eberhart, R.: Particle Swarm Optimization. In: Proc. IEEE Int. Conf. on Neural Networks. vol. 4, pp. 1942 –1948 (1995)
25. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
26. de Lemos, R., et al.: Software engineering for self-adaptive systems: A second research roadmap. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II, Lect. Notes Comp. Sci., vol. 7475, pp. 1–32. Springer (2013)
27. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009), Proc. 1st Wsh. Formal Languages and Analysis of Contract-Oriented Software (FLACOS)
28. Luckey, M., Thanos, C., Gerth, C., Engels, G.: Multi-staged quality assurance for self-adaptive systems. In: Proc. 6th Int. Conf. Self-Adaptive and Self-Organizing Systems Wsh. (SASOW). pp. 111–118 (2012)

29. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proc. 8th USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 267–280. USENIX Association (2008)
30. Nguyen, C.D.: Testing Techniques for Software Agents. Ph.D. thesis, Università di Trento (2009)
31. Nguyen, C.D., Marchetto, A., Tonella, P.: Automated oracles: An empirical study on cost and effectiveness. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Proc. Joint Meet. Europ. Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE). pp. 136–146. ACM (2013)
32. Padgham, L., Thangarajah, J., Zhang, Z., Miller, T.: Model-based test oracle generation for automated unit testing of agent systems. IEEE Trans. Softw. Eng. 39(9), 1230–1244 (2013)
33. Pezzé, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons (2005)
34. Popovic, M., Kovacevic, J.: A statistical approach to model-based robustness testing. In: Proc. 14th IEEE Conf. and Wsh. Engineering of Computer-Based Systems (ECBS). pp. 485–494 (2007)
35. Püschel, G., Götz, S., Wilke, C., Aßmann, U.: Towards systematic model-based testing of self-adaptive software. In: Proc. 5th Int. Conf. Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE). pp. 65–70 (2013)
36. Püschel, G., Götz, S., Wilke, C., Piechnick, C., Aßmann, U.: Testing self-adaptive software: Requirement analysis and solution scheme. Int. Journal on Advances in Software 7(1 & 2), 88–100 (2014)
37. Ramchurn, S.D., Vytelingum, P., Rogers, A., Jennings, N.R.: Putting the "Smarts" into the Smart Grid: A Grand Challenge for Artificial Intelligence. Communications of the ACM 55(4), 86–97 (2012)
38. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., Knoester, D.B.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Alexander, P., et al. (eds.) Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering (ASE). pp. 568–571. IEEE (2011)
39. Samih, H., Le Guen, H., Bogusch, R., Acher, M., Baudry, B.: An approach to derive usage models variants for model-based testing. In: Merayo, M., de Oca, E. (eds.) Proc. 26th IFIP WG 6.1 Int. Conf. Testing Software and Systems (ICTSS), Lect. Notes Comp. Sci., vol. 8763, pp. 80–96. Springer (2014)
40. Sammodi, O., Metzger, A., Franch, X., Oriol, M., Marco, J., Pohl, K.: Usage-based online testing for proactive adaptation of service-based applications. In: Proc. 35th IEEE Computer Software and Applications Conf. (COMPSAC). pp. 582–587 (2011)
41. Schmeck, H., Müller-Schloer, C., Çakar, E., Mnif, M., Richter, U.: Adaptivity and self-organization in organic computing systems. ACM Trans. Auton. Adapt. Syst. 5(3) (2010)
42. Scott, P., Thiébaux, S., van den Briel, M., Van Hentenryck, P.: Residential demand response under uncertainty. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming, Lect. Notes Comp. Sci., vol. 8124, pp. 645–660. Springer (2013)
43. Smidts, C., Mutha, C., Rodríguez, M., Gerber, M.J.: Software testing with an operational profile: Op definition. ACM Comput. Surv. 46(3), 39:1–39:39 (2014)
44. Steghöfer, J.P., Anders, G., Siefert, F., Reif, W.: A System of Systems Approach to the Evolutionary Transformation of Power Management Systems. In: Proc. of

Informatik 2013 – Workshop on "Smart Grids". Lecture Notes in Informatics, Bonner Köllen Verlag (2013)
45. Stott, D.T., Floering, B., Burke, D., Kalbarczyk, Z., Iyer, R.K.: NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proc. IEEE Int. Computer Performance and Dependability Symp. (IPDS). pp. 91–100. IEEE (2000)
46. Thillen, F., Mordinyi, R., Biffl, S.: Isolated testing of software components in distributed software systems. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering, Lect. Notes in Business Information Processing, vol. 166, pp. 170–184. Springer (2014)
47. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using schedule bounding: An empirical study. SIGPLAN Not. 49(8), 15–28 (2014)
48. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: incorporating uncertainty into the specification of self-adaptive systems. In: Proc. 17$^{\text{th}}$ IEEE Int. Requirements Engineering Conf. (RE). pp. 79–88. IEEE Computer Society (2009)
49. Wotawa, F.: Adaptive autonomous systems — from the system's architecture to testing. In: Hähnle, R., et al. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, pp. 76–90. Comm. Comp. Inf. Sci., Springer (2012)
50. Wu, J., Yang, L., Luo, X.: Jata: A language for distributed component testing. In: 15$^{\text{th}}$ Asia-Pacific Software Engineering Conf. (APSEC). pp. 145–152 (2008)
51. Yao, Y., Wang, Y.: A framework for testing distributed software components. In: Proc. IEEE Conf. Electrical and Computer Engineering. pp. 1566–1569. IEEE (2005)
52. Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: Decker, et al. (eds.) Proc. 8$^{\text{th}}$ Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1333–1334. IFAAMAS (2009)