Back-to-Back Testing of Self-organization Mechanisms

Benedikt Eberhardinger, Axel Habermaier, Hella Seebach, Wolfgang Reif

Back-to-Back Testing of Self-Organization Mechanisms

Benedikt Eberhardinger, Axel Habermaier, Hella Seebach, and Wolfgang Reif

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

Abstract When developing SO mechanisms, mapping requirements to actual designs and implementations demands a lot of expertise. Among other things, it is important to define the right degree of freedom for the system that allows for self-organization. Back-to-back testing supports this hard engineering task by an adequate testing method helping to reveal failures in this design and implementation procedure. Within this paper we propose a model-based approach for back-to-back testing. The approach is built on top of the S# framework and integrated into the Visual Studio development environment, enabling the creation of executable test models with comprehensive tooling support for model debugging. By applying the concepts to a self-organizing production cell, we show how it is used to fully automatically reveal faults of a SO mechanism.

Keywords: Adaptive Systems, Self-Organization, Software Engineering, Software Testing, Quality Assurance, Back-to-Back Testing, Model-Based Testing

1 Introduction

The increasing complexity of current software systems has led to an increase of autonomy of software components that are resilient, flexible, dependable, versatile, recoverable, customizable, and self-optimizing by adapting to changes that may occur in their environments [11]. Self-organization (SO) has become a keystone in the development of autonomous systems, allowing them to adapt their behavior and structure in order to fulfill their goals under ever-changing environmental conditions. Mechanisms of SO are built on top of the concepts of classical feedback loops (cf. [9, 16]). Therefore, the environment and the components are sensed and controlled, using the feedback to adapt the behavior and/or structure of the components. Different architectural concepts were developed to engineer SO mechanisms, e.g., the *MAPE Cycle* [9] or the *Observer/Controller Architecture* [16]. As an important part of the development of SO mechanisms, testing needs to be integrated in order to achieve the required quality level of the system.

This paper presents a thorough approach for supporting the engineering of SO mechanisms by *back-to-back (BtB) testing* [19] of feedback loop-based self-organization mechanisms. In our experiences in developing SO mechanisms, mapping requirements to actual designs and implementations demands a lot of expertise. Among other things, it is important to allow the system the right degree of freedom to enable self-organization. Back-to-back testing supports this engineering task with an adequate testing method helping to reveal failures in this design and implementation procedure. In order to supply BtB testing for SO mechanisms, we are faced by the following challenges:

1. *Supplying test oracles* that are able to cope with the unbounded decision space formed by different possible configurations of the systems controlled by the SO mechanism(s) as well as the huge state space of the mechanisms themselves.
2. *Systematic test case selection* is needed since exhaustive testing is not possible due to the unbounded state space. Additionally, most SO algorithms are based on heuristics, making their behavior quite non-deterministic and their state space non-uniform. Thus, common test case selection strategies relying on structured program behavior cannot be used.
3. *Automation* of test execution and evaluation is a keystone for the success, since this is the only way to execute the large test suites.

We address these challenges in a model-based approach for BtB testing where the test model mainly consists of two parts: (1) the model of the system controlled by the SO mechanism, i.e., the environment model, and (2) the model of the intended behavior of the SO mechanism, i.e., the test model. The latter is based on our concept of the *corridor of correct behavior (CCB)* [7] that describes the intended behavior of the system as a set of constraints. The concept of the CCB is used as part of the test oracle by evaluating the constraints on the current state of the model resp. system [3].

The model of the system to be controlled by the SO mechanism is used for test case generation as well as for their execution. This is possible due the S# modeling framework used by our approach: With S#, executable model instances can be composed together with a high degree of flexibility in order to test different system configurations. Furthermore, it is possible to integrate the concrete SO mechanism(s) under test into the execution environment provided by S# and to map the mechanism's state back into the model instances for evaluation within S#. The evaluation is based on checks whether the current state matches the constraints made in the model of the intended behavior.

Overall, the following main contributions will be presented:

1. A *model-based BtB testing* concept for SO mechanisms that is fully integrated into Visual Studio.
2. An approach for *systematic test case selection* for BtB testing of SO mechanisms.
3. *Automated evaluation* of test results within our test model which is based on the concepts of the CCB.

The paper is organized as followed: The next section embeds the approach into our overall testing concept for self-organizing, adaptive systems. After the introduction of the case study (Section 3), Section 4 gives an overview of our S# modeling framework and the BtB testing model. Section 5 describes model of intended behavior of the SO mechanisms. Section 6 shows how test cases are generated and executed. Section 7 evaluates the approach. We consider related work in Section 8 and conclude in Section 9.

2 The Corridor Enforcing Infrastructure

Our approach for testing *self-organizing, adaptive systems (SOAS)*—and consequently for testing SO mechanisms—is based on the *Corridor Enforcing Infrastructure (CEI)* [3]. The CEI is an architectural pattern for SOAS using regio-central or decentralized feedback loops to monitor and control single components or small groups of components in

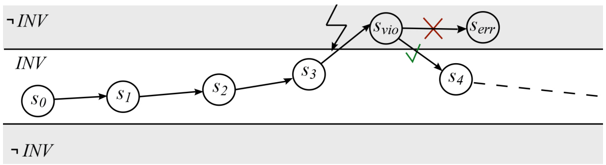

Figure 1. Schematic state-based view of the corridor of correct behavior; INV is the conjunction of all constraints of the system controlled by the CEI [3].

order to ensure that the system's goals are fulfilled at all times. The CEI implements the concepts and fundamentals of the *Restore Invariant Approach (RIA)* [7]. RIA defines the *Corridor of Correct Behavior* (CCB), which is described by the system's structural requirements, formalized as constraints. Concerning a self-organizing production cell scenario the CCB is formed by the constraints describing valid configurations of the system. The conjunction of all these constraints is called the *invariant* (INV). An exemplary corridor is shown in Fig. 1: The system's state is inside the corridor if INV is satisfied; otherwise, the system's state leaves the corridor, indicated by the flash. In that case, 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 if a transition outside of the corridor is taken, like the one marked by a cross, although a transition back into the corridor exists.

The CEI implements the RIA either with centralized or decentralized pairs of monitors and controllers, as proposed by the MAPE cycle [9] or the Observer/Controller (O/C) architecture [16]. Figure 2 shows a schematic view of one possible 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; the observer (O), i.e., the component monitoring the state of the SuOC (in- or outside of the CCB) and providing information to the controller; and the controller (C), i.e., the SO algorithms controlling the SuOC. Note that the CEI consists of sets of nested feedback loops controlling the entire system. Figure 2 further sketches the different layers for testing to cope with the complexity of the system: agent, interaction, and system layer.

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 a valid organizational structure. The concrete choice of the SO algorithms and their constraints has no impact on our approach. Since the system behaves like a traditional software system inside the CCB, traditional test approaches can be used to ensure the quality of the SuOC. The CEI, by contrast, enables self-organizing and adaptive behavior of the system and demands new concepts for testing to cope with the challenges described in Section 1.

In order to grasp SO mechanisms for testing, we need techniques to stepwise examine the CEI and its mechanisms, covering the following responsibilities of the CEI: 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 single agents or small groups of agents controlled by the CEI (R-Distribution). In this paper, we focus on revealing SO mechanism failures which relate to (R-Detect) and (R-Solution), extending our approach of isolated testing of SO algorithms presented in [2].

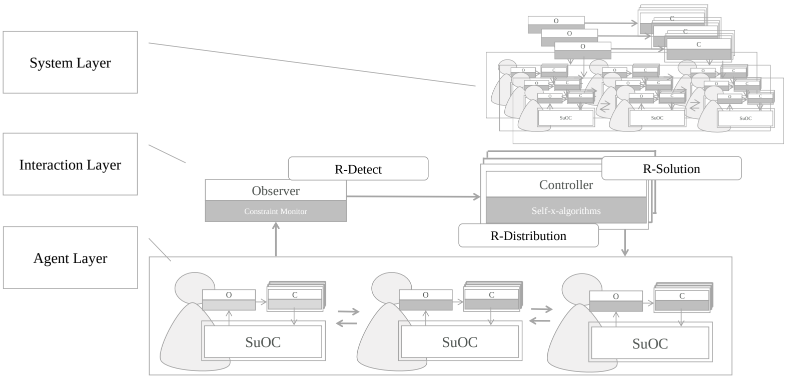

Figure 2. Schematic view of one *CEI* implementation and its different layers for testing (agent, interaction, and system layer) [2].

3 Case Study: The Self-Organizing Production Cell

Future production scenarios demand for much more flexibility [4] than today's shop floor design to cope with the trend towards small series production, individualized products and the reuse of production stations for different tasks. This flexibility becomes possible due to the increased automation and data exchange in manufacturing technologies. These future cyber-physical systems will integrate self-organization mechanisms to resolve the tasks of decentralized decision making, to optimize the systems structure, and to autonomously react to component failures at runtime increasing the system's robustness. We envision self-organizing production cells, where the production stations are modern robots equipped with toolboxes and the ability to change their tools whenever necessary (self-awareness). They are connected via mobile platforms (carts) that are able to carry workpieces and to reach robots in any order. Thus, the production cell is able to fulfill any task which corresponds to tools (capabilities) available in the cell. This is possible due to the SO mechanisms that reorganize the carts and robots in a way that the tools are applied to the workpieces in the correct order. Finding a correct allocation of tools to robots and according routes to carts (system configuration) constitutes a constraint satisfaction problem. Any violation of the calculated configuration (represents a state of the system within the CCB) at run-time triggers the SO algorithm calculating and distributing a new system configuration. A tool-supported approach to systematically model and analyze these kind of systems is shown next.

4 Building the Environment Model of SO Systems

The self-organizing production cell is an instance of the system class of self-organizing resource-flow systems; a metamodel [18] for this system class based on CEI is explained by Section 4. The case study maps to the metamodel as follows: The robots and carts are Agents monitored by the Observer/Controller. The carts transport workpieces, i.e., Resources, between the robots, which have several switchable tools, i.e., Capabilities, such

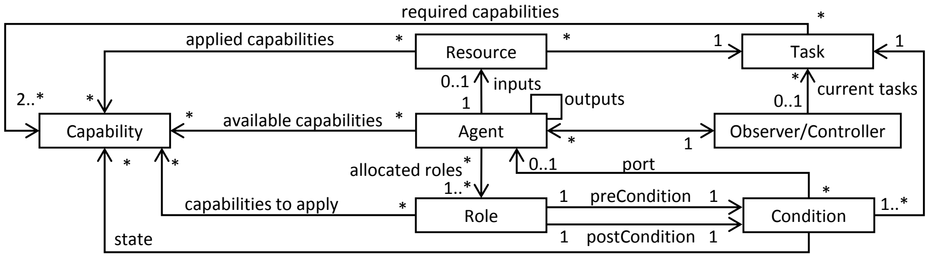

Figure 3. A UML class diagram giving a simplified overview of the metamodel for self-organizing resource-flow systems (according to [18]): Resources are passed along a set of Agents, each applying certain Capabilities in order to conduct a step towards the completion of the Resource's Task. The Observer/Controller—encompassing the SO mechanism—monitors the Agents and assigns their Roles such that all Resources are eventually fully processed with the correct order of Capability applications. Such a resource flow is specified by the Pre- and PostConditions of all Roles within the system, as well as the inputs and outputs of the Agents that establish their interconnections.

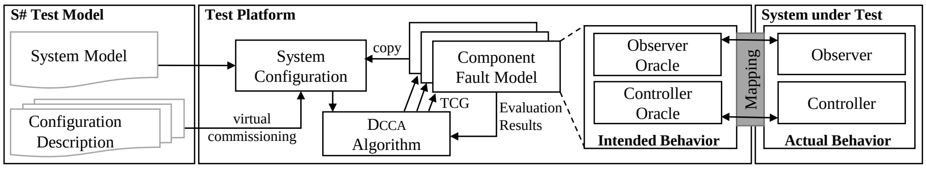

Figure 4. Our approach is divided into three parts: the first part consists of the S# test model as well as corresponding configuration descriptions (cf. Section 4). The second part is the test platform that instantiates a system configuration as the basis for the test case generation with DCCA; the component fault models represent the test cases that are evaluated (cf. Sections 4 and 6) and provide the test oracles for the observer and controller (cf. Section 5). The third part represents the actual behavior of the SO mechanism which must be mapped to the intended behavior after each execution for evaluation purposes (cf. Section 6).

as drills and screwdrivers that they use on the workpieces. A Task requires a workpiece to be processed by a sequence of tool applications, e.g., by applying the drill, insert, and tighten Capabilities. Therefore, the robots and carts are responsible for processing incoming workpieces in a given sequence of tool applications. The Roles assigned to each robot and cart indicate which tools they apply on the workpieces or which robots the resources are transported between, respectively. The Observer/Controller forms the SO mechanism of the system; it is responsible for reconfiguration in order to compensate for broken tools, blocked routes, or to incorporate new tools, robots, or carts, for instance.

The case study is modeled using the S# modeling and analysis framework for safety-critical systems [8]. As its modeling language is based on the C# programming language, the metamodel shown in Section 4 can be directly represented by a set of C# classes, with two additional classes `RobotAgent` and `CartAgent` derived from `Agent` encapsulating the production cell-specific parts of the model. Even though the S# model of the case study is represented by a C# program, it is still a model and not

```
CreateWorkpieces(5, produce(), drill(), insert(), tighten(), polish(), consume());
CreateRobot(produce());
CreateRobot(insert());
CreateRobot(tighten(), polish(), tighten(), drill());
CreateRobot(polish(), consume());
CreateCart(new Route(Robots[0], Robots[1]), new Route(Robots[0], Robots[3]));
CreateCart(new Route(Robots[2], Robots[3]));
```

Listing 1. Parts of the S# instantiation code for a configuration of the case study consisting of five workpieces that require the task PDITPC to be carried out on them. Four robots are created with some minor redundancy in available capabilities. The two carts connect all four robots via bidirectional routes.

the actual implementation; for instance, it completely abstracts from any distribution concerns, executing all modeled agents locally to simplify modeling and analysis. S# also allows the composition of a model to be automated: Arbitrary C# code can be executed to instantiate system components and to connect them together in order to build up the overall model, thereby providing meta-constructs for model creation. These capabilities are particularly useful for the creation of different system configurations when testing the case study; model instantiation with S# is illustrated by Listing 1.

S# executes the models as regular C# programs, taking care of potential non-determinism in the models such that all combinations of non-deterministic choices are fully analyzed. It is also possible to spawn additional processes during model checking, enabling the integration of other tools into the models and the analyses: For the case study, for example, a constraint solver is used to model the SO algorithm within the Observer/Controller. Whenever a reconfiguration is required, the S# model encodes the current system configuration for the constraint solver, requests a solution from it, and applies the returned solution back onto itself.

5 The Test Model for the Intended Behavior of the SO Mechanisms

S# integrates, as shown in Section 4, the complete testing framework, including test cases derivation, test cases execution, as well as test case evaluation and logging. In order to enable the evaluation of test cases, the overall model needs to be extended by the test model. The extension encompasses a definition of the intended behavior of the system under test within the S# model, as shown in the right part of Section 4. Within the BtB testing approach, we propose this one important step: to co-develop the intended behavior of the system and check it against the actual behavior. Thus, the co-development of the intended behavior is used in order to check whether it corresponds to the actual behavior. The intended behavior is modeled in two parts, consisting of (1) a description of valid system states, i.e., the INV of the CCB for the SO mechanism, as well as further constraints concerning the form of the results of the SO algorithms itself and (2) an evaluation mechanism that is able to state whether there is a possible configuration for the current system state in order to spot whether the SO algorithm missed a valid solution.

Valid system states are determined using one of the major advantages of the CCB for testing SOAS: *INV* is a way to distinguish between correct and incorrect actions of SO mechanisms—as described in Section 2. A failure occurs if a violation of the CCB is not

```
agent.Constraints = new List<Func<bool>>() {
   // I/O Consistency
   () => agent.AllocatedRoles.All(role => role.PreCondition.Port == null  ||
       agent.Inputs.Contains(role.PreCondition.Port)),
   () => agent.AllocatedRoles.All(role => role.PostCondition.Port == null ||
       agent.Outputs.Contains(role.PostCondition.Port)),
   // Capability Consistency
   () => agent.AllocatedRoles.All(
             role => role.CapabilitiesToApply.All(capability =>
                 agent.AvailableCapabilites.Contains(capability))),
   /* ... */
}
```

Listing 2. Partial S# model representing a subset of the constraints defined for the oracle.

detected (R-Detect), the computed solution does not lead to a system configuration inside the CCB (R-Solution), or a correct solution is distributed incorrectly (R-Distribution). The approach of this paper focuses on the first two aspects. As a basis for the evaluation, the constraints that form the CCB are developed separately—an important step of co-development in BtB testing—and integrated into the model. Listing 2 shows how parts of the production cell case study's CCB constraints are specified in the S# model.

The constraints that form the oracle are divided into two parts, one for the observer and one for the controller (cf. Section 4). The observer part describes all violations of the CCB that have to be detected by the observer.The constraints of the observer oracle are evaluated after the observer decided whether to reconfigure or not and the oracle judges this decision. Afterward, the controller might be activated—in case of an activation by the observer—and the result is evaluated by the controller oracle. Note that for both evaluations, the mapping between the actual SO mechanism has to be establish with the test system, i.e., the results need to be interfaced. For the evaluation of the controller a set of the constraints needs to be evaluated that is part of the controller oracle. In most cases there are overlaps between the two constraint sets, however, mostly the set of the controller oracle is a superset of the constraints of the observer oracle, however, it also might be vice versa. The additional constraints in the oracles might be due to the fact that additional requirements are necessary to fully check the results of the different parts of the SO mechanism. In our case study, this is the case for the controller oracle. The additional constraint concern the assigned roles for the robots and carts: they must be connected in the correct order for any task after reorganization so that they are applied the right way (for instance, drill, then insert, then tighten). This constraint would not be checked by the observer, since no environmental influence would change the role definitions; only the tools within the roles are affected, for instance.

The satisfiability check of the oracle focuses on another obligation for the SO algorithm: if a solution for a new system configuration is feasible on the current system instance, the SO algorithm must find it in order to find a valid configuration for the running system. If we do not check that second part of the solution we would neglect faults resulting from too strict restrictions made as a design decision in the development. Indeed, in the BtB testing approach, we aim at revealing such errors. For this purpose, it is necessary to search in the configuration space for possible configuration(s) of the system that fulfill all requirements resp. constraints considered in the previous paragraph. We use a search algorithm that systematically checks every given configuration for validity; if one is found then a solution is possible and the SO algorithm has to find it. Section 5, for

Algorithm 1 Checks whether a reconfiguration is possible for a given set of robot and cart agents as well as the tasks to be carried out.

```
Input: robotAgents, cartAgents, tasks
Output: a Boolean value indicating whether a reconfiguration is possible
$m \leftarrow$ GetConnectionMatrix(robotAgents) // transitive closure of all connected robots
for all $t \in$ tasks do
    if $\neg\forall c \in t$.Capabilities: $\exists a \in$robotAgents: $c \in a$.AvailableCapabilities then
        return false
    end if
    $A \leftarrow \{a \in$ robotAgents $\mid t$.Capabilities[0] $\in a$.AvailableCapabilities $\}$
    for $i = 0$ to |task.Capabilities| $-1$ do
        $A \leftarrow \{a \in m[a'] \mid a' \in A \wedge t$.Capabilities[i + 1] $\in a$.AvailableCapabilities $\}$
        if $|A| = 0$ then
            return false
        end if
    end for
end for
return true
```

instance, shows a search algorithm that evaluates whether a reconfiguration is possible for the case study of the production cell: It starts by checking whether all capabilities needed for the tasks that should be applied in the system are available, e.g., if a task requires drilling at least one robot must be able to drill. If that prerequisite is satisfied, the algorithm checks whether the robots with the necessary capabilities are connected by carts such that workpieces can be transported between them in way that the tasks can be fulfilled. Such an algorithm, if one exists at all, might be expensive in time and space. But this is acceptable due to the following facts: the check has to be performed only occasionally during the testing process as it is only executed when no solution is found by the SO algorithm; when the SO algorithm cannot find a solution, the configuration space is small in most cases.

6 Generating and Executing Test Cases with S#

A necessary prerequisite for deriving and executing test cases is to instantiate the model with a concrete configuration, e.g., the numbers and kinds of robots in the production cell. Within one such configuration the number of different test cases are determined by the different possible environmental events the system has to adapt to. Since the number of different configurations is unbounded, a concrete configuration is chosen for testing. Subsequently, test cases for a chosen configuration are defined by triggering environmental events that are modeled as S# *component faults*, e.g., a tool breaks, a path for carts get blocked, and so on. All of these events result in reconfigurations, i.e., executions of the SO mechanism. The huge number of configurations and component faults make exhaustive testing impossible, thus, we follow a two-part test selection strategy. d latter adding the concrete observer to a tested SO algorithm.

6.1 Test Case Generation for SO Mechanisms

The test strategy we purpose is based on the ideas of virtual commissioning and boundary interior testing. On the one hand, we only consider one concrete configuration and use the concepts of *virtual commissioning* to check other configurations on demand; on the other hand, the concepts of *boundary interior testing* are applied to SO mechanisms to find relevant test cases more quickly.

Virtual Commissioning of SOAS Systems for Test Case Reduction The concept of virtual commissioning is mainly applied in the field of large manufacturing systems where a virtual manufacturing system is built in order to simulate individual manufacturing processes for optimization and validation purposes [10]. Within this virtual environment, the real controller is executed on the virtual plant enabling to test, tune, or initialize it for a specific configuration of the plant. We adopt this concept for the reduction of possible configurations of the system to be tested. The idea is to base the tests on only one configuration, namely, the one which should be rolled out afterward. Indeed, there will be changes at run-time, e.g., new robots are integrated, new tools are added, or tasks change. Before such a change is rolled out to the running system, the model instance must first be updated and the tests have to be re-run on the new instance. Since the change of the current configuration of the system is due to a human intervention—we assume the system not to extend itself by other components or similar—it is possible to run this test-first-deploy-after strategy at run-time. Thus, we select only the configuration for testing that is crucial for the deployment and have the ability to test new configurations on demand. This is possible due to the generic S# test model in which it is easy to instantiate new configurations (cf. Listing 1) and to automatize the testing process.

Boundary Interior Testing for SO Mechanisms For test case selection within one configuration of the system we adopt the concepts of boundary interior testing, where the idea is to select test cases at the boundary of expected behavior changes. The boundaries of SO mechanisms are states of the system where reconfiguration is rarely possible, i.e., where only few solutions are still possible. Reconfigurations, as we consider them, are mainly driven by changing environmental conditions that force the system to reorganize itself. In our test model, we define these changing conditions as *component faults* of the controlled system such as a robot being unable to apply its tools in the production cell case study. The component faults are part of the S# model.

In order to find the component faults that bring the system to its boundaries, we use *Deductive Cause-Consequence Analysis (*DCCA*)*. DCCA is a fully automated model-based analysis techniques integrated into S#, usually used to assess the system's safety by computing all minimal cut sets for a hazard [8]. Minimal cut sets are combinations of component faults that can cause a hazard, characterizing a cause-consequence relationship between component faults (the causes) and the hazard (the consequence): a set of component faults is a cut set for a hazard if and only if there is the possibility that the hazard occurs and before that, at most the faults in the fault set have occurred. DCCA has exponential complexity as it has to check all combinations of component faults. In practice, however, the number of required checks usually is significantly lower, as the cut set property is monotonic with respect to set inclusion. DCCA can also be used to compute the boundaries of SO mechanisms: the hazard is simply defined as the inability for further reconfigurations. To compute the boundaries, the combinations of component faults are checked in order to determine whether such a set does or does not have the potential to cause that hazard. DCCA automatically chooses the next set of component faults to be activated, i.e., the next test case, and executes it.S#'s standard approach for DCCA checks the fault sets by increasing cardinality, thus the approach also includes test cases for inner boundary tests. The concepts of boundary interior testing for SO mechanism is exemplified in Section 6.1. The left box refers to a naïve approach where test cases are selected in a depth-first attempt and the boundaries of the SO mechanisms are not taken into account. That implies that many negative test cases are executed where less faults are expected to be revealed; in our evaluation no fault has been detected by these negative tests. The middle box of Section 6.1 is representing DCCA for boundary interior testing and covers the interiors and boundaries. The right box shows an extension that is currently under development and evaluation where only the boundaries are considered by conducting DCCA with different heuristics for selecting sets of component faults. They optimize the search of the boundaries by selecting component fault sets first where more faults of the same kind are activated and subsumption relations between component faults are exploited.

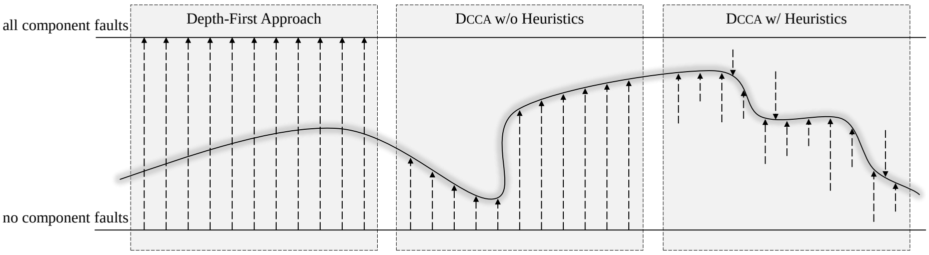

Figure 5. The three boxes represent different test case generation strategies. The x-axises of the graphic shows different system states for a particular SO mechanism under test that are formed by the possible different configurations and settings of the system under control. The boundaries of the SO mechanism relative to the number of component faults activated is shown by the black line. The idea of boundary interior testing for SO mechanisms is to stay inside and at the boundaries for testing. The left box represents a naïve depth-first search whereas the middle box shows the boundary interior approach with the standard DCCA and the right box it extension by heuristics.

6.2 Test Case Execution with S#

In order to achieve significant results, we advocate to integrate different parts of the SO mechanism step-wise: first isolate the SO algorithms, which form the controller, and afterward hook up the observer part. That enables to assign possible failures to the different parts, e.g., if a test suite is re-run after it has passed for the SO algorithm with a hooked up observer, the failure is most likely due to a faulty observer. In order to unhook the observer, the controller is triggered after every execution step, causing the system to continuously reconfigure itself. This leads also to a reduction of failure overlapping due to a missing activation of the controller by the observer. Since S# models are fully executable, the generation and selection of test cases, their execution, and their evaluation are automatically performed together. Additionally, it is also possible to manually execute or re-execute given test cases. The integration of S# into Visual Studio lets the development and execution of the test model benefit from the whole tool support of Visual Studio, e.g., the debugger. Thus, it is possible to step through every test case to monitor and control the execution and the state changes, making fault localization and test model development much easier.

7 Evaluation

For our evaluation—whose implementation is fully available at `http://safetysharp.isse.de`—we addressed the following four research questions:

R1 Is the proposed approach for testing SO mechanisms applicable to real scenarios?

R2 Is the approach able to reveal failures in SO mechanisms?

R3 Do real faults reflect the ideas proposed and exploited for test case generation (i.e., occur faults at the boundaries of the SO mechanism?)?

R4 Does the mechanism for reaching these boundaries outperform a naïve approach?

For the evaluation, we co-developed a SO mechanism for the self-organizing production cell, described in Section 3. However, the test cases and constraints are applicable to the whole system class of self-organizing resource-flow systems and in particular, the concrete implementation of the SO algorithm can be replaced by any other implementation. In the case at hand, we used MiniZinc[1] as a constraint modeling language with *FlatZinc* as the low-level solver. The system constraints have therefore been translated into a MiniZinc model that describes valid configurations for the production cell; the MiniZinc input for a system configuration is shown in Listing 3. Thus, it is possible to feed the SO algorithm with a specification of a task, the number of agents (carts and robots), the capabilities, and the routing table. If satisfiable, the SO algorithm returns a solution that assigns each tool needed for the task to some robot and that routes the carts between the robots accordingly. This SO algorithm has been plugged into S# via an interface that provides the specification of the problem to be solved by the SO algorithm and that parses MiniZinc results. The constraints of the observer of the SO mechanism—originally developed in *Java* for our implementation of the production cell based on the multi-agent system *Jadex*[2]—have been converted to C# in order to integrate them into the S# model. This completes the integration of the developed SO mechanism into the S# model and shows that real scenarios are realizable with our approach *(R1)*.

For evaluation purposes, we analyzed different configurations (cf. Table 1) of the production cell. The configurations differ in the number of agents (robots and carts), the average number of capabilities per robot, the number of tasks, and the number of routes established by the carts between the robots.

One main achievement of the evaluation is that we were able to reveal the following faults with the implementation of the SO mechanism; each fault is annotated with the responsibility of the SO mechanism where the fault was detected (cf. Section 2):

[1] `http://www.minizinc.org/`

[2] `http://www.activecomponents.org/`

```
task = [1,2,3,4,5,6]; noAgents = 6;
capabilities = [{1},{3},{4,5,4,2},{5,6},{},{}];
isConnected = [|true,false,false,false,true,false
               |false,true,false,false,true,false
               |false,false,true,false,false,true
               |false,false,false,true,true,true
               |true,true,false,true,true,false
               |false,false,true,true,false,true|]
```

Listing 3. The input model for MiniZinc describing a task, the available capabilities of the robots, as well as the connection matrix based on the carts' routes, corresponding to the configuration instantiated by Listing 1.

F1 The fault affected route handling: the MiniZinc implementation interpreted transitive routes as direct ones. Its computed configurations included direct connections that were not available, e.g. $0 \rightarrow 2 \neq 0 \rightarrow 1 \rightarrow 2$ (R-Solution).

F2 The fault was that the SO algorithm expected the routes to be unidirectional while they were in fact bidirectional. The failure manifested itself as overlooked solutions even though at least one existed (R-Solution).

F3 The fault was a wrong implementation of the interface for the SO algorithm. The interface expected first the capability of a designated agent, but got the first capability of the task assigned to the designated agent (R-Solution).

F4 The fault was a wrong format for the mapping of the solution from the SO algorithm to the system model concerning the pre- and postconditions of a role (R-Distribution). The pre-/postconditions contained the state of the workpiece in form of the remaining part of the task, e.g., for task $[D, I, T]$ the precondition contained $[D, I, T]$ and the postcondition $[I, T]$ if D had been performed. But the mapping should lead to states of the workpiece representing the part of the task which already had been done, e.g., for task $[D, I, T]$ the precondition should contain $[]$ and the postcondition $[D]$ if D had been performed (R-Distribution). This fault was detected even though the testing approach was initially not focused on R-Distribution.

F5 The fault was a too narrow restriction in the SO algorithm that did not allow to use intermediate robots that apply no tools since the maximum length of concatenated roles was restricted. Thus, Listing 3 was mistakenly considered to be *unsatisfiable* instead of returning the following solution, for instance: agents = $[1, 5, 4, 6, 3, 6, 4, 5, 2, 5, 4, 6, 3, 3, 6, 4]$; workedOn = $[1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0, 4, 5, 0, 6]$ (R-Solution).

F6 The fault was a missing constraint with the observer, namely the I/O-Consistency constraint checked in the oracle of Listing 2. The failures occurred after activating a component fault that deactivates a cart that is part of the active task (R-Detection).

The faults F3, F4, and F6 have been detected in all investigated configurations. Indeed, F1, F2, and F5 mainly depend on the routing structure used in the configuration, e.g., smaller configurations would not be able to reveal the faults. F6 mainly depends on changing the active robots or carts of a task, since their removal might not be detected and the controller is consequently not activated. All detected faults mainly concern misinterpretation of requirement specifications. The kind of faults that we detected underpins one of the strengths of our approach: the ability to reveal faults which are the result of a misinterpretation of the specification *(R2)*.

#robots	#carts	#capabilities per Robot	#capabilities per Task	#routes	#test cases	time (in min)
4	3	2.75	6	6	131,000	570
3	2	1.67	5	4	49	0.2
3	2	3.67	5	4	26,763	69.25
3	2	1.67	5	6	157	0.78
3	2	1.67	8	4	47	0.38
5	2	1.6	5	5	1,577	6.88
3	4	1.67	5	5	369	1.08

Table 1. Statistical data concerning the configuration used in the evaluation, the number of test cases generated and executed, the demanded time. Note that all detected faults have been removed and the time is used for complete testing of the interior and boundaries. Note that the runs within our framework are deterministic, i.e., there is no need to consider mean values or standard derivations.

To answer *R3* and *R4*, we focused on the performance of test case generation and execution, investigating the abilities of the boundary interior testing approach for SO mechanisms. The results concerning the failures revealed, especially F1, F2, and F5, that the failures are more likely occur on the boundaries where SO switches between being possible and impossible; e.g., F1 was revealed when only one possible routing was left to fulfill the task, while F2 was revealed when no more routing is possible for the task. For *R4*, we used a test case generation algorithm using a *depth-first search* strategy that systematically explores the input space without respecting the boundaries of SO, unlike our proposed approach. The overall testing times required by the test system to reveal the failures and the number of test cases used is measured in Table 2.

At a first glance, the results indicate that the proposed approach for test case generation does not payoff as expected in most cases. That is mainly an effect of the kinds of faults we detected in the SO mechanisms which are able to be revealed with quite a lot different combinations of component faults and thus detected very early on. However, for F5 the potential of the approach especially for a bit more sophisticated faults is shown. Furthermore, it is even possible to optimize the concepts based on how the DCCA is used for reaching the boundaries. Currently, DCCA is applying a kind of depth-first search towards the states where no reconfiguration is possible anymore. Within this search, DCCA further performs optimizations according to the activation of the component faults based on monotonicity of the cut set property.

Metrics	Boundary-interior						Depth-first					
	F1	F2	F3	F4	F5	F6	F1	F2	F3	F4	F5	F6
#Test Cases	13	15	1	1	1,609	4	10	16	1	1	16,813	5
Time (in s)	3.12	3.91	< 1	< 1	420	0.96	2.25	3.11	< 1	< 1	7583.33	1.13

Table 2. Statistical data comparing boundary-interior testing of SO mechanisms (DCCA w/o heuristics) with a simple depth-first search. The configuration that is compared is the first one of Table 1.

8 Related Work

The approach for testing adaptive system could be clustered into run-time and design-time approaches; both have identified non-determinism and the emergent behavior as the main challenges for testing adaptive systems.

Run-time approaches for testing take up the paradigm of run-time verification [5, 12]. They shift testing into run-time to be able to observe and test, e.g., the adaptation to new situations. Camara et al. [1] use these concepts to consider fully integrated systems. Their testing approach focuses mainly on testing the non-functional properties of resilience of the adaptive system. The gained information is used as feedback for the running system. A similar approach is taken by Ramirez et al. [15], also focusing on non-functional requirements. The authors use the sampled data from a simulation to calculate a distance to the expected values derived from the goal specification of the system. This information is subsequently used to adapt the system or its requirements proactively. The run-time approaches 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 testing approach, by contrast, we benefit from the piecemeal integration of the system for testing. Thus, it is possible to avoid error masking by testing the SO mechanism in an isolated way. Another important difference to the aforementioned work is that we use these techniques for finding failures instead of analyzing the current system state for generating feedback for adaptation. Still, we also use the basic concepts of run-time testing. The CEI allows us to split the evaluation into the three responsibilities of R-Detection, R-Solution, and R-Distribution which in turn enables us to evaluate the runs without the evaluation of complex system states on the system level.

Design-time approaches like [13, 20] test the systems in a classical manner during development. All of these approaches consider some dedicated parts of the system. Consequently, it is not possible to give evidence about the correct functionality of the overall system. Zhang et al. [20] compose their tests towards fully integrated system tests, but they do not consider adaptivity or SO explicitly since they focus on testing the correct execution of plans within multi-agent systems. Nguyen [13] promotes an approach for a component test suite, but does not consider interactions between or organization of components 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. [6] and Nguyen et al. [14]. Both approaches rely on goals reflecting the requirements of the system that are somewhat loosened in order to reflect the ever-changing environment the components have to adapt to: The approaches mitigate the goals with the *RELAXed* approach or consider soft goals that do not need to hold at all times. Consequently, the decisions of the test oracle are rather fuzzy. In our approach, the definition of correct and incorrect behavior is given by the CCB that enables us to clearly decide whether a failure indeed occured.

Back-to-back testing was initially proposed by Vouk [19] and describes the concept of the co-development of a test framework and the actual system or mechanisms based on the same requirements, letting the systems compete with each other in order to reveal discrepancies and errors. Back-to-back testing is focused on functional testing of the system with a special attention on the correct interpretation of the actual requirements and their implementation. The assumption made is that two different developers resp. development teams will not make the same mistake twice, i.e., misinterpret or neglect functional requirements, and so the discrepancies between the two systems reveal potential development errors. In [17], we already showed how BtB testing could be successfully applied to *constraint programming*, since our basic ideas of testing adaptive, self-organizing system is based on constraining the SO algorithms. This paper extends these concepts from constraint programming to SO mechanisms.

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

9 Conclusion & Outlook

We motivated the need for systematic testing of adaptive, self-organizing systems and purposed a systematic approach for BtB testing of SO mechanisms. The concept of BtB testing supports the challenging task of engineering SO mechanisms in a co-development manner and is able to reveal different kinds of faults concerning the functional specification of the system. The evaluation showed the utility of the approach by revealing different faults within a real development endeavor. The model-based approach presented is built upon a model of the system and its intended behavior, with the latter being based on our concepts of the CCB that enables fully automated evaluation of test runs. The test cases to be executed are derived on the basis of the system model; the test case selection strategy is based on ideas of virtual commissioning and boundary interior testing. Test case generation, execution, evaluation, and logging is fully automated and proved to be able to reveal different failures, as shown in the evaluation. The integration in the S# modeling framework allows to use our BtB testing concepts within Visual Studio, enabling model refactoring and debugging, among others.

Future work includes, among other things, the enhancement of heuristics in test case generation, enhancing the fault diagnostics, and integrating the approach into our overall framework for testing SOAS. The *heuristics for test case generation* should allow to reach the boundaries of SO mechanisms more efficiently. A first concept might be to start with bigger initial sets of component faults, e.g., to activate the component faults for all drills of all robots except of one. This leads, in a first evaluation, to better converge towards the boundaries of SO mechanisms and should reveal failures with less testing effort. *Fault diagnostic* is already possible in sense that we are able to track faults back to a part of the SO mechanism as well as to a set of activated component faults and a system configuration. However, the non-deterministic behavior of the SO mechanisms is still a challenge that we are going to address in future research. At last, the approach needs to be *integrated into an overall approach for testing SOAS* to supply a complete framework for testing the class of self-organizing resource flow systems.

Acknowledgments. This research is sponsored by the research project *Testing Self-Organizing, adaptive Systems (TeSOS)* of the German Research Foundation. Additionally, we thank our college Alexander Schiendorfer for his support with MiniZinc.

References

1. 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)
2. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Knapp, A., Reif, W.: An approach for isolated testing of self-organization algorithms. In: Software Eng. for Self-Adaptive Systems III, LNCS, vol. 9640. Springer (2016)
3. Eberhardinger, B., Seebach, H., Knapp, A., Reif, W.: Towards testing self-organizing, adaptive systems. In: Proc. 26th IFIP Int. Conf. Testing Software and Systems (ICTSS), LNCS, vol. 8763, pp. 180–185. Springer (2014)
4. ElMaraghy, H., Monostori, L.: Variety management in manufacturing cyber-physical production systems: Roots, expectations and r&d challenges. Procedia CIRP 17, 9 – 13 (2014)
5. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Proc. 9th Int. Conf. Software Engineering and Formal Methods (SEFM). LNCS, vol. 7041, pp. 204–220. Springer (2011)
6. 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)
7. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for organic computing systems. In: Proc. 2nd IEEE Int. Conf. Self-Adaptive and Self-Organizing Systems (SASO). pp. 233–242 (2008)
8. Habermaier, A., Eberhardinger, B., Seebach, H., Leupolz, J., Reif, W.: Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#. In: Proc. 9th IEEE Int. Self-Adaptive and Self-Organizing Systems Workshops (SASOW). pp. 128–133 (2015)
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1) (2003)
10. Lee, C.G., Park, S.C.: Survey on the virtual commissioning of manufacturing systems. J. Computational Design and Engineering 1(3), 213–222 (2014)
11. de Lemos, R., et al.: Software Engineering for Self-Adaptive Systems II, chap. Software Eng. for Self-Adaptive Systems: A Second Research Roadmap, pp. 1–32. Springer (2013)
12. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
13. Nguyen, C.D.: Testing Techniques for Software Agents. Ph.D. thesis, Uni. di Trento (2009)
14. Nguyen, C.D., Marchetto, A., Tonella, P.: Automated oracles: An empirical study on cost and effectiveness. In: Proc. Joint Meet. Europ. Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE). pp. 136–146. ACM (2013)
15. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., Knoester, D.B.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering (ASE). pp. 568–571. IEEE (2011)
16. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: Informatik 2006 (2006)
17. Schiendorfer, A., Eberhardinger, B., Reif, W., André, E.: Back-to-Back Testing a Soft Constraint Model for a Smart Exhibition Space. In: Proc. 14th Int. Wsh. Constraint Modelling and Reformulation (ModRef) (2015)
18. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: How to Design and Implement Self-organising Resource-Flow Systems, pp. 145–161. Springer (2011)
19. Vouk, M.A.: Back-to-back testing. Inf. Softw. Technol. 32(1), 34–45 (1990)
20. Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: Proc. 8th Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS). pp. 1333–1334 (2009)