Formal modeling and verification of self-* systems based on observer/controller-architectures

Florian Nafz, Jan-Philipp Steghöfer, Hella Seebach, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Nafz, Florian, Jan-Philipp Steghöfer, Hella Seebach, and Wolfgang Reif. 2013. "Formal modeling and verification of self-* systems based on observer/controller-architectures." In *Assurances for self-adaptive systems: principles, models, and techniques*, edited by Javier Cámara, Rogério Lemos, Carlo Ghezzi, and Antónia Lopes, 80–111. Berlin: Springer. https://doi.org/10.1007/978-3-642-36249-1_4.

Formal Modeling and Verification of Self-* Systems Based on Observer/Controller-Architectures

Florian Nafz, Jan-Philipp Steghöfer, Hella Seebach, and Wolfgang Reif

Institute for Software & Systems Engineering,
University of Augsburg, 86135 Augsburg, Germany
{nafz,steghoefer,seebach,reif}@informatik.uni-augsburg.de

Abstract. Self-* systems have the ability to adapt to a changing environment and to compensate component failures by reorganizing themselves. However, as these systems make autonomous decisions, their behavior is hard to predict. Without behavioral guarantees their acceptance, especially in safety critical applications, is arguable. This chapter presents a rigorous specification and verification approach for self-* systems that allows giving behavioral guarantees despite of the unpredictability of self-* properties. It is based on the Restore Invariant Approach that allows the developer to define a corridor of correct behavior in which the system shows the expected properties.

The approach defines relies (behavior the components can expect) and guarantees (behavior that each component will provide) to specify the general requirements on the interaction between the components of the system on a formal basis. If heterogeneous multi-agent systems with self-* properties are modeled so that relies are implied by the other components' guarantees, it is possible to formally verify correct system behavior. When using observer/controller architectures the approach also allows systematic decomposition and modular verification. We illustrate the approach by applying it to two different case studies – an adaptive production cell and autonomous virtual power plants.

Keywords: Adaptive Systems, Self-* Properties, Formal Methods, Verification, Multi-Agent Systems, Observer/Controller.

1 From Design Time to Runtime

Adaptive systems are not yet the silver bullet they are often hyped to be. It turns out that attempts to manage the complexity of modern cyber-physical systems or large-scale IT-systems often introduce a lot of complexity. While this might make the surrounding infrastructure simpler, e.g., by decreasing the number of administrators required to supervise a server farm, the transparency and controllability of the systems are reduced and thus is their trustworthiness. It is obvious that systems in which decisions are willingly relegated from design time to runtime pose many new challenges to regulators, standardization committees, and certification authorities, especially with regard to deployment of self-* systems in safety- and mission-critical domains.

One of the main tools in the certification of safety-critical systems are formal methods. A thorough formal analysis of a computer system can reveal flaws and bugs that

are not identifiable by validation techniques such as testing. However, formal methods usually rely on sophisticated models of the system and its individual components. If the system is open and not all components are known at design time, it is impossible to create such a comprehensive model. Additionally, it is quite difficult to grasp the complex and diverse interactions that can occur in an open, adaptive system and it is even more difficult to verify all possible cases of interleaved communications.

If, however, it were possible to specify the external behavior of each system component in an abstract fashion and show that the individual components do not interfere with each other, internal models could be discarded while the behavior of the ensemble could still be verified. The rely/guarantee (R/G) paradigm first introduced by Jones in [21] and Misra and Chandy in [26] provides such a theoretical framework. It allows specifying guarantees provided by the components if they can rely on properties guaranteed by the environment or other components. This allows integrating arbitrary components without knowledge of their internal behavior, a major difference to most of the related approaches that are outlined in Sect. 2. The R/G paradigm is also ideally suited to capture the modularity of a system that can be decomposed into several types of components. We use this ability to decompose the system into a functional part and a part incorporating the self-* intelligence, represented by an observer/controller (o/c) [34]. This observer/controller architectural pattern encapsulates a feedback loop and is similar to the MAPE cycle [20] in the field of Autonomic Computing [28].

This chapter presents an integrated approach that enables the engineer to verify functional properties and thus give behavioral guarantees during design time without restricting the flexibility of the system during runtime. Its strengths are modularity, a top-down view of the system consistent with software engineering processes [38], and its independence of the self-adaptation algorithm used. It can therefore deal with arbitrary system changes at runtime and is scalable with respect to the number of agents in the system. The approach consists of the following elements:

- the Restore Invariant Approach (RIA), introduced in Sect. 3, is the theoretical framework required to model adaptive behavior and detect misbehavior at runtime;
- a verification approach based on RIA and the observer/controller architectural pattern (Sect. 4) that uses the rely/guarantee paradigm to show correct functional and reconfiguration behavior at design time as detailed in Sect. 5;
- an online result checking technique that allows the use of arbitrary self-* algorithms while maintaining functional correctness of the system, detailed in Sect. 6.

The target systems of this approach are systems based on an observer/controller architecture which implement their self-* properties by changing and adapting component configurations. In the following, we will always refer to self-* systems and imply this characteristic. This chapter focuses on the conceptional and theoretical foundations and omits the implementation details due to space restrictions, although the presented concepts were implemented for the case studies of an adaptive production scenario presented in Sect. 7 and autonomous virtual power plants, presented in Sect. 8. For more details on the implementation issues refer to [2, 30, 39].

2 State of the Art

There are several approaches for formal specification and verification of self-* systems related to the work presented here. This section presents the most important of them. Additional related work which focuses on single aspects of the overall approach will be introduced in the respective sections.

Wooldridge and Dunne state in [47] that the environment is essential for the verification of agents. They present a formal model in which the behavior of an agent and its interaction with the environment are described as a sequence of interleaved agent and environment actions. The framework used in this chapter reflects this idea and allows detailed modeling of the feedback loops in a self-* system, while still providing the ability for arbitrary system behavior. In contrast to Wooldridge and Dunne, the distinction of environment and system transitions in our approach is part of the used logical framework and thus allows the use of a comprehensive verification theory, including compositional reasoning with rely/guarantee. Further the behavior is restricted by a corridor of correct behavior formulated by constraints, which allows to specify the agent's behavior on an abstract level without having to consider the particular implementation.

In [41], Smith and Sanders present a top-down approach for incremental formal development of self-organizing systems. An abstract specification for the complete system is refined stepwise down to component level. The correctness of the system is ensured by verification of the refinement steps. The Z notation is used as specification formalism. Their approach does not distinguish self-* and functional behavior, as they do not focus on a particular architecture. Instead they look at various applications and show how refinement can be applied in each specific case. In contrast, by focusing on an observer/controller-architecture, we can derive generic properties for applications based on this architecture. Nevertheless, their work provides good strategies for the refinement between different abstraction layers which are similar to the decomposition steps presented in this chapter and can provide useful guidance for further refinement on agent level, e.g., when considering hierarchical observer/controller-architectures.

Giese et al. present a modeling and verification approach in [9, 18] for self-adaptive mechatronic systems. The interaction between the components is modeled by so called coordination patterns describing the structural adaptation process. The system states are modeled as graphs and their dynamic behavior as graph transformations. A compositional verification approach also utilizing the rely/guarantee paradigm allows verifying safety properties that can be formalized as structural invariants over the graph transformation system. The coordination patterns in their approach are similar to parts of the corridor specification in the Restore Invariant Approach, as both are specifying a correct system structure. The rely/guarantees used here specify the requirements on the components' behavior in order to exhibit the desired properties as long the system is within the corridor and the requirements on the self-* process. Their approach does not make this distinction and directly uses the specification of the component behavior, which together with the coordination patterns combines functional and self-* behavior. Their approach therefore does not allow to change the implementation of the self-* behavior without the need of performing the complete verification again.

There are a number of further approaches focused on analyzing adaptive systems. Kramer and Magee [24] use automata to specify the properties of an adaptive system and use LTSA (Labelled Transition System Analyser) for automatic analysis of execution scenarios. Their approach does not consider modular reasoning as presented here. In [49] Zhang et al. present a modular approach based on model checking for adaptive programs against global invariants and transitional properties formulated in Linear Temporal Logic (LTL). They present a model checking algorithm for the verification of adaptive programs, such as an adaptive routing protocol. In contrast to the work here, they focus solely on the verification technique and not on the specification of adaptive systems. They also do not consider the formalization of uncertainty introduced by the environment.

3 The Restore Invariant Approach

The Restore Invariant Approach (RIA) allows defining a corridor of correct behavior. The system tries to operate within the corridor as long as possible. Due to unexpected disturbances the system leaves the corridor. Disturbances can be changes in the environment, failures, new or leaving agents, or new objectives, for instance. Whenever the corridor is left the system initiates a self-* phase and tries to reconfigure in order to return to the corridor. This concept is further elaborated in Sect. 3.1. As Sect. 3.2 shows, the concepts of RIA and the behavioral corridor are the foundation for system verification. An additional element necessary to enable successful reconfiguration is a safe state the system can reach in case of a failure, as outlined in Sect. 3.3. Finally, the concepts of RIA also allow to give a clear distinction between systems that have self-* properties and those that don't as described in Sect. 3.4.

3.1 Corridors of Correct Behavior

The basic idea behind the Restore Invariant Approach is to constrain the behavior of the system so that it only exhibits correct behavior. An advantage of this approach is that the system retains its flexibility and is still able to adapt during runtime and make decisions autonomously.

From a formal point of view, a system can be described as a transition system $SYS = (S, \rightarrow, I, AP, L)$, where S is the set of states, $\rightarrow \subseteq S \times S$ a transition relation, $I \subseteq S$ a set of initial states, AP a set of atomic propositions and L a labeling function. A trace π of the system is then given by a sequence of states $s_i \in S$ whose states are related by the transition relation and which starts in an initial state s_0.

$$\pi = s_0, s_1, s_2, \ldots, s_n$$

Fig. 1 shows an example trace of an abstract transition system SYS which tries to stay within the corridor. The system recognizes a violation of the corridor and triggers a self-* process in order to reach a state within the corridor.

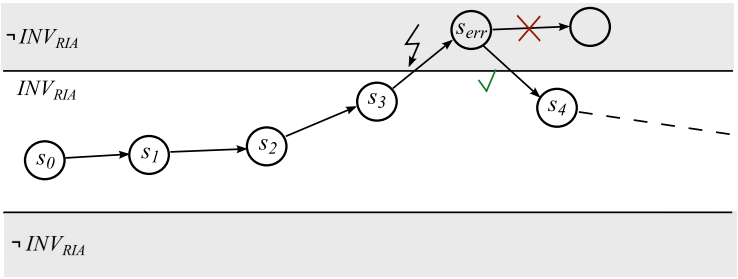


Fig. 1. Corridor of correct behavior of a self-* system [31]

Formally the specification of the corridor corresponds to a predicate logic formula[1] – the invariant INV_{RIA} – which is evaluated over a system state. The term "invariant" is used as the system's goal is to maintain the invariant on the entire system trace. The invariant differentiates the system states into those that exhibit correct behavior and those that do not. This allows to separate the states into two disjoint sets: a set S_{func} of functional states and a set S_{reconf} of reconfiguration states. The functional states are states within the corridor in which the system shows its desired behavior. The reconfiguration states are states outside the corridor in which reconfiguration is necessary. This abstract definition can accommodate a variety of situations in the system that can lead to adaptations. If, e.g., new agents that enter the system should trigger a reconfiguration, the invariant will have to be formulated so that an idle agent or one that has not been configured violates it. The system will then switch to a reconfiguration state as soon as such a situation occurs.

Related Work: In [17], Gärtner presents a similar classification of the state space for fault tolerant systems. He distinguishes three kinds of states: a set of *invariant states*, in which the system exhibits the desired properties, corresponding to the functional states of RIA; a set of states constituting the *fault span*, containing all invariant states and additionally all states which are tolerable by the system and from which the system eventually returns into an invariant state; finally, the set of all possible states.

Another classification of the state space of Organic Computing systems is proposed by Schmeck et al. in [37]. The *target space* contains the states the system should try to reach. If this is not possible, the system should at least try to get into a state of the *acceptance space*. The *survival space* consists of all states outside the acceptance space from which the system can get back into the acceptance or target space. All remaining

[1] Theoretically, a temporal logic formula could be used instead of a predicate logic formula. However, it is unclear how a system can evaluate a temporal invariant during runtime and decide whether it is violated or not. In order to decide this, the system would have to predict the future behavior. In the area of runtime verification the correctness of temporal logic properties is checked during runtime. For example, Leucker et al. try to monitor temporal logic properties during runtime [6]. In each step the property can be true, false or inconclusive. In this chapter predicate logic is used to formulate the invariant, although the general approach is not limited to it. The use of predicate logic means that the invariant can be evaluated in each state and it can be decided whether a state is within the corridor or not.

states are states within the *dead space* with no possibility to get back into the acceptance space. Compared to the corridors of RIA, Schmeck et al. split the functional states into target and acceptance space to distinguish optimal and non-optimal but correct states.

Both classifications separate the reconfiguration states into a set of states in which a path back into a functional state exists and a set where no path exists anymore. The classifications are used to describe the behavior of a self-* system on an abstract level. The specification of behavioral corridors in RIA exceeds these classifications by providing the tools to clearly define the different sets of states and to use these definitions both at design time to provide techniques for formal analysis (see Sect. 5) as well as at runtime to monitor the correct behavior of the system (see Sect. 6).

3.2 Behavioral Guarantees Based on RIA

By distinguishing functional and reconfiguration states, the requirements for the self-* properties of the system can be specified using the invariant. Whenever the invariant is violated, the system has to try to return to the corridor and to restore the invariant. The invariant is also a sufficient condition for system states that exhibit the expected behavior. That means that the system exhibits correct behavior when in a state in which the invariant holds. The correctness of the functional behavior of the system can therefore be verified independently of the self-* mechanisms. For the verification of the functional system it is assumed that there exists a mechanism that restores the invariant when it is violated. For a specific self-* mechanism it has to be proven that this assumption holds.

The definition of corridors has several more advantages compared to an explicit listing of all states. First, it is usually hard or expensive to find and list all states that are valid. It is often easier to formulate common properties that valid states need to exhibit. The abstraction induced by the invariant reduces the complexity of formal reasoning and the separate treatment of functional properties and self-* behavior can be exploited in order to give behavioral guarantees. In Sect. 5, a more detailed insight into this and an approach for providing behavioral guarantees will be given.

3.3 Safe Reconfiguration with Quiescent States

To ensure correct reconfiguration the system may not perform any actions that interfere with the reconfiguration process. Therefore the first task in a self-* phase is to transition the involved system components to a consistent and passive state in which they perform no critical actions. In literature this state is often called *quiescent state* [13, 22, 33, 48]. Kramer and Magee define a quiescent state in [22] as a state in which an agent is in a locally consistent and passive state, where it performs no actions which disrupt the reconfiguration. They also identify a quiescent state as a necessary condition for reconfiguration [23]. In a later work Vandewoude et al. [45] presented *tranquility*, a weaker condition for consistent reconfiguration. It allows an agent to still be involved in a transaction if it stops actively processing requests. As quiescence implies tranquility we use the stronger concept in the following, although it is possible to use the weaker condition in order to specify the requirements on an agent's behavior regarding a consistent reconfiguration.

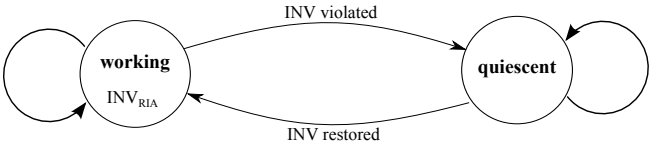


Fig. 2. Abtract view of a system's behavior

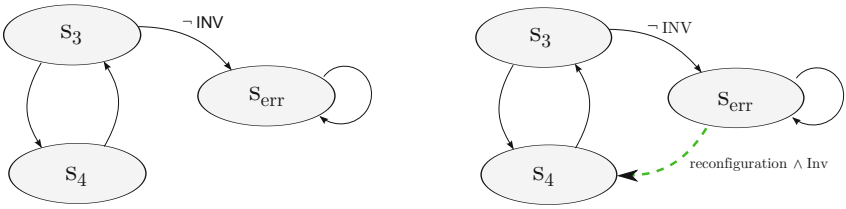
Fig. 2 shows the life-cycle of a self-* system that uses the Restore Invariant Approach. As long as the invariant holds, the system is within the corridor and can exhibit the expected behavior. When a failure occurs, the invariant is violated and the system starts a self-* phase. The first step is a transition to a quiescent state to be able to perform the actual reconfiguration process. In many cases, it is not necessary for the whole system to enter a quiescent state. Often it is sufficient that only the affected part of the system becomes quiescent, while the rest of the system can still be working. A challenge here is to identify the parts that need to be included into the reconfiguration. This question is not examined here. For details on this topic refer to [1, 40]. As soon as reconfiguration is finished and the invariant is restored, the system leaves the quiescent state and starts working again (functional phase).

What quiescence means is application-specific and has to be defined in the context of the considered application. A quiescent state can be a truly passive state in which an agent stops all actions until the reconfiguration process finishes (see Sect. 7) but also a state in which the agent continues acting according to the old configuration until the new one is calculated (see Sect. 8). The specifics of the quiescent state depend on the kind of reconfiguration used in a system and the conditions for a consistent switch to a new configuration. In Sect. 5 we will have a closer look at how these transitions are initiated with respect to a certain system architecture.

3.4 Comparing Systems with and without Self-* Properties

Based on the classification of functional and reconfiguration states, the difference between systems with and without self-* properties can be explained. Fig. 3 shows an abstract system with three states. If the system leaves the corridor and enters a state s_{err} that violates the invariant, it is therefore outside the corridor. In safety-critical applications, s_{err} is typically some kind of fail-safe state to avoid harm to human beings and the system's environment. In a fail-safe state the system still fulfills its safety properties, but usually does not guarantee any liveness properties such as progress or termination [17].

A traditional system with no self-* capabilities nor redundancy cannot reach a functional state once it has reached an error state (see Fig. 3(a)). In contrast, a self-* system (Fig. 3(b)) can return to a functional state (s_4), e.g., by reconfiguring itself. This implies that the relevant agents have to be put into a quiescent state in order to be reconfiguered consistently. During the reconfiguration, the system is changed so that redundancy within the system can be used to compensate for failures. Thus, component failures reduce the level of redundancy limiting potential future reconfigurations. The changes in the system are often enacted as part of a self-* process that changes the internal structure of the system. We distinguish two kinds of redundancy:



(a) System without self-* capabilities (b) System with self-* capabilities

Fig. 3. Behavior of a traditional system compared to a system with self-* capabilites

local redundancy, where one part of the system contains all redundancy; and distributed redundancy, where the redundancy is spread over the system. While the former can be exploited without the need of self-* properties, the latter can only be enabled by enacting a new system structure. Such distributed redundancy plus self-* properties is also able to compensate for the complete failure of a part of the system. Pure local redundancy is always subject to single-point failures and thus limits a system's recoverability. Hence the combination of redundancy and self-* properties can considerably contribute to a robust and flexible system.

4 Observer/Controller-Architectures

So far, the self-* system was considered in a very abstract and formal fashion. In this section, the observer/controller-architecture, a common architecture of self-* systems is considered. Based on this architecture, the formal modeling and verification approach with RIA is explained in detail in Sect. 5.

A way to integrate adaptivity in a system is the introduction of feedback loops [12, 14]. A change in the system or its environment triggers a reaction within the system that causes a subsequent change and so forth. Such loops can be used to model the adaptiveness of a system and to understand the dynamics that occur in a system [42].

In this work, the generic observer/controller-architecture proposed by Richter et al. in [34] is used and refined. The architecture shown in Fig. 4 is one realization of the feedback loop principle: a functional system is observed and the observations are reported to the controller which in turn effects the system in a way that it deems best to reach the system's goals. The actual effect is monitored by the observer. Thus, a feedback loop is established that guides the adaptation of the system and its behavior.

The functional system in Fig. 4 consists of several autonomous, interacting components, so called agents. These components react to control signals from the controller component, but only in case of changes in the environment which necessitate a reconfiguration of the functional system. As long as the system behaves correctly according to the corridor, the o/c-layer is passive and does not interfere with the rest of the system. However, the observer monitors the functional system which is equivalent to the monitoring of the invariant. Whenever the invariant is violated, the observer notifies the controller. The controller then initiates a reconfiguration of the system. After it has advised the agent to enter a quiescent state, it starts a reconfiguration mechanism

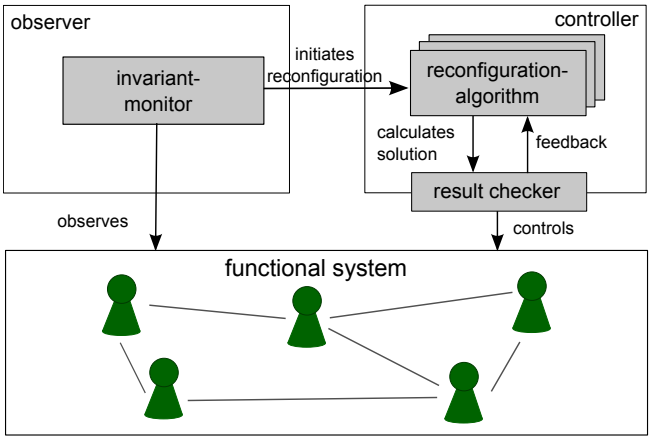


Fig. 4. An observer/controller-architecture for systems using RIA [16]. It contains a monitor component to observe the invariant as well as a result checker to verify solutions of the reconfiguration algorithm.

calculating a new configuration for the system. The new configuration has to fulfill the invariant and thus, the system will again exhibit correct behavior after the reconfiguration. The interaction between the o/c-layer and the functional system is always observer/controller-initiated.

Fig. 4 suggests that the o/c-layer is a central instance within the system. This is a sophism since the architecture can be realized in several ways [11]. Depending on the system's properties and its application area, each agent can have an individual o/c-layer, thus achieving a completely decentralized architecture. Multiple layers of observation and control can be implemented achieving a decentralized, hierarchical architecture.

Components in the system interact horizontally in each layer and vertically between the layers. Interactions can thus take place only between agents in the functional layer, between the o/c-layer and the functional layer, and between several o/c-layers, depending on the chosen architecture. Again, what all these possible variations of o/c-architectures have in common is the strict separation of functional system and o/c-layer which enables the specification and formal analysis of desired system properties as well as behavioral guarantees as shown in this chapter.

5 Formal Model of an O/C-Based System

In this section a generic formalization of self-* systems with an o/c-architecture is presented. Based on the formal framework described in Sect. 5.1 and a compositional reasoning paradigm outlined in Sect. 5.2, a formal model for systems based on o/c-architectures is developed in Sect. 5.3. After showing how such a system can be decomposed properly in Sect. 5.4, conditions for correct behavior of the individual components and interaction between the layers are formulated in Sect. 5.5. The formulated conditions can be instantiated for specific applications to retrieve the proof obligations

for the particular agents and the o/c-interaction, effectively allowing a composition verification approach.

5.1 Formal Framework

To begin with, we want to give a short overview of the formal framework used for modeling and verification. The full details, including a specialized logic and calculus along with the respective semantics, can be found in [4, 8]. For a tool-supported verification, these elements have been integrated in the interactive theorem prover KIV [5].

From a formal point of view a run of the system is a sequence of states, which is called a trace. A state is defined by an evaluation of the systems variables V. A step consists of a so called *system transition* followed by an *environment transition*. A step therefore consists of three states: an unprimed state s_i at the beginning of the step, an intermediate primed state s_i' formalizing the evaluation after the system transition, and a double primed state s_i'' for the evaluation after the environment transition. The double primed state is equal to the unprimed state for the subsequent step ($s_i'' = s_{i+1}$). Thus the system and environment transition alternate, as depicted in Fig. 5. This trace based view of the system is necessary as the properties one expects from the system and the guarantees about its behavior are temporal properties. Desired guarantees are, e.g., that the system *never* shows some unwanted behavior or a property *always* holds.

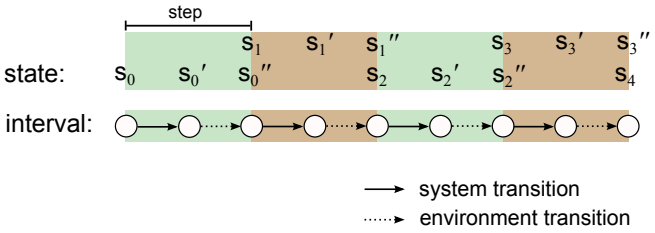


Fig. 5. Relation between unprimed, primed and double primed states. A step consists of a system transition followed by an environment transition.

Besides the variables v there are also primed and double primed variables in order to accurately formalize the transitions. For each variable $v \in V$ there is a corresponding primed variable v' and double primed variable v''. The sets of all primed/double-primed variables is denoted accordingly by V' and V''.

A *system transition* ($s_i \mapsto s_i'$) can therefore be formulated as a predicate logic formula over V and V' that describes the relation between the values of the variables $v \in V$ before and after the system transition. An *environment transition* ($s_i' \mapsto s_i''$) analogously describes the changes during an environment transition. As the double primed state is the unprimed state of the successive state, the value of v'' in state s_i'' is equal to the value of v in the next successive state s_{i+1}. For example, $x' = x + 1$ expresses that the system increases x by one and $x'' = x'$ expresses that during an environment transition x is not changed. If there is no statement made for a variable in the environment step, this means that the value can be changed arbitrarily by the environment. This is a crucial feature

of the framework: if no restriction is put in place, no assumption is made of what the environment is capable of. It is therefore not necessary to explicitly model everything the environment can do, but verify the system for completely arbitrary changes in the environment by consciously under-specifying certain aspects.

While it is a great advantage to be able to leave many aspects of the environment open, an explicit model of *some* of the behavior of the environment can be very beneficial in the case of self-* systems [47]. If regarded properly, the system boundary can be clearly established and thus a clear separation between the system and its environment can be achieved. This also aids in the modeling of the interactions between the system and its environment, as exemplified in Sect. 7 and Sect. 8.

Parallel components are expressed through an interleaving operator $\|$. The interleaving of two components (agents) Ag_i and Ag_j means that either Ag_i or Ag_j can make a system transition. The particular agent cannot distinguish how many transitions the other agents have done between two of its steps. From its local point of view everything occurred in a single environment transition. That means from an agent's point of view the system transitions of the other agents are in its environment transitions as well as changes made by the global environment. This is illustrated in Fig. 6.

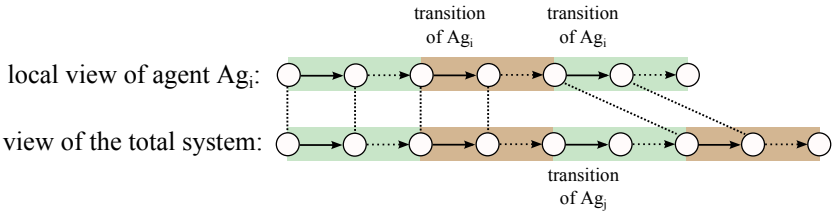


Fig. 6. Local view of an agent Ag_i and its relation to the run of the total system

5.2 Compositional Reasoning with Rely/Guarantee

So far we established a global view of the system. However, the goal is to retrieve properties of single agents and to have a local view on the system but still be able to guarantee properties of the complete system. The observer/controller-architecture provides a natural way for the decomposition into several subcomponents. This is depicted in Fig. 7. The complete system can at first be split into the observer/controller o/c and

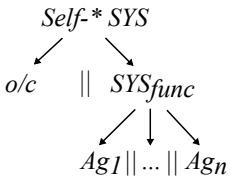


Fig. 7. Compositional view of the system structure

the functional system SYS_{func}. Both are running in parallel which is represented by the interleaving operator $||$. The functional system can again be split into several agents $(Ag_1,\ldots,Ag_n)$ that are running in parallel as well. Of course, the observer/controller can also consist of several parallel components. This is not considered here, as for the verification of functional properties only the *specification* of the complete o/c-layer is required. However, the approach presented for the functional system works for a decomposition of the o/c as well.

This modular structure can be used for a compositional verification approach. The idea behind compositional verification is to reason about properties of the global system by proving properties of single components only. The main advantage is that reasoning over single components is usually less complex then reasoning over a parallel system. A common compositional proof technique is the *rely/guarantee paradigm* which is used here and was introduced by Jones in [21] and by Misra and Chandy under the term assumption-commitment in [26].

The basic idea is that each component guarantees a specific behavior as long as it can rely on some properties of its environment. The behavior of a component is specified by a *guarantee* $G(V,V')$ provided by the component. This is expressed as a predicate over the component's transitions. To be able to guarantee the specified behavior, the component needs to be able to make assumptions about its environment, as it *relies* on certain – but not necessarily completely specified – aspects of behavior of its environment. If no relies are formulated at all and the environment is thus completely arbitrary, a component will not be able to give any guarantees, as every system action can immediately be revoked by the environment[2]. To create relies that limit the behavior of the environment as little as possible, they are usually defined by excluding some particular behavior. A typical property of the environment is that it does not change a component's internal variables. Formally, a rely $R(V',V'')$ is specified over the environment transitions.

The behavior of a component Ag_i can then be specified using both rely and guarantee. As long as the rely $R_i(V',V'')$ holds, the component guarantees $G_i(V,V')$. This property is formalized as $R_i(V',V'') \stackrel{+}{\rightarrow} G_i(V,V')$. The rely/guarantee specification abstracts from the internal implementation of the component and specifies the external behavior a component should exhibit. It is therefore a black box specification.

In order to be able to reason about the global system, a compositionality theorem developed by Bäumler et al. [7] is used. It describes the necessary correlations between the local rely/guarantees R_i/G_i of the components C_i and defines the proof obligations in order to guarantee a global rely/guarantee property R/G of the combined system. The main obligation is to prove that each component behaves according to its local rely/guarantee specification. The other obligations ensure the compatibility and consistency among the rely/guarantees, e.g., the guarantee of one component does not violate the rely of another component.

Theorem 1 (Compositionality theorem). *If:*

i. $\forall\, i = 1,\ldots,n : C_i, Init(V) \vdash R_i(V',V'') \stackrel{+}{\rightarrow} G_i(V,V')$
ii. $\forall\, i = 1,\ldots,n : G_i(V',V'') \rightarrow G(V',V'') \wedge \bigwedge_{j=1,\ldots,n \wedge j \neq i} R_j(V',V'')$
iii. $\forall\, i = 1,\ldots,n : G_i(V,V)$

[2] Note, that from the local view of a component, the environment contains all other components.

iv. $\textit{for all } i = 1,\ldots,n : R_i(V,V') \wedge R_i(V',V'') \rightarrow R_i(V,V'')$
v. $R(V',V'') \rightarrow \quad _{i=1,\ldots,n} R_i(V',V'')$
vi. $\exists\, V \,:\, \textit{Init}(V)$

then: $C_1 || \ldots || C_n, \textit{Init}(V) \vdash R(V',V'') \xrightarrow{+} G(V,V'))$

The informal meaning of the proof obligations of this theorem are as follows:

i. All components must sustain their guarantee as long as the rely holds. It can be assumed that an initial condition $\textit{Init}(V)$ holds in the first step.
ii. The guarantee of each component preserves the global guarantee and does not violate the relies of all other components.
iii. The local guarantee is reflexive, that means it must hold if nothing (no variable) is changed.
iv. The relies of all components are transitive. With this property, a component's relies are preserved even if other components make several steps in a row.
v. All component relies hold if the global rely holds. Therefore, no component rely is violated in the environment step. This implies that an agent cannot assume that no failures occur, for instance.
vi. An initial configuration for the system must exist. This ensures that obligation *i* is consistent.

If the rely/guarantees fulfill these properties and the component implementation is correct with respect to its local rely/guarantee property then the global guarantee holds for the complete interleaved system.

If the system consists of several identical components of the same type only one of these components has to be proven. The theorem then allows to reason about a system consisting of an arbitrary (but finite) number of components running in parallel, also including changing numbers of agents. The proofs are also valid when the number of agents changes during runtime. Thus, the kinds of adaptivity that can be covered with this technique include component failures, as well as adding and removing agents. The adaptivity of the components is realized by changing their parameters.

The compositionality theorem was proven with the interactive theorem prover KIV and can therefore be directly applied during a proof for a particular system. The advantage is that the reasoning is tool-supported and that for a particular system only the local R/G properties have to verified against the components implementation. More information about the theorem, the resulting proof obligations and technical details can be found in [36].

5.3 Formal Model of a System Based on an Observer/Controller-Architecture

With these formal foundations in place, it is now possible to define an abstract formal model for systems based on an observer/controller-architecture that can be used to formally specify the corridor of correct behavior and the requirements for a correct o/c implementation. The model can then be instantiated for a concrete system to provide behavioral guarantees by formal verification of the functional system part and an observer/controller implementation against their respective specifications.

The system can be described as a set of variables V_{all} which is split into a set V_{func} of variables defining the state of the functional system and a set V_{env} describing the system's environment. The variables in V_{func} again can be split into the following disjoint sets, as depicted in Fig. 8:

- A set V_{conf} of variables which contain the configuration of the functional system. These are the variables (parameters) that can be changed by the observer/controller during a self-* phase in order to restore the invariant.
- A set V_{int} of variables that contains the agent's internal variables. They can only be changed by the agents. This set contains variables which the agents use for internal calculations and to store intermediate data.
- All other variables are in V_{rest}. These variables can be changed by the environment and describe, e.g., sensor data or hardware status which can both include errors or be changed at random points in time.

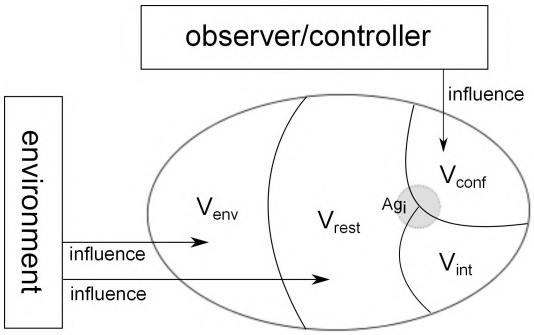


Fig. 8. Abstract o/c-system with different variable sets for environment and functional system

The set of the variables of the functional system therefore is defined as $V_{func} := V_{conf} \cup V_{int} \cup V_{rest}$. The set of V_{env} models the environment of the system and allows to express the effect an agent's action has on its environment. As the agents' actions alter the environment, and the environment and the agents are interleaved, feedback loops can thus be formalized.

Each agent has its own set of variables V_{func}. Additionally the agents have two variables (flags) *reconf* and *deficient* which model the o/c-interaction. The flag *reconf* signals an agent that a self-* phase has started and that it should behave passively. The flag *deficient* signals an agent that a reconfiguration occurred. The second flag is necessary as it is theoretically possible that between two steps of an agent several steps of the remaining system – including a complete reconfiguration – occurred due to the asynchronous execution. In a concrete implementation these flags not necessarily have to be flags that are set by the o/c directly, they can also be refined to a message passing communication model. Such a model allows interactions between the layers to be conveyed by messages and complex protocols, e.g., handshake protocols with multiple messages.

Formally the state of an agent is represented as a tuple:

$$\begin{aligned} \text{state}_{ag} := (\ &.\text{id} : \text{ID} \times .\text{reconf}: \text{bool} \times .\text{deficient} : \text{bool} \\ &\times .\text{vconf} : V_{conf} \times .\text{vint} : V_{int} \times .\text{vrest} : V_{rest}); \end{aligned}$$

The dynamics are modeled as state transitions specifying how the variables of an agent change during a system step. Formally, this is expressed as transition predicates relating unprimed and primed variables.

5.4 Decomposition of the Observer/Controller-Architecture

As described above and illustrated in Fig. 7, the self-* system can be decomposed into several components running in parallel. If we apply the rely/guarantee approach to the observer/controller-architecture the system can be decomposed in two steps (see Fig. 9).

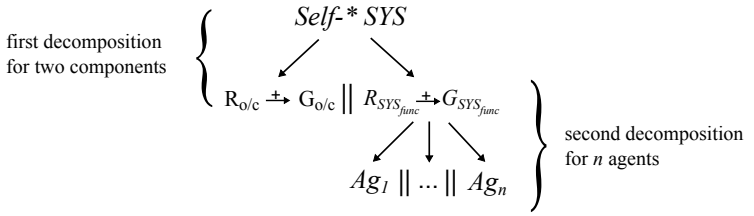


Fig. 9. System structure and compositionality

At first the system is decomposed into the o/c part and the functional system. The behavior of both is specified with corresponding rely/guarantee properties. On this level we have a parallel system consisting of two components. For a particular implementation, the following has to be proven:

- For the o/c part it must be proven that the implementation of the observer/controller (o/c_{impl}) is correct with respect to its specification.

$$o/c_{impl} \models R_{o/c} \xrightarrow{+} G_{o/c} \tag{1}$$

- For the functional system it must be verified that the agents behave according to their rely/guarantee property.

$$SYS_{func} \models R_{SYS_{func}} \xrightarrow{+} G_{SYS_{func}} \tag{2}$$

The first decomposition leads to a separation of concerns. Both system parts can be treated separately. The rely/guarantee properties specify the interaction between both layers. This specification and the properties each layer must guarantee ensure that the complete system exhibits the expected properties. The correctness of the global system is ensured by a compositionality theorem.

The functional system itself consists again of a number of agents and can be decomposed in a second step into several agents. The course of action is the same as for the first decomposition step. The rely/guarantee property of the functional system

(Eq. 2) is broken down to local rely/guarantee properties for the individual agents and are enriched with properties about the interaction between the agents themselves. For a particular implementation Ag_i of an agent it must be proven that it is correct according to its R/G specification:

$$Ag_i \models R_i \xrightarrow{+} G_i \tag{3}$$

The R/G specification contains the interaction of the agent with its observer/controller and other agents with shared variables. It also contains, e.g., the individual contribution of one agent to the global task.

In the next sub-section we define generic rely/guarantee properties for the first decomposition step. They specify the requirements on the interaction between the observer/controller and the functional system in order to prove that a property *Prop* always holds.

5.5 Rely-Guarantee Definition of the Interaction between Functional and Self-*-Layers

Before we consider the verification of Eq. 1 and Eq. 2 against a particular implementation, we need to specify the rely/guarantee properties first. The observer/controller-architecture reflects the distinction of the two phases of the restore invariant approach. The functional part is mainly responsible for establishing the functional properties of the system and therefore is active during the functional phases. The observer/controller-layer is responsible for monitoring the invariant and reconfiguration in case of an invariant violation. It puts the functional part into a quiescent state and is mainly active during the self-* phases.

Observer/Controller Specification. First, we look at the observer/controller specification and its relies and guarantees. A correct o/c implementation has to guarantee that at the end of every self-* phase the invariant is restored. That means whenever the o/c signals an agent to leave its quiescent state, INV_{RIA} must hold.

Note that this does not require that the o/c always finds a solution. This would imply a perfect o/c, which is not realistic as sometimes there is no solution possible, e.g., when no more redundancy is available in the system. Eq. 4 is sufficient to guarantee safety properties. Additional requirements can be added to express some quality criteria concerning the reconfiguration. Further, the observer/controller has to guarantee that it does not interfere with the functional system in functional phases and that it always puts the agent into a quiescent state before changing the configuration parameters ($noInterference(V_{all}, V'_{all})$). It also guarantees not to change the agent's internal variables ($Unchg_{sys}(V_{func} \setminus V_{conf})$) and not to violate the system properties that should be proven (denoted here by *Prop*). These system properties are usually defined by the developer and are retrieved from the requirements on a particular system.

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, i \,:\, (Ag_i.reconf \wedge \neg\, Ag'_i.reconf \rightarrow INV_{RIA}(V'_{all}))) \\ & \wedge\, (\forall\, i \,:\, \neg\, Ag'_i.reconf \rightarrow noInterference(V_{all}, V'_{all})) \\ & \wedge\, (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ & \wedge\, Unchg_{sys}(V_{func} \setminus V_{conf}) \end{aligned} \tag{4}$$

To be able to guarantee this behavior the observer/controller relies on the agents not to leave their quiescent state themselves. In terms of the reconf variable, this means that it is only changed on the o/c's initiative.

$$R_{o/c}(V'_{all}, V''_{all}) :\leftrightarrow \quad (\forall\, i\, :\, Ag'_i.reconf \leftrightarrow Ag''_i.reconf) \\ \wedge\, (Prop(V'_{all}) \rightarrow Prop(V''_{all}))$$

The observer/controller also assumes that the property is maintained by the rest of the system (all the agents currently participating in the system). This is necessary as we want to prove that the property is never violated by the complete system and the functional system is part of the system as well.

Functional System Specification. The functional system guarantees that it does not change the configuration on its own. It also has to guarantee the considered property *Prop*. Further, the functional system must guarantee to be quiescent during the self-* phase and only to leave the quiescent state on o/c notifications.

$$G_{SYS_{func}}(V_{all}, V'_{all}) :\leftrightarrow\ V_{conf} = V'_{conf} \\ \wedge\, (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ \wedge\, (\forall\, i\, :\, Ag_i.reconf \rightarrow quiescence(V_{all}, V'_{all})) \\ \wedge\, (\forall\, i\, :\, Ag_i.reconf \rightarrow Ag'_i.reconf)$$

To ensure this, it relies on a correct o/c behavior, i.e., the o/c only changes the configuration variables in self-* phases and the internal variables of the agents are only changed by themselves. It further relies on the remaining part of the system not to violate the expected property as well.

$$R_{SYS_{func}}(V'_{all}, V''_{all}) :\leftrightarrow \quad (\forall\, i\, :\, \neg\, Ag''_i.reconf \wedge \neg\, Ag''_i.deficient \rightarrow Unchg_{env}(V_{conf})) \\ \wedge\, (\forall\, i\, :\, Ag'_i.deficient \rightarrow Ag''_i.deficient) \\ \wedge\, (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ \wedge\, Unchg_{env}(V_{int})$$

Applying the compositionality theorem from Sect. 5.2 it can be proven that the abstract property *Prop* also holds for the combined system if both parts behave according to their rely/guarantee specification. The environment of the complete system is still allowed to arbitrarily change the environment (V_{env}) and the agents' variables V_{rest} which are, e.g., specifying the agents' sensor data or hardware status. It is just assumed that the environment cannot change an agent's internal variables or configuration parameters.

5.6 The Verification Process

We now have all elements in place that are required to actually verify a concrete application. This process is exemplified with two case studies in Sect. 7 and Sect. 8. In all cases, four general steps have to be followed:

1. Define formal model, system dynamics, and property:
   - define a formal model of the functional system, respectively the agents;
   - define the variables describing the system state;
   - specify the dynamic behavior, i.e. the state changes;
   - specify the property the functional system has to adhere to.
2. Define the corridor and reconfiguration behavior:
   - specify the application specific corridor based on the formal model;
   - define what the quiescent state of an agent is;
   - specify what *noInterference* means for the observer/controller in the particular application.
3. Instantiate the abstract rely/guarantees in this section with the application specific variables and formulas:
   - for the observer/controller;
   - for the functional system.
4. Verify that the observer/controller and the functional system behave according to the instantiated rely/guarantee properties.

Following this course of action will verify that the functional system behaves according to its specification and that the interaction between observer/controller and the functional system is correct. This includes all behavior that takes place when the system is reconfigured. However, the actual reconfiguration algorithm implemented in the controller part of the o/c is not subject to verification. Therefore, an additional measure has to be put in place. This measure is discussed in the following section before the verification process is exemplified with two case studies.

6 Observer/Controller Correctness by Verified Result Checking

The verification of the functional system relies on a correct observer/controller-layer. For the verification of the functional system it was assumed that the observer/controller applies configurations that fulfill the invariant. To verify this property, one option is to reason about the reconfiguration algorithm in question by direct verification. Depending on the algorithms' complexity this task can be arbitrarily difficult or even infeasible as often bio-inspired algorithms, learning techniques or stochastic approaches are used to implement the self-* features. These algorithms are not necessarily sound nor complete and thus do not always return valid results which disqualifies them for direct verification. In this section we want to give a brief insight into a technique that allows to formally verify the correctness independent of the particular algorithm with a verified result checker. Sect. 6.1 outlines the concepts we developed while Sect. 6.2 shows how the result checker can be derived from the specification and be used in the system. For more details, refer to [16].

6.1 Foundations of Verified Result Checking

As a technique to avoid direct verification and as an alternative to pure online verification [25] of the complete system we developed the concept of *verified result checking*,

which combines the classical idea of result checking by Blum, Wasserman and Kannan [10, 46] and formal program verification.

Result checking is a way to ensure the correctness of a program by another program. In contrast to testing and verification the correctness is not enforced by ensuring the correctness of the used algorithm itself, but by checking the correctness of all of its results at runtime. To actually be able to give guarantees in advance, we combine the result checking approach with design time verification of the particular result checker. This allows for runtime assurance but design time verification of the algorithm. It also makes it possible to switch algorithms during runtime, e.g., to have specialized algorithms for different situations. The verification task is reduced to the verification of the result checker. The program to check whether a configuration is correct is usually simpler than the program to calculate a configuration. This makes verification of the result checker less complex than the verification of the reconfiguration algorithm.

A result checker is therefore a short program *RC* that reads the output of the program to check and returns `correct` if the result is correct and `incorrect` otherwise (see specification below). It is executed after the reconfiguration algorithm and reads the calculated configuration (V_{conf}). If the configuration restores the invariant, the checker returns `correct` and forwards the configuration. If the configuration is incorrect, it is blocked and feedback is provided to the reconfiguration algorithm. This feedback can consist of the parts of the invariant that are violated or – if a metric is available – a measure of the error of the solution.

Specification of **Result Checker** (*RC*)	
input	configuration (V_{conf}) of o/c_{impl}
output	'correct' if $Inv_{RIA}(V_{conf})$, 'incorrect', otherwise

While this technique allows to check new configurations for validity, the lack of a direct verification of the algorithm means that no statement can be made about termination of the algorithm. Therefore only correctness and quality properties of a configuration that is forwarded to the system but no liveness criteria in the sense of "something good will eventually happen" about the self-* phase itself can be proven with the result checking approach. Liveness properties ensure that the system makes progress in some manner, while safety properties ("something bad will never happen") are properties that ensure that there are no threats to life and limb. However, liveness properties in an self-* system are hard to verify as failures can always occur and additional assumptions about the environment have to be made, e.g., assumptions about the frequency of failures. Failures eat redundancy, in the sense of possibilities that another component can take over missing functionality. If no redundancy is available, the system can not compensate a failure any more and therefore, no liveness properties can be guaranteed. They can thus be treated as a kind of quality properties of a considered implementation. This might not be a very relevant limitation in practice but will have to be considered in formal verification. However, the system is still able to guarantee safety properties which are most interesting for self-* systems anyway.

Related Work: A similar idea in order to enable the use of unsound algorithms and still ensure correct results, but not with the focus on formal verification and correctness guarantees, is presented by Rochner and Müller-Schloer in [35]. They add a so called *guard* to their Observer/Controller-architecture that filters the actions calculated by the controller. In principle, this corresponds to the result checking idea. However, they do not consider the correctness of the *guard* itself nor do they describe how such a filter can be derived systematically.

A related field is *runtime verification* [25] which deals with checking the correctness of a property during runtime. This approach tries to completely move the verification from design to runtime, by developing suitable monitors in order to accurately decide whether a property holds or not. In contrast to the approach presented here *runtime verification* does not deal with the question how a system can be adapted if a violation is detected. However, ideas and results from the field can be used to develop appropriate observers in order to detect system failures and invariant violations.

6.2 Deriving und Using a Result Checker

The implementation and the formal specification of a result checker can be systematically derived from the specification of the invariant. The result checker implementation is then verified to prove that it returns `correct` for a configuration if and only if the invariant $INV_{RIA}(conf)$ evaluates to true. In [16] a systematic development and verification of a result checker for self-organizing resource-flow systems is described in detail.

The invariant can be seen as constraints on the configuration variables, as the corridor is formulated as a predicate $INV_{RIA}(V)$ over all variables V. It usually constrains the configuration variables in relation to the remaining variables and therefore describes correct configurations with respect to the system's situation. If the invariant is violated, e.g., due to a failure which is reflected in a change of a variable's value, the observer/controller tries to find a new evaluation for the configuration variables, which re-establishes the invariant.

The reconfiguration task can therefore be considered a constraint satisfaction problem (CSP) [15, 44]:

$$CSP_{reconf} := (V_{conf}, D_{conf}, INV_{RIA}(V))$$

The decision variables are the configuration variables V_{conf} with corresponding domains D_{conf} and the constraints are defined through the invariant. The goal is to find a valid evaluation of the configuration variables such that the constraints (invariant) are fulfilled [29]. Usually one is not only interested in whether a solution is correct or incorrect, but also in how good a solution is. In case of an incorrect solution detailed feedback is required in order to find a new and valid configuration. Therefore the result checker can be extended with a penalty function which quantifies the quality of a configuration.

In case of an invalid result the result checker provides feedback about which constraints are violated for which agents. This detailed feedback can than be used by the self-* algorithm to find a better solution. For instance, if a genetic algorithm is used, the result checker can be called by the fitness function and used as one element in the calculation of the fitness values of the configurations in a population.

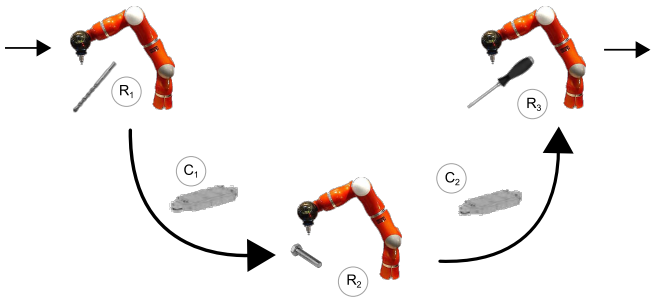


Fig. 10. An adaptive production cell with three robots, two transport units, and three tools [19]

7 Application to an Adaptive Production Cell

In this section, the application of the specification and verification approach to a simple adaptive production cell is described. The cell depicted in Fig. 10 consists of three robots and two autonomous transport units (carts) connecting them. Every robot can accomplish three tasks: drill a hole into a workpiece (D); insert a screw into this hole (I); and tighten the screw with a screwdriver (T). For each task the robots have different tools which they can switch.

Every workpiece entering the cell has to be processed according to a given order, e.g., drill, insert, tighten. In case one or more tools break and the current configuration allows no more correct processing of the incoming workpieces, the observer/controller is reconfiguring the cell and re-assigning the different tools such that production can continue. Further the carts have to be re-routed in order to preserve the right processing sequence. They always have to transport from the drilling robot to the inserting robot and from the inserting robot to the robot that is tightening the screw. As the system is deciding on its own which robot is applying which tool, we at least want to have the guarantee that workpieces are processed correctly: the tools are applied in the right order and workpieces leaving the cell are fully processed with all three tools.

1. Define Formal Model, System Dynamics, and Property. The first step is to build a formal model of the production cell. Each robot has, besides the two flags *reconf* and *deficient* for reconfiguration, a variable *availableTools* describing the set of available tools and a variable *assignedTool* for the currently assigned tool. Access to a variable of a Robot *r* is denoted by *r.assignedTool*. The set *robots* contains all currently participating robots. The task is specified as a sequence of tools which should be applied to a workpiece. A workpiece *wp* therefore has two variables: *wp.state* modeling its current processing state and *wp.task* for the task for this workpiece. The set of workpieces currently in the production cell is denoted with *cell*. In order to formulate properties about workpieces leaving the cell, the leaving workpieces are stored in a *storage* (list of workpieces). The expected property (*Prop*) the system should exhibit then is, that

all workpieces have correct state and leaving workpieces are fully processed. It can be formulated as follows, where $\sqsubseteq$ is the standard prefix operator for lists:

$$\begin{aligned} correctProcessing(V_{all}) :\leftrightarrow \quad & (\forall\, wp \in cell : wp.state \sqsubseteq wp.task) \\ & \wedge\, (\forall\, wp \in storage : wp.state = wp.task) \end{aligned}$$

This property should always be maintained by the complete system and its validity on the entire trace is the behavioral guarantee we want to give for the system.

In [39] an extended variant of the adaptive production cell is described. It includes correct routing of the carts which was not considered here. This leads to additional properties in INV_{RIA}, describing correct routes. Carts are also specified with rely/guarantees. The specification of the corridor and its verification is presented in [32]. Further, the paper contains a role concept for the agents and a detailed model of the dynamics of the agents modeled with UML-statecharts. The implementation of the production scenario is described in [30].

2. Define the Corridor and Reconfiguration Behavior. The invariant specifies correct configurations of the production cell which leads to correct behavior. A valid configuration is one that assigns all needed tools. In other words, for each workpiece in the cell, all tools of the workpiece's task have to be assigned. Further, only tools available to a robot can be assigned.

$$\begin{aligned} INV_{RIA} := \quad & (\forall\, wp \in cell\, \forall\, t \in wp.task : \exists\, r \in robots : r.assignedTool = t) \\ & \wedge\, (\forall\, r \in robots : r.assignedTool \in r.availableTools) \end{aligned}$$

In this application quiescence means that a robot stops during reconfiguration and does not perform any processing steps, like applying a tool. The observer/controller furthermore guarantees that it does only interfere when a self-* phase was started in beforehand. This means that the o/c does only change a robot's *assignedTool* in functional phases.

3. Instantiate the Abstract Rely/Guarantees. In order to retrieve the rely/guarantee properties the variables have to be assigned to the generic variable sets.

$$\begin{aligned} V_{int} &:= \{r.ID \mid \forall\, r \in robots\} \\ V_{conf} &:= \{r.assignedTool \mid \forall\, r \in robots\} \\ V_{rest} &:= \{r.availableTools \mid \forall\, r \in robots\} \\ V_{env} &:= \{wp.state, wp.task \mid \forall\, wp \in cell \cup storage\} \end{aligned}$$

For this scenario, the robots' internal variables are merely their IDs. The set of configuration variables contains the *assignedTool* variable of each robot. The *availableTools* of each robot can be arbitrarily changed by the environment and are therefore in V_{rest}. The environment is thus allowed to change this set which models tool failures but also maintenance in case the set is extended. From the point of view of the production cell, the task according to which a workpiece should be processed, the storage and the state of the workpieces are in V_{env}. For example, the application of a tool changes the environment as the workpiece's state is manipulated. The task of new workpieces entering the cell is set by the environment.

Observer/controller specification. The next step is to instantiate the generic R/G property of the observer/controller. The observer/controller has to guarantee that it correctly restores the invariant at the end of a self-* phase, indicated by changing the robots *reconf* variable, and that it does not interfere in functional phases. It also guarantees that it does not violate the *correctProcessing* property and that it only changes the configuration variables. The instantiated guarantee then looks like this:

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, r \in robots \;:\; r.reconf \wedge \neg\, r'.reconf \rightarrow INV_{RIA}(V'_{all})) \\ & \wedge\, (\forall\, r \in robots \;:\; \neg\, r'.reconf \rightarrow r'.assignedTool = r.assignedTool) \\ & \wedge\, (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\ & \wedge\, Unchg_{sys}(V_{func} \setminus V_{conf}) \end{aligned}$$

The rely $R_{o/c}$ for the o/c-layer is the same as the generic one which assumes that the functional system does not end the self-* phase on its own. Every implementation, distributed or central, that fulfills this R/G-property is a valid o/c-implementation. For the production cell scenario a central and a distributed o/c-layer was implemented [1, 29]. The central one uses a constraint solver in order to calculate new valid assignments. The distributed one is based on coalition formation and tries to find a minimal set of robots that is able to reconfigure the cell. The correctness in both cases is ensured via a verified result checker (see Sect. 6) which ensures that only correct configurations are forwarded to the functional system.

Functional system specification. The rely/guarantee property for the functional system is retrieved analogously by instantiating the generic property:

$$\begin{aligned} G_{SYS_{func}}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, r \in robots \;:\; r.assignedTool = r'.assignedTool) && (1) \\ & \wedge\, (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) && (2) \\ & \wedge\, (\forall\, r \in robots \;:\; r.reconf \rightarrow V'_{all} = V_{all}) && (3) \\ & \wedge\, (\forall\, r \in robots \;:\; r.reconf \rightarrow r'.reconf) && (4) \end{aligned}$$

The functional part does not change the configuration on its own (1) and guarantees the expected property *correctProcessing* (2). It also guarantees to enter the quiescent state, when a self-* phase was initiated (3). The functional system ensures that it only leaves the quiescent state when notified by the o/c (4). To be able to guarantee this, the functional system relies on the o/c not to change the configuration in functional phases. Further it relies on others to not change its internal variables. In this case, the rely is identical to the generic $R_{SYS_{func}}$ shown on page 96.

As the functional system consists of several robots the next step is to split the R/G property into properties for the single robot in a second decomposition step. In this decomposition the interaction between the robots must also be considered, as from the point of view of each individual robot, the o/c as well as the other robots are in its environment. The local rely/guarantees are retrieved by restricting the property to the scope of a single robot. Additionally, each robot has to guarantee that it does not change the variables of the other robots. The local R/G property for a single robot r then is:

$$
\begin{aligned}
G_r(V_{all}, V'_{all}) :\leftrightarrow \; & (r.assignedTool = r'.assignedTool) \\
& \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\
& \wedge (r.reconf \rightarrow V'_{all} = V_{all}) \\
& \wedge (r.reconf \rightarrow r'.reconf)
\end{aligned}
$$

The rely is restricted to the local scope analogously.

$$
\begin{aligned}
R_r(V'_{all}, V''_{all}) :\leftrightarrow \; & (\neg\, r''.reconf \wedge \neg\, r''.deficient \rightarrow Unchg_{env}(V_{conf})) \\
& \wedge (r'.deficient \rightarrow r''.deficient) \\
& \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\
& \wedge Unchg_{env}(r.ID)
\end{aligned}
$$

4. Verification. For the particular implementation it must be proven that it is correct with respect to this R/G-specification. Hence, the number of proofs depends on the number of different agent implementations. In case of a homogeneous system consisting of identical agents – like the robots – only one proof has to be made, while in heterogeneous systems in which different agent types can have different dynamics, the proofs have to be performed for each agent type separately.

In this application the functional and self-* phases are alternating. Hence, in the quiescent state the robots come to a full stop while being reconfigured. On the other end of the spectrum of quiescent behavior are systems in which the self-* layer works in parallel with the functional system permanently. In such a case, the o/c-layer is constantly applying new configurations and the requirements for the quiescent state must be less restrictive. Such a system is presented in the next section.

8 Application to Autonomous Virtual Power Plants

Future energy systems require autonomous, decentralized management to deal with the enormous number of power generators and controllable consumers. Decentralized power generation and the limitations of the power network make it necessary to locally manage the balance between power production and consumption in a decentralized fashion. Autonomous Virtual Power Plant s(AVPP) [1, 2] could be the building blocks of such a future system. One AVPP controls a number of small energy producers such as biogas plants, solar plants and run-of-the-river power plants. The plants are divided into stochastic ones such as solar and wind generators and controllable ones. The AVPP's task is to control the plants in such a way that the load equals the combined production of the plants by calculating schedules for the controllable plants. The control decisions are based on forecasts of the plants' power production and of the load. A correct schedule for the individual plants can be calculated by a genetic algorithm or a particle swarm optimizer. The schedule changes as new prognoses come in and old ones are revised. The AVPP thus self-adapts constantly to new information and to the new environmental situation, meaning that there are no strictly discernible self-* and functional phases. In addition, a reactive algorithm compensates for slight errors

in the predictions by adapting each power plant's output slightly if current data about production and load become available [3]. However, the calculation of new power plant schedules can be seen as the main self-adaptive feature and is thus designated as the self-* phase in the following.

1. Define Formal Model, System Dynamics, and Property. An AVPP constitutes an observer/controller that manages a functional system consisting of a set N of individual power generators. If a power plant does not produce the power it forecast or the load changes unexpectedly, the invariant of the AVPP is violated and a new schedule has to be calculated. The total load which should be met is denoted as L_c and is available to the observer/controller and all power plants. The AVPP calculates schedules for the power plants based on a load prognosis L_{prog} which approximates the future load. Each power plant $i \in N$ has a scheduled target output $P_{target,i}$ that is derived from the AVPP's target output which in turn is determined by the prognosed load ($P_{target} = L_{prog} = \sum_i^N P_{target,i}$). As the schedule is made for several timesteps in advance, P_{target}, L_{prog} and $P_{target,i}$ are lists of values. The scheduled target output for time t is denoted by $P^t_{target,i}$.

The property (*Prop*) that is of importance in the energy system is grid stability. The power grid is sensitive to imbalances between consumption and production. If they differ, the network frequency changes which can lead to power outages and destroy equipment. Therefore, the AVPP has to guarantee that – if the forecast of the upcoming load is good enough – it will always produce as much power as requested. This boils down to an approximate equality between the scheduled target output of the power plants for the current timestep and their actual output in this time step:

$$gridStability(V_{all}) := P^{now}_{target} \approx \sum_i^N P_{actual,i}$$

Again, it is not sensible to demand strict equality since the prognosis can never be guaranteed to be exactly equal to the actual load. As there is a band in which the power grid can operate and the reactive mechanism can compensate slight deviations, this is not strictly necessary.

2. Define the Corridor and Reconfiguration Behavior. The next step is to formalize the corridor of correct behavior. A valid configuration is one that describes a valid schedule for the system and that ensures that in sum as much power is produced as currently is consumed and that the schedule will be able to cover the consumption predicted for each timestep t.

The first constraint describes that a plant's assigned target output has to be either zero or between the power plant's minimal and maximal output possibilities.

$$C_{cons} : \forall\, i,t : P^t_{target,i} \neq 0 \rightarrow P_{min,i} \leq P^t_{target,i} \leq P_{max,i}$$

Further a valid schedule has to assure that the change of output power from one time step to the next is not greater than the rate of change of a plant (v_i).

$$C_{change} : \forall\, i,t : |P^{t+1}_{target,i} - P^t_{target,i}| \leq v_i$$

In every timestep each plant's target output should be approximately equal to the current output ($t = now$).[3]

$$C_{balanced} := \forall i : P^{now}_{target,i} \approx P_{actual,i}$$

The current power output varies due to the reactive behavior of the power plants. A further constraint describes that the difference between the next target output and the current output may not exceed the rate of change of the plant.

$$C_{variance} := \forall i : |P^{now+1}_{target,i} - P_{actual,i}| \leq v_i$$

The schedule for the stochastic power plants also must assure that the scheduled output approximately equals the forecast:

$$C_{stoch} : \forall\, i,t : P^{t}_{target,i} \approx P^{t}_{pred,i}$$

The corridor is then defined by the conjunction of all the constraints.

$$INV_{RIA}(V) := C_{cons} \wedge C_{change} \wedge C_{balanced} \wedge C_{variance} \wedge C_{stoch}$$

The most common kinds of stochastic power plants are solar and wind power plants. Their output is directly dependent on the weather which thus has to be modeled within the system. It is captured in variables that are combined in the set *Weather*. The output of a stochastic plant is then a function of *Weather*.

The *noInterference* property of the AVPP states that it changes the schedule only after signaling it. The *quiescent state* of a power plant in this case only requires the power plant not to signal the end of a self-* phase itself. This is necessary in order to guarantee that the invariant holds whenever the end of a self-* phase is signaled. It is interesting to note that, in comparison to the agents in the production cell, the power plants can not simply stop whenever a constraint is violated. Instead, in the quiescent state, the power plants stick to their current schedule until a new schedule has been calculated. This behavior, however, has no influence on the verification approach.

3. Instantiate the Abstract Rely/Guarantees. The next step is to assign the variables to the different sets and to instantiate the generic rely/guarantee properties.

$$\begin{aligned} V_{int} &:= \{P_{max,i}, P_{min,i}, v_i, P_{pred,i}\} \\ V_{conf} &:= \{P_{target,i}\} \\ V_{rest} &:= \{P_{actual,i}\} \\ V_{env} &:= \{L_c, Weather\} \end{aligned}$$

The constants describing the physical limitations of the power plants are internal variables, like the maximal possible output. Also the forecast of the future output of a plant is an internal variable. These can not be changed by the environment or the AVPP. The configuration variables consist of the assigned schedule for each power plant. These can not be changed by the environment, but by the AVPP. Each power plant has a variable for the actual power output, which can be changed by the environment. This models failures such as a broken power generator or connection loss to the power grid which lead to a change in the actual output. The environment contains the consumer load and the variables describing the weather.

[3] For the verification this is formalized as the difference may not exceed a certain ε.

Observer/controller specification. Instantiating the generic rely/guarantee properties for the observer/controller we retrieve the specification of the AVPP's self-* behavior.

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, i\, :\, reconf_i \wedge \neg\, reconf'_i \rightarrow INV_{RIA}(V'_{all})) \\ & \wedge\, (\forall\, i\, :\, \neg\, reconf'_i \rightarrow noInterference(V_{all}, V'_{all})) \\ & \wedge\, (gridStability(V_{all}) \rightarrow gridStability(V'_{all})) \\ & \wedge\, Unchg_{sys}(V_{func} \setminus V_{conf}) \end{aligned}$$

The AVPP ensures that it always calculates valid schedules for the system, specified using the invariant $INV_{RIA}(V'_{all})$ and that it does only interfere in self-* phases. The AVPP also guarantees not to violate *gridStability* with its actions. As it does neither produce any power output nor consumes any power and a change of the schedule only comes into effect in the steps of the power plants, this is trivially true. The AVPP only changes the configuration variables, i.e., $P_{target,i}$, of the power plants. It relies on the power plants not to violate *gridStability* either and not to leave the quiescent state during a self-* phase.

R/G-specification of the functional system. The R/G-specification for the functional system is also received by instantiating the generic R/G property. As the functional system is composed of the individual power plants, the second decomposition step is to formulate R/G-Properties for the particular power plant. These local rely/guarantee properties are obtained as in the previous application by restricting the parts which are quantified over all power plants to the particular power plant. In this case, there is no direct interaction between the power plants themselves and thus no additional properties describing such a communication are necessary. The guarantee for a single power plant i then looks like:

$$\begin{aligned} G_i(V_{all}, V'_{all}) :\leftrightarrow \quad & P'_{target,i} = P_{target,i} \\ & \wedge\, (gridStability(V_{all}) \rightarrow gridStability(V'_{all})) \\ & \wedge\, (reconf_i \rightarrow quiescence(V_{func}, V'_{func}) \\ & \wedge\, (reconf_i \rightarrow reconf'_i) \end{aligned}$$

A power plant guarantees not to change the schedule on its own and not to violate the *gridStability* property. The quiescent state of a power plant is that it adheres to the "old" schedule as long as a reconfiguration takes place until the AVPP has finished writing the new schedule and that it does not abort reconfiguration on its own.

In order to be able to guarantee this it must rely on the AVPP not to change the schedule without notification. It further assumes that no internal variables are changed, e.g., the maximal and minimal power output of the plant.

$$\begin{aligned} R_i(V'_{all}, V''_{all}) :\leftrightarrow \quad & (\neg\, reconf''_i \wedge \neg\, deficient''_i \rightarrow Unchg_{env}(P_{target,i})) \\ & \wedge\, (deficient'_i \rightarrow deficient''_i) \\ & \wedge\, Unchg_{env}(\{P_{max,i}, P_{min,i}, P_{pred,i}, v_i\}) \end{aligned}$$

4. Verification. For each type of power plant implementation it has to be proven that it satisfies these R/G-properties. Analogously it must be verified that the AVPP's reconfiguration mechanism adheres to its specification and only calculates valid schedules. Then it can be deduced that for a finite but arbitrary number of power plants and an AVPP implementation that adheres to the specification, the *gridStability* property is maintained under the assumption that enough power is available in the whole system to fulfill the load. This rely is formulated in the global environment. In a more detailed model, the AVPP has the ability to throw-off load and therefore to control the load of the system in case the power available is insufficient.

Examples for Invariant Violations. The invariant can be violated in several ways. In case of a stochastic power plant, unforeseen environmental influences such as sudden weather changes can invalidate the forecast output which leads to a violation of C_{stoch} for this power plant. This can lead to a power deficit or surplus, both of which will have to be dealt with by rescheduling the available deterministic plants. A deterministic power plant is less likely to deviate from its prognoses, but it is still possible that the power plant goes offline unexpectedly. In this case, $C_{balanced}$ is violated and other plants have to be rescheduled to compensate for the missing power. A further violation can occur if the load suddenly changes and the reactive mechanism of the deterministic plants changes the actual power output. If this was not foreseen in the prognoses, this leads to a deviation form the scheduled target load of the plant. If this deviation is too big, either $C_{balanced}$ or $C_{variance}$ are violated.

In all cases the AVPP calculates a new schedule that is adapted to the new situation and which fulfills the invariant again. The reactive mechanism of the power plants ensures that grid stability is maintained. An invariant violation shows that the dynamics of the systems are unable to handle the current circumstances and adaptation is needed.

9 Conclusion and Outlook

This chapter presented an approach for formal modeling and compositional verification of self-* systems based on observer/controller-architectures which realize self-* capabilities by adapting configuration parameters of the participating components. The architecture separates the self-* and the functional behavior of the system. This separation is exploited by the Restore Invariant Approach in order to allow a separate verification of the functional and self-* part of the system. For the verification of the functional part a particular behavior of the observer/controller-layer is assumed and vice versa. This is specified by an invariant that defines the corridor of correct behavior in which functional correctness is ensured. The functional system is decomposed into properties over single agents. Compositional reasoning ensures the correctness of the overall system by proofs over single agents.

For the verification of the observer/controller behavior, verified result checking is applied. This allows moving the verification task to design time while correctness is assured during runtime. As correctness is ensured independently of the reconfiguration algorithms, these can be switched during runtime. This also allows the use of algorithms often used in self-* and nature-inspired systems that are neither sound nor complete.

By focusing on a specific architecture, generic properties that all applications have in common can be formulated. These can be instantiated for an application to retrieve the respective proof obligation. For the formal model and its verification, common formalisms and techniques are used that have successfully been applied to traditional systems. This allows to benefit of all advantages, like proof support, but still allows to express the important aspects of self-* systems. The framework includes explicit consideration of the environment, enabling reasoning about feedback loops which are a major aspect when considering self-* systems. However, the explicit model of the environment does not restrict the approach, as unforeseen changes of the environment are implicitly assumed when not specified otherwise. By providing such a fragmentary specification, uncertain behavior or arbitrary behavior can be modeled.

The approach was applied to two applications to illustrate the various aspects and differences, like continuous adaptation without the need of a strong quiescent state compared to alternating phases which need an explicit quiescent state for synchronization and correct reconfiguration.

In conclusion, the presented approach allows giving behavioral guarantees despite the self-* properties of the system. It provides a framework for formal modeling and verification of a system which enables formal proofs without restricting the flexibility of the system to adapt to unforeseen situations. This will hopefully raise the acceptance of self-* systems and facilitates the use of these techniques in safety-critical domains.

Future research will include the consideration of hierarchical architectures with a multi-layered observer/controller structure paired with the functional system, e.g., as presented by Müller-Schloer and Sick in [27]. Such an architecture can be beneficial in the AVPP scenario, where superordinate AVPPs could coordinate the actions of ones located on a lower level. Every level introduces new interaction possibilities that need to be considered in the formal model.

Another interesting extension of the presented approach is to allow the definition of soft corridors where a violation does not cause a reconfiguration right away but allows a fine-grained reaction to problems. The mentioned works in the field of runtime verification could provide useful instructions for the development of monitors in order to recognize a corridor violation quickly. An interesting challenge to solve is to synthesize appropriate monitors based on a given corridor specification and to find methods to decide on a violation early enough.

So far, the verification approach was only applied for safety properties which state that a property is never violated. A next step is to investigate the verification of liveness properties, like e.g., progress properties. This could be done in a similar fashion as presented in [43], where the rely/guarantee approach was already successfully applied to prove liveness properties of lock free algorithms. However, the verification of self-* systems is more complicated, as failures have to be considered. If failures can happen arbitrarily often, there is no possibility to ensure progress of the system. Therefore an approach must be found that restricts the environment, e.g., the frequency of failures. The challenge is to make assumptions that are realistic and still allow failures to happen.

Behavioral guarantees are a major step towards the acceptance of self-* systems and their application in safety-critical domains. Therefore the development of suitable techniques and tools is important. This chapter presented one approach to tackle these

challenges. However, it also illustrated how difficult the task is and that there are open questions that will need to be tackled in order to extend the scope of formal techniques to the full range of self-* systems.

Acknowledgments. This work has been sponsored by the German Research Foundation (DFG) in the projects SAVE ORCA as part of the priority program SPP 1183 Organic Computing and ForSA as part of the research unit FOR 1085 OC-Trust. The authors would like to thank Alexander Knapp and Bogdan Tofan for their feedback and the valuable discussions.

References

1. Anders, G., Seebach, H., Nafz, F., Steghofer, J.-P., Reif, W.: Decentralized reconfiguration for self-organizing resource-flow systems based on local knowledge. In: 2011 8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe), pp. 20–31 (April 2011)
2. Anders, G., Siefert, F., Steghöfer, J.P., Seebach, H., Nafz, F., Reif, W.: Structuring and Controlling Distributed Power Sources by Autonomous Virtual Power Plants. In: Proc. of the Power & Energy Student Summit 2010 (PESS 2010), pp. 40–42 (October 2010)
3. Anders, G., Hinrichs, C., Siefert, F., Behrmann, P., Reif, W., Sonnenschein, M.: On the influence of inter-agent variation on multi-agent algorithms solving a dynamic task allocation problem under uncertainty. In: Proceedings of the 2012 Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE Computer Society, Los Alamitos (2012)
4. Balser, M.: Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction. Ph.D. thesis, University of Augsburg, Augsburg, Germany (2005)
5. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 330–337. Springer, Heidelberg (1999)
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14 (2011)
7. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. In: Formal Aspects of Computing, FAC (2009)
8. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. European Journal on Artificial Interlligence (AI Communication) 23(2-3), 285–307 (2010)
9. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. ACM Press (2006)
10. Blum, M., Kanna, S.: Designing programs that check their work. In: STOC 1989: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing, pp. 86–97. ACM, New York (1989)
11. Branke, J., Mnif, M., Müller-Schloer, C., Prothmann, H.: Organic Computing - Addressing Complexity by Controlled Self-organization. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2006, pp. 185–191. IEEE (2008)
12. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)

13. Chandy, M., Misra, J.: An example of stepwise refinement of distributed programs: quiescence detection. ACM Trans. Program. Lang. Syst. 8, 326–343 (1986)
14. De Wolf, T., Holvoet, T.: Designing self-organising emergent systems based on information flows and feedback-loops. In: First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, pp. 295–298 (July 2007)
15. Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)
16. Fischer, P., Nafz, F., Seebach, H., Reif, W.: Ensuring correct self-reconfiguration in safety-critical applications by verified result checking. In: Proceedings of the 2011 Workshop on Organic Computing, OC 2011, pp. 3–12. ACM, New York (2011)
17. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. 31, 1–26 (1999)
18. Giese, H.: Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In: Kordon, F., Sztipanovits, J. (eds.) Monterey Workshop 2005. LNCS, vol. 4322, pp. 258–280. Springer, Heidelberg (2007)
19. Güdemann, M., Ortmeier, F., Reif, W.: Safety and Dependability Analysis of Self-Adaptive Systems. In: Proceedings of ISoLA 2006. IEEE CS Press (2006)
20. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM Corporation (2006)
21. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
22. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Softw. Eng. 16, 1293–1306 (1990)
23. Kramer, J., Magee, J.: Analysing dynamic change in distributed software architectures. IEE Proceedings Software 145(5), 146–154 (1998)
24. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study, pp. 91–100 (1998)
25. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
26. Misra, J., Chandy, K.M.: Proofs of Networks of Processes. IEEE Transactions on Software Engineering SE-7(4), 417–426 (1981)
27. Müller-Schloer, C., Sick, B.: Controlled emergence and self-organization. In: Organic Computing. Understanding Complex Systems, vol. 21, pp. 81–103. Springer, Heidelberg (2008)
28. Murch, R.: Autonomic Computing. IBM Press (2004)
29. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A Universal Self-Organization Mechanism for Role-Based Organic Computing Systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 17–31. Springer, Heidelberg (2009)
30. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.P., Reif, W.: A generic software framework for role-based Organic Computing systems. In: SEAMS 2009: ICSE 2009 Workshop Software Engineering for Adaptive and Self-Managing Systems (2009)
31. Nafz, F., Seebach, H., Steghöfer, J.P., Anders, G., Reif, W.: Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing - A Paradigm Shift for Complex Systems. Autonomic Systems, vol. 1, pp. 79–93. Springer, Basel (2011)
32. Nafz, F., Seebach, H., Steghöfer, J.-P., Bäumler, S., Reif, W.: A Formal Framework for Compositional Verification of Organic Computing Systems. In: Xie, B., Branke, J., Sadjadi, S.M., Zhang, D., Zhou, X. (eds.) ATC 2010. LNCS, vol. 6407, pp. 17–31. Springer, Heidelberg (2010)
33. Pissias, P., Coulson, G.: Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. Iet Software/IEE Proceedings - Software 2, 348–361 (2008)

34. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for Organic Computing. In: INFORMATIK 2006 – Informatik für Menschen!, vol. P-93, pp. 112–119 (2006)
35. Rochner, F., Müller-Schloer, C.: Emergence in Technical Systems. it - Information Technology 47(4), 195–200 (2005)
36. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of Temporal Representation and Reasoning (TIME). IEEE, CPS (2011)
37. 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, 10:1–10:32 (September 2010)
38. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: A software engineering guideline for self-organizing resource-flow systems. In: IEEE International Conference on Self-Adaptive and Self-Organizing System (SASO), pp. 194–203. IEEE Computer Society, Los Alamitos (2010)
39. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: How to Design and Implement Self-organising Resource-Flow Systems. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing - A Paradigm Shift for Complex Systems, Autonomic Systems, vol. 1, pp. 145–161. Springer, Basel (2011)
40. Shehory, O., Kraus, S.: Methods for task allocation via agent coalition formation. Artificial Intelligence 101(1-2), 165–200 (1998)
41. Smith, G., Sanders, J.W.: Formal Development of Self-organising Systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 90–104. Springer, Heidelberg (2009)
42. Sterman, J.D.: Business Dynamics – Systems Thinking and Modeling for a Complex World. McGraw-Hill (2000)
43. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal Logic Verification of Lock-Freedom. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 377–396. Springer, Heidelberg (2010)
44. Tsang, E.: Foundations of Constraint Satisfaction. Computation in Cognitive Science. Academic Press, Inc., London and San Diego, USA (1993)
45. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: An alternative to quiescence: Tranquility. In: 22nd IEEE International Conference on Software Maintenance, ICSM 2006, pp. 73–82 (September 2006)
46. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. J. ACM 44(6), 826–849 (1997)
47. Wooldridge, M.J., Dunne, P.E.: The Computational Complexity of Agent Verification. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 115–127. Springer, Heidelberg (2002)
48. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 371–380. ACM, New York (2006)
49. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD 2009, pp. 161–172. ACM, New York (2009)