A formal framework for compositional verification of organic computing systems

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

Angaben zur Veröffentlichung / Publication details:

Nafz, Florian, Hella Seebach, Jan-Philipp Steghöfer, Simon Bäumler, and Wolfgang Reif. 2010. "A formal framework for compositional verification of organic computing systems." In *Autonomic and Trusted Computing: 7th International Conference, ATC 2010, Xi'an, China, October 26-29, 2010, proceedings*, edited by Bing Xie, Juergen Branke, S. Masoud Sadjadi, Daqing Zhang, and Xingshe Zhou, 17–31. Berlin: Springer. https://doi.org/10.1007/978-3-642-16576-4_2.

A Formal Framework for Compositional Verification of Organic Computing Systems*

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

Department of Software Engineering and Programming Languages,
University of Augsburg, 86135 Augsburg, Germany
{nafz,seebach,steghoefer,baeumler,reif}@informatik.uni-augsburg.de

Abstract. Because of their self-x properties Organic Computing systems are hard to verify. Nevertheless in safety critical domains one may want to give behavioral guarantees. One technique to reduce complexity of the overall verification task is applying composition theorem. In this paper we present a technique for formal specification and compositional verification of Organic Computing systems. Separation of self-x and functional behavior has amongst others, advantages for the formal specification. We present how the specification of self-x behavior can be integrated into an approach for compositional verification of concurrent systems, based on Interval Temporal Logic. The presented approach has full tool support with the KIV interactive theorem prover.

Keywords: Organic Computing, Formal Methods, Compositional Reasoning.

1 Introduction

In Organic Computing (OC) systems [24], a potentially vast number of components interact with each other and make local decisions in order to fulfill global goals. Such systems are highly desirable as they exhibit characteristics of self-organization and are therefore highly resilient, adaptive and robust. Therefore, they should be ideally suited for environments in which safety is a critical concern and those characteristics are requirements of the domain.

However, such domains usually require a rigorous process of analysis and verification in order to get a system approved for deployment. In automotive and aviation systems, certification authorities require proof that a system behaves safely under all circumstances. Such a proof is more often than not provided by formal analysis and verification. But due to the dynamic, complex, and highly-parallel nature of Organic Computing systems, such an analysis is extremely hard to perform and many tools and techniques (e.g., model checking) are not suitable for sufficiently large examples.

This paper introduces a different approach: "conventional" complex systems have been analyzed with compositional methods in which parts of the system are regarded separately before the analyses are combined to make statements for the entire system.

* This work is partly sponsored by the German Research Foundation (DFG) in the special priority program SPP 1183 "Organic Computing".

This compositional and local reasoning is now applied to OC-systems whose correct behavior can be expressed by rely/guarantees [18,22,7]. These systems can be modularized in a natural way as they usually consist of several components (e.g. agents).

For this purpose, self-organization behavior and functional behavior are regarded separately. The functional behavior can then be expressed as rely/guarantees and be verified with a compositional technique. Especially one wants to guarantee that after a reconfiguration the system again works as intended. This property is a safety property [2]. In this paper we present an approach for verification of this kind of properties. The specification of functional behavior with rely/guarantees for systems and the integration of the self-x behavior into the environment is shown. The approach is applied to self-organizing resource-flow systems, a system class in which self-x principles are beneficial.

Previous work included formal analysis of failure modes for OC-systems which only applied to specific instances of a system and was therefore limited [15]. Further, separation of self-organization and functional behavior was not considered. The general approach to formally specify a system with invariants which express correct system behavior has been detailed in [14] and a way to use these invariants for a reconfiguration mechanism has been proposed in [25]. The present paper builds on these foundations and provides the formal framework which is necessary to thoroughly and rigorously analyze and verify such OC-systems.

The paper is structured as follows: Section 2 describes the formal framework as the foundation of the verification approach which is introduced in Section 3. The technique is then applied to self-organizing resource-flow systems in Section 4 where a formal model is given and the necessary specification and verification steps are sketched. Section 5 compares the approach with related work before the paper closes with a discussion of the benefits and limitations of the proposed framework and an outlook to future work.

2 Formal Framework

In this section we provide an overview of the formal framework we use for modeling and verification of Organic Computing systems. We will start by giving an introduction into the logic framework and its semantics. Afterwards we show how to define behavioral corridors on the system model to exclude unwanted behavior. This technique also allows to separate functional behavior from self-x behavior of the system, which has several advantages for system verification.

2.1 Temporal Logic Framework

In the following, an informal overview over the temporal logic calculus used is given, which is also integrated in the interactive theorem prover KIV [5]. A formal semantics and a detailed description can be found in [6,4].

The used temporal logic (ITL^+) is a variant of interval temporal logic (ITL) [23,9] that is extended by explicitly including the behavior of the environment into each step. Further ITL^+ combines temporal formulas and program constructs within the same

formalism. The basis for ITL$^+$ are infinite sequences[1] of states, which are called *traces* or *intervals*. A state is defined by one evaluation $e \in eval(v_1, ..., v_n)$, where $eval(V)$ is the set of all possible evaluation of the variables $v_i \in V$.

In our setting we introduce an additional intermediate state $'_i$ to distinguish between system and environment transitions. Therefore besides variables v there are also *primed* v' and *double primed* v'' variables. For each variable v *in* V there is a corresponding primed and double primed variable. The sets of all primed/double primed variables is denoted accordingly by V' and V''. The relation between v and v' is called *system transition*, whereas the relation between v' and v'' *environment transition*. The value of v'' in a state must be equal to the value of v in the next successive state ($''_0 = {}_1$). Thereby the system and the environment transition alternate. For an intuition this is depicted in Figure 1.

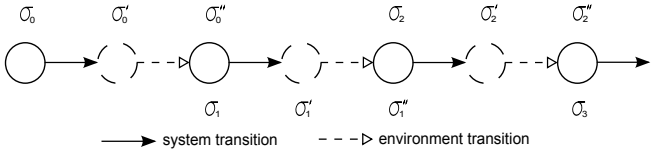

Fig. 1. A trace as sequence of states

The explicit inclusion of the environment allows for a separation of system and environment. Further, an arbitrary environment is considered without stipulating syntactic restrictions for the formula describing the system behavior. Especially in the case of OC-systems which interact with their environment, an explicit model of the behavior of the environment is advantageous [30]. It allows a detailed modeling of the environments properties and the interaction between system and its environment.

The logic contains the standard predicate logic operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (equivalence) and quantifiers $\forall, \exists$. Predicate logic formulas however are only evaluated over a triple of states ${}_i$, $'_i$ and $''_i$. For example $p(V, V')$ denotes a predicate which is evaluated over the unprimed and primed state. To express properties over intervals the following operators – which are also standard operators in linear temporal logic (LTL) – can be used.

	holds now and *always* in the future
	holds now or *eventually* in the future
until	eventually holds and holds *until* holds
unless	always if does not hold, then holds or held earlier
$\circ$	there is a next step which satisfies (*strong next*)
last	the current state is the *last*
${}_1 \parallel {}_2$	*interleaving*

Further, the formulation of programs in SPL (Simple Programming Language) [21], a program like syntax is supported. The semantics of both formulas and programs can be

[1] For simplicity we assume that the systems have no terminal states as it does not impose any serious restrictions. Therefore all traces we consider are infinite.

expressed as a set of traces. In ITL$^+$ programs and temporal formulas can be mixed. This can be used for the parallel composition of programs with the interleaving operator [4]. The calculus supports symbolic execution of parallel programs, which is a successful technique for interactive verification (e.g. Dynamic Logic [16,17]). It is a very intuitive strategy for programs as the proof advances step by step similar most humans do it when trying to understand a program [20]. Furthermore, it can be automated to a large extend.

2.2 Organic Computing Systems

In self-x systems - just like in traditional systems - failures and environmental disturbances can not be prohibited. Disturbances and failures force the system into a state where it can not provide its functionality. Therefore we can distinguish the state space of a system into two sets.

- A set S_{func} of *functional* states, in which the system can provide the desired functionality,
- A set S_{reconf} of *erroneous* or *reconfiguration* states in which the system can‘t provide the functionality and a reconfiguration has to take place to get back to a state within S_{func}.

For those sets $S := S_{func} \cup S_{reconf}$, *with* $S_{func} \cap S_{reconf} = \emptyset$ holds. In opposition to most traditional systems self-x systems are characterized by their ability to compensate disturbances. Traditional systems without self-x properties and any degree of freedom have traces in which the system switches to an error state s_{err}, for example caused by component failures or other environmental influences. In that state the traditional system can’t provide its functionality anymore and without restarting it will never return to a functional state. The OC-system, however, has the ability to compensate the failure (e.g. by self-reconfiguration) and get back into a functional state, where it can again meet its requirements.

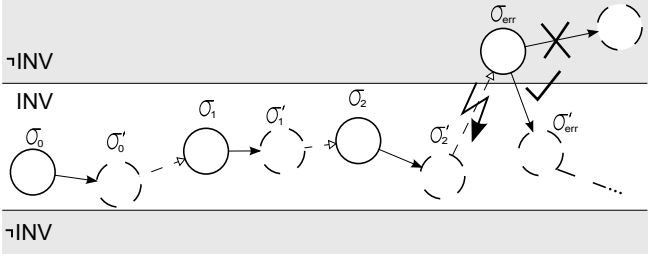

Fig. 2. Behavioral Corridors and Traces

In Figure 2 a possible trace of a self-x system is shown. In this example the set of functional states is $S_{func} = \{ \sigma_0, \sigma'_0, \sigma_1, \sigma'_1, \sigma_2, \sigma'_2, \sigma'_{err}, ...\}$ and there is one error state in $S_{reconf} = \{ \sigma_{err}\}$. The system switches via an environment transition (e.g. a component failure) into an error state σ_{err}. The self-x system then starts a self-organization and reaches a state σ'_{err} in which functional correctness can be assured again.

2.3 Specification of Functional Corridors

The last example leaves some open questions. For example, it is unclear how to distinguish functional states from erroneous states in which reconfiguration has to occur. Further, we want to restrict the system transitions in such a way that it always stays in functional states and tries to get back into a functional state, if not in one. The proposed technique is called *restore invariant approach* and is described informally in [14]. The idea is to define behavioral corridors, by defining a predicate $INV(V)$ which holds in all functional states and does not hold otherwise. The predicate can in some sense be seen as an invariant, as the systems goal is that this invariant holds on the entire trace.

In the above OC-system the set of functional states is then defined by $S_{func} := \{ \in S | INV( ) \}$. As long as *INV* holds the system is in a state within the corridor, whenever it is false the system has left the corridor and needs to get back. Traces only consisting of states out of S_{func} are in some sense "good" traces within the corridor. It is desired that an OC-system has only traces that consist of states out of S_{func} or whenever a failure occurs and it enters a state out of $S_{econf} := S \setminus S_{func}$ there will eventually be some state $s \in S_{func}$ later in the trace. This property can be expressed as a temporal logic formula $INV \vee (\neg INV \rightarrow \diamond INV)$. This formula can also be used to specify the self-organization (SO) mechanism of the system, as it describes what the effect of the self-organization is. Usually self-organization after a component failure decreases redundancy, as hardware components that broke can't be recreated, but the systems functionality can still be provided as another component can take over, for example. This also means that there is some point where restoration of the invariant is not possible anymore. For a realistic (non-perfect) self-organization we need to modify the specification of the SO-mechanism by adding a predicate stating that no solution is possible anymore or weaker, no solution was found. With some simplifications the specification of the reconfiguration then looks as follows:

$$(\neg INV \rightarrow \diamond(INV \vee))$$

can be seen as some kind of quality predicate as depending on how it is formulated some algorithms can fulfill the specification or not. For example, the weakest is stating "Algorithm result is, 'found no solution'" whereas the strongest is $= false$ ("Reconfiguration is always possible and successful'). Of course, one wants to have something in between like "As long as enough redundancy is available" or "As long as a functional state is reachable". An example is given in the case study.

The idea of the *restore invariant approach* is reflected in a two-layered architecture (see [14]). One layer is the observer/controller (o/c) layer, which is responsible for the self-x intelligence of the system and incorporates the self-organisation mechanism, for instance. It further observes the invariant and starts a self-organization phase whenever it is violated. This o/c can be either implemented as a central o/c agent or as several o/c agents on top of each component of the functional system. For an implementation o/c_{impl} the self-organisation property for a defined must hold and be proven.

$$o/c_{impl} \models (\neg INV(V) \rightarrow \diamond(INV(V') \vee)) \quad (1)$$

The second layer is the functional layer, which contains the functional part of the system. The functional layer does not necessarily consist of one monolithic component.

Especially in a self-x system it will be composed out of several components (one may say ”agents“) providing the functionality. The presented approach will show how component specifications can be composed to one global system.

3 Formal System Specification and Verification

In this section first we will briefly describe the rely/guarantee view on systems, which is common in compositional reasoning. This allows us to reason about individual components and derive global system properties by doing so. Therefore we present the used compositional theorem. Afterwards we describe the idea of the formal approach and how the separation is advantageous for the compositional specification.

3.1 Rely/Guarantee Specifications

The behavior of a system is specified as a transition system described by the formula $G(V,V')$. This expresses the guarantee the system gives. To be able to guarantee its specified behavior the system relies on a certain, but not necessarily completely fixed or predefined behavior of its environment. In an arbitrary environment a system will not be able to give any guarantees. The environment is specified by a transition formula $R(V',V'')$. A typical property of the environment R is that the local variables of the particular system are not changed $R(V',V'') :\Leftrightarrow V' = V''$. The system behavior is then formulated using the “sustain”-operator $^+$, which is used in most rely/guarantee based compositional proof techniques.

$$R(V',V'') \ ^+ \ G(V,V')$$

Informally the formula means, that if rely R holds up to step i, then guarantee G must hold up to step $i+1$. It allows to formulate that a system or component violates its guarantee G only after its assumption R is violated. This is needed to break circularity when applying compositional reasoning. Guarantees are formulated as propositional predicates over unprimed and primed variables, while for relies predicates over primed and double primed variables are used. In this way it can be formalized which steps are allowed for the system and which for the environment. The $^+$ operator therefore can be derived using the standard TL **unless** operator:

$$R \ ^+ \ G := G \ \textbf{unless} \ (G \wedge \neg R)$$

3.2 Modularization

Usually systems consist of several components running in parallel. A component Ag_i is specified in the same way by a local rely R_i and a local guarantee G_i. Hence, from a point of view of each component, the other components are in its environment. The local rely can therefore also contain some properties assuming “good“ or also ”bad” behavior of the other components in the system. To be able to give global guarantees by local reasoning we use a compositional theorem. Details and proofs can be found in [6].

Theorem 1. *If:*

i. *for all* $i = 1,\ldots,n$: $G_i(V,V') \vdash G(V,V') \wedge \bigwedge_{j\in\{1..n\}\wedge j\neq i} R_j(V,V')$
ii. *for all* $i = 1,\ldots,n$: $R_i(V,V') \wedge R_i(V',V'') \vdash R_i(V,V'')$
iii. $R(V,V') \vdash \bigwedge_{i\in\{1..n\}} R_i(V,V')$

then: $R_1(V',V'') \ ^+ \ G_1(V,V') \parallel \ldots \parallel R_n(V',V'') \ ^+ \ G_n(V,V') \vdash R(V',V'') \ ^+ \ G(V,V')$

Premises *i* - *iii* contain only predicate logic formulas, which are considerably easier to be proven, than interleaved temporal logic formulas. These three proof obligations have the following informal meaning:

i. The guarantee of each component preserves the global guarantee and does not violate the assumptions of all other components.
ii. The assumptions of all components are transitive. With this property, the components assumption is preserved even if other components make several steps.
iii. All component assumptions hold if the global assumption holds. Therefore, no component assumption is violated in the environment step.

The use of this theorem enables the proof of a global rely/guarantee (r/g) property by reasoning over local r/g properties for the individual components. If the system consists of identical components of the same type only the premises for one of this components have to be proven. The theorem allows then to reason about a system with an arbitrary number of these components. The theorem was proven with KIV and can therefore be applied directly during a proof.

3.3 Organic Computing System – Specification and Verification

The global behavior of an OC-system is formulated by a global r/g property $R_{ocsys} \ ^+ \ G_{ocsys}$. One big advantage of the separation (Sect. 2.3) of the complete system into an observer/controller (o/c) and a functional system (sys) is, that from the point of view of the functional system, the o/c is contained in the environment, and vice versa. Using the compositional approach we consider both parts as abstract components which run in parallel, specified by local r/g properties. The according proof obligation for the separation then looks like:

$$\begin{aligned} & R_{o/c}(V',V'') \ ^+ \ G_{o/c}(V,V') \parallel R_{sys}(V',V'') \ ^+ \ G_{sys}(V,V') \\ \vdash\ & R_{ocsys}(V',V'') \ ^+ \ G_{ocsys}(V,V') \end{aligned} \tag{2}$$

The behavior of the functional system is specified as a transition system described by the formula $G_{sys}(V,V')$. This expresses the guarantee the functional system gives. Hence, for the system verification, the specification of the self-organization mechanism can be integrated into the environment, which can be assumed by the functional system.

$$R_{sys}(V',V'') :\Leftrightarrow (V' = V'') \vee SO(V',V'')$$

This formula states that either no changes are made to the system variables V (no self-organization takes place) or in case a self-organization takes place the variables are

changed according to $SO(V',V'')$. Here the effect of a reorganization is assumed to occur in one transition. This does not mean that the self-organization takes one step, just that the effect is seen by the agent in one point of time. An o/c implementation needs to reflect this, which is expressed by refining formula (1).

$$o/c_{impl} \models \quad (\neg INV(V) \rightarrow (V = V')) \textbf{ until } \quad (INV(V') \vee \quad) \\ \wedge \, (\, INV(V) \rightarrow (V = V'))$$

For the verification of safety properties [2] it is sufficient to use the safety closure here, and leave the liveness part of the formula beside[2]. Based on this $SO(V',V'')$ can be specified as

$$SO(V',V'') :\Leftrightarrow \quad (\neg INV(V') \rightarrow (V' = V'' \vee INV(V'') \vee \quad) \\ \wedge (INV(V') \rightarrow (V' = V''))$$

Informally the specification of the functional system states that under the assumption of a correct self-organization algorithm and environment R_{sys}, the system will guarantee G_{sys}. This specification can later be replaced by an actual implementation, for example given as a pseudo program $Prog_{sys}$, whose syntax is supported in ITL$^+$. This leads to the following proof obligation for the implementation.

$$Prog_{sys} \vdash R_{sys}(V',V'') \;^{+}\; G_{sys}(V,V') \tag{3}$$

The o/c layer is defined similar. It needs to guarantee the assumed self-organization for the functional system $SO(V,V')$. Further it relies on the global environment to not change its local variables.

$$o/c_{impl} \vdash R_{o/c}(V',V'') \;^{+}\; G_{o/c}(V,V') \tag{4}$$

For a particular system usually some more concrete assumptions and guarantees are made, like the particular effect of the o/c layer on the functional system. Further, the global environment usually is allowed to change some of the variables, in order to model failures. We will have a closer look on this in the next section when applying the approach to a case study.

Verification: Besides the verification of (3) and (4), the derived proof obligations when applying the compositional theorem to (2) need to be verified. As both parts of an OC-system usually consist of several components again, the strategy is to also use modularity for verification of the local relies of o/c and sys.

4 Self-organizing Resource Flow Systems

In the previous section the general approach was described. In this section we want to focus on the class of self-organizing resource-flow systems to demonstrate the

[2] For each formula exists formulas s and l, where s is a safety and l is a liveness formula, so that following equation holds: $\leftrightarrow s \wedge l$. The safety closure of is the strongest formula s that satisfies this equation [1].

application of the presented approach. For the sake of brevity, we will present the compositional verification of the functional part of the system. But as the relies of the functional systems are the guarantees of the o/c layer, all interesting issues should be tackled.

4.1 Design of Self-organizing Resource-Flow Systems

The components of resource-flow systems can be described by a pattern called Organic Design Pattern (ODP-RFS) as seen in Fig. 3.

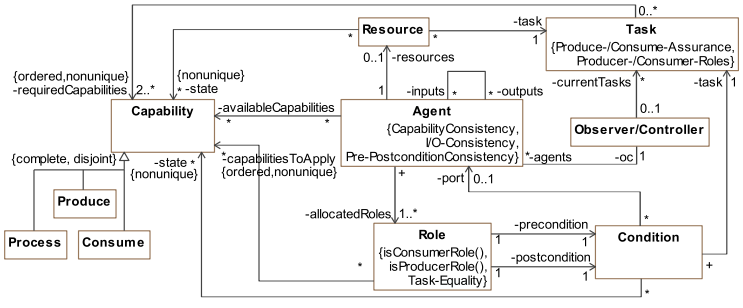

Fig. 3. Components of Resource-Flow Systems

Agents are the main components in these systems, processing the *resources* according to a given *task*. Every *agent* has several *capabilities*, divided into producing, processing, and consuming *capabilities* (*produce*, *process*, and *consume*). Consequently, the *task* is a sequence of *capabilities* beginning with a producing *capability* and ending with a consuming *capability*. Furthermore, the agent knows a couple of agents he can interact with and hand over *resources*. This is encapsulated in the *inputs* and *outputs* relation. The *role* concept is introduced to define correct resource-flows through the system. This means an *agent* has *roles* allocated telling him from which *agent* he receives the *resource* (*precondition/port*), which *capabilities* to apply, and then to which *agent* to hand over the *resource* (*postcondition/port*). Thus, the *roles* establish the connections between the *agents* and the combination of all *roles* forms the resource-flow. The OCL-constraints in Fig. 3 specify correct allocations of *roles* to *agents* and are used in the formal model of the resource-flow systems to specify the self-organization mechanism, as described in the next section. For more details on the SE process and modeling of self-organizing resource-flow systems see [26]. In this case study, self-organziation is done by role allocation. In case of a failure the system calculates a new valid role allocation, to be able to fulfill the task again.

4.2 Formal Model and Verification

The functional part of the system model described above, is formally represented as an abstract resource-flow system by defining data types for all the concepts and their relations. For instance, the static part of an agent is represented as a tuple

$$agent := (\quad .id : Nat \times .availableCapabilties : list(capability) \times .inputs : list(agent) \\ \times .outputs : list(agent) \times .allocatedRoles : set(role))$$

".identifier : type" splits into a selector *identifier* and the type of this element of the tuple. E.g. for an agent Ag, $Ag.availableCapabilties$ selects the list of capabilities the agent Ag has. The agent is a five-tuple consisting of a natural number as identifier, a list of the capabilities it can perform, two lists of agents for possible inputs and outputs and a set of roles which are currently assigned to this agent. To express the location of the individual resources we use a so called *store*, which is a data structure (associative array) mapping agent identifiers to resources $locST : Nat \Rightarrow Resource$. The resource at agent id is selected by $locST[id]$. If there is no resource a special resource value $\perp$ is returned. A role is represented as a tuple:

$$role := (\quad .precondition : condition \times .capabilitiesToApply : list(capability) \\ \times .postcondition : condition)$$
$$condition := (.port : agent \times .state : list(capability) \times .task : Task)$$

Task, *Resource* are defined in the same way. Capabilities are defined abstractly as sorts. The set *Agents* contains one variable of type *agent* for each agent in the system. *#Agents* is then the amount of agents in system. In this case study we verified a system with one simultaneous task in the system. As described in Section 2.3 we strictly separate the self-x from the functional behavior. For the verification of the functional system only the specification of the self-organisation is used and its implementation is verified separately.

Global system behavior: In the example one global guarantee the system should give is that resources are always processed according to their task. Formally speaking for every key m of the location store holds that the state of the associated resource is a prefix of the task.

$$G(V,V') :\Leftrightarrow \forall\ m : (m \in locST) \rightarrow \\ locST[m].state \sqsubseteq locST[m].task \rightarrow locST'[m].state' \sqsubseteq locST'[m].task'$$

The behavioral corridor for that class of system is specified as $INV(V)$. It expresses a valid role allocation. For instance one part of INV is stating that only capabilities can be assigned that are available.

$$INV_i(V) :\Leftrightarrow \forall ag \in Agents, \forall r \in ag.allocatedRoles : \\ r.capabilitiesToApply \subseteq ag.availableCapabilties$$

The complete invariant is a conjunction of several predicate logic formulas like this one. They can also be derived from the ODP-RFS, by translating the OCL constraints into the equivalent predicate logic formula. [25] provides more details about defining corridors for self-organizing resource-flow systems and how the constraints look like. Further a possible realization via a constraint solver is presented. On part of in this example is defined as:

$$:\Leftrightarrow \exists c \in Task, \neg\exists ag \in Agent : c \in ag.availableCapabilities$$

This states that for at least one needed capability, there is no agent that can perform it. $SO(V,V')$ is derived by substitution of (1). Besides the specification of the self-organization we assume that the environment does not change internal system variables, except *availableCapabilities*. This is done by specifying a conjunction $\bigwedge_{w \in V \setminus L} w' = w''$ of all variables, except the variables in L, which in this case consist of all *available-Capabilities* of *Agents*. The environment is allowed to arbitrarily take or even give capabilities to the agents. Together with $SO(V',V'')$ this forms the rely for the complete system.

Local behavior: Next we define the r/g specification for a single agent. Besides the global rely, a single agent also relies on good behavior of the other agents. That means e.g. it assumes that another agent is not taking its resource away or changing it. Only during a reconfiguration this is allowed. For technical reasons we introduce a variable *reconfCnt* which is counting the reconfigurations, indicating if there was a reconfiguration or not. The rely R_i for an agent i is below. For better readability of the formulas we use a special formatting and the following abbreviations: *allocR := allocatedRoles*, *prec := precondition* and *postc := postcondition*. *capToApp := capabilitiesToApply*.

$$
\begin{aligned}
R_i := \quad & (\neg\, isEmpty(locST'[allocR.prec.port'] \\
& \wedge\, reconfCnt' = reconfCnt'') \\
& \rightarrow locST'[allocR.prec.port'] = locST''[allocR.prec.port'']\ ) \\
\wedge \quad & (isEmpty(locST'[allocR.postc.port']) \\
& \rightarrow isEmpty(locST''[allocR.postc.port''])) \\
\wedge\ & (reconfCnt' \leq reconfCnt'') \\
\wedge\ & (SO(V,V'))
\end{aligned}
$$

Note that only a part of the global rely is included as from a local point of view, more variables are allowed to change (e.g. location store). Under this assumption an agent guarantees that it will process the resource correctly.

$$
\begin{aligned}
G_i := (\quad & \neg\, isEmpty(locST[allocR.prec.port]) \\
& \wedge\, isEmpty(locST[allocR.postc.port]) \\
\rightarrow\ & locST = locST' && (1) \\
& \vee (\quad locST[allocR.prec.port].state = allocR.prec.state && (2) \\
& \quad \wedge\, locST[allocR.prec.port].task = allocR.prec.task \\
& \quad \wedge\, locST'[allocR.prec.port].state := \\
& \qquad\qquad locST'[allocR.prec.port].state + allocR.capToApp\ ) \\
& \vee (\quad locST[allocR.prec.port].state = allocR.postc.state && (3) \\
& \quad \wedge\, locST[allocR.prec.port].task = allocR.postc.task \\
& \quad \wedge\, locST'[allocR.postc.port] := locST[allocR.prec.port] \\
& \quad \wedge\, locST'[allocR.prec.port] := \perp\ )
\end{aligned}
$$

Formula G_i describes the behavior of an agent i. If there is a resource in its input port waiting to be process and the outgoing port is free the agent can ...

(1) ... do nothing.
(2) ... process it according to the role, if the resource has a state and task that fits to its assigned role.
(3) ... give it to the next agent if the workpiece is already processed.

$R_i \xrightarrow{+} G_i$ describes the guaranteed behavior of a particular agent i. In this case study we have nearly homogeneous agents, therefore most agents have the same behavior. The only slight difference in agent behavior is that some agents have a producer or consumer role. That means they create or remove resources from the system. Here the rely/guarantee looks analogous except that a producer agent doesn't have to wait for an incoming resource and a consumer agent doesn't have to forward the resource.

A system with one producer, one consumer and n-process agents can than be specified by interleaving their local rely-guarantee formulas.

$$SYS_{RFS} := R_{producer} \xrightarrow{+} G_{producer} \parallel R_1 \xrightarrow{+} G_1 \parallel ... \parallel R_n \xrightarrow{+} G_n \parallel R_{consumer} \xrightarrow{+} G_{consumer}$$

The resulting sequence we want to prove is then given by:

$$SYS_{RFS} \vdash R(V', V'') \xrightarrow{+} G(V, V')$$

Applying Theorem 1, we get a total of seven proof obligations. For premise (*i*) and (*ii*) we get one proof obligation for each type of agent (producer, consumer, process). Further we get one proof obligation for premise (*iii*).

Verification: All proof obligations were formally proven with the interactive theorem prover KIV, which fully supports higher order logic, concurrency and the presented temporal logic. As the resulting proof obligations are all predicate logic and the agents are no more interleaved after modularization, the proofs are straightforward. They start with a case distinction of the conjunctions on the right-hand side of the sequence. Most premises can then be closed by the simplifier of KIV automatically. Only some interaction for reasoning over the location store was needed.

5 Related Work

There is a lot of work done in the area for compositional reasoning for concurrent systems in general. Cau and Collette [10], use a similar technique for defining relies and guarantees but without the focus on a calculus and tool support. Solanki et. al. [28] use compositional reasoning together with ITL. They use a rely/guarantee variant that allows guarantees to be formulated in ITL. The tool they use (ana)Tempura [9,23]. This technique is applied to a semantic web service description. Both do not consider self-x properties, like self-reconfiguration of the system.

Formal verification of self-x systems is a young research area [11]. But there is already some interesting work done on this topic. In [30] Wooldridge states that for the verification of agents the environment is essential. He presents a formal model where the behavior of an agent within an environment is described as a sequence of interleaved environment states and agent actions. This is similar to the idea of ITL$^+$ and also allows

for explicit modeling of the environment. We further show the integration of this idea into a modular approach for compositional reasoning.

In [13] a Temporal Belief Logic (TBL), was introduce to define semantics and formalize and verify properties of systems specified in the *Concurrent METATEM* language. Here modal operators are used to distinguish knowledge of different agents. The main difference to our approach is that we utilize compositionality and can do local reasoning. In [3] a policy-based modeling approach based on PobSAM is presented, which also proposes a separation of the self-x part from the functional part of the system. Further an operational semantics based process algebras is presented, but they do not consider compositionality. In [29] the specification language ASSL and a tier-based framework for specification of autonomic systems is presented. Formal verification is not considered.

In the area of compositional verification of multi-agent systems [12] presents a formalization using Temporal Multi-Epistemic Logic (TMEL). Here the system structure is exploited for compositional verification and their proofs were constructed by hand. In [8,19], Jonker et al. present a compositional approach for one-to-many negotiation protocols of agents. They verify one abstraction level against another level until they reach a so called primitive component, where they can apply standard verification techniques. For specification of dynamic properties they use different variants of temporal logic, depending on the type of properties.

Smith and Sanders present an incremental top-down approach for the formal development of self-organizing systems in [27]. The verification of the global system is done by verifying all refinement steps down to component level. Their work provides good strategies for the refinement between different abstraction layers, which also could be used for further refinement on the component level in this paper. While in this paper a strict separation of self-x and functional behavior is assumed, they make no explicit distinction.

6 Summary and Outlook

In summary, we have presented a method for formal specification and compositional verification of systems with self-x properties. We first presented how the separation of the self-x mechanism and the functional system can be used to split the verification task into a verification of the particular self-x algorithm and a verification of the functional system by using only the specification of the self-x algorithm. Furthermore a technique for compositional verification of such systems is presented. As basis for this work an ITL variant [4] that provides a compositional interleaving operator and a rely-guarantee theorem for modularization is used. The logic is fully integrated into the interactive theorem prover KIV and all proofs where done within this tool. The advantage of the use of an interactive theorem prover is, that it can deal with infinite datatypes, because it provides powerful techniques for abtraction.

The approach is applied to a resource-flow system as case study. This case study was chosen as therefore already a software engineering process and design framework was developed and a continuous approach including verification was desired. Nevertheless the approach is applicable to all kind of sytems fitting into the two layered architecture. The main advantage of these technique is that it is independent of the number of

agents. Only one rely-guarantee for each agent type is needed. Further for a particular implementation only local reasoning is needed.

Next steps are to extend this approach to liveness properties. So, we can prove properties like, “eventually the system will produce some output”, so it shows some progress. This mainly depends on the behavior of the environment, as it can interfere with every progress by breaking enough of the system. Here, the environment needs to be adapted to allow expressions like “if long enough nothing breakes”. First experiments in this direction were very promising.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. 17(3), 507–535 (1995)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3) (1987)
3. MohammadReza, M.: PobSAM: Policy-based managing of actors in self-adaptive systems. ENTCS. Elsevier Science B.V., Eindhoven (2009)
4. Balser, M.: Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction. PhD thesis, University of 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. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications 23(2-3), 285–307 (2010)
7. Bäumler, S., Nafz, F., Balser, M., Reif, W.: Compositional proofs with symbolic execution. Ceur Workshop Proceedings, vol. 372 (2008)
8. Brazier, F.M.T., Cornelissen, F., Gustavsson, R., Jonker, C.M., Lindeberg, O., Polak, B., Treur, J.: Compositional verification of a multi-agent system for one-to-many negotiation. Applied Intelligence 20(2), 95–117 (2004)
9. Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK (2002), http://www.cse.dmu.ac.uk/STRL/ITL/
10. Cau, A., Collette, P.: Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. Acta Inf. 33(2), 153–176 (1996)
11. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: 08031 – software engineering for self-adaptive systems: A research road map. In: Software Engineering for Self-Adaptive Systems (2008)
12. Engelfriet, J., Jonker, C.M., Treur, J.: Compositional verification of multi-agent systems in temporal multi-epistemic logic. In: Rao, A.S., Singh, M.P., Müller, J.P. (eds.) ATAL 1998. LNCS (LNAI), vol. 1555, pp. 177–193. Springer, Heidelberg (1999)
13. Fisher, M., Wooldridge, M.: On the formal specification and verification of multi-agent systems. Int. J. Cooperative Inf. Syst. 6(1), 37–66 (1997)
14. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for Organic Computing systems, pp. 233–242. IEEE Computer Society Press, Los Alamitos (2008)

15. Güdemann, M., Ortmeier, F., Reif, W.: Formal modeling and verification of systems with self-x properties. In: Yang, L.T., et al. (eds.) ATC 2006. LNCS, vol. 4158, pp. 38–47. Springer, Heidelberg (2006)
16. Harel, D.: Dynamic logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, vol. 2, pp. 496–604. Reidel, Dordrechtz (1984)
17. Heisel, M., Reif, W., Stephan, W.: A Dynamic Logic for Program Verification. In: Meyer, A.R., Taitslin, M.A. (eds.) Logic at Botik 1989. LNCS, vol. 363, pp. 134–145. Springer, Heidelberg (1989)
18. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
19. Jonker, C.M., Treur, J.: Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. In: International Journal of Cooperative Information Systems, pp. 51–92. Springer, Heidelberg (1998)
20. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
21. Manna, Z., Pnueli, A.: Temporal verification diagrams. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 726–765. Springer, Heidelberg (1994)
22. Misra, J., Mani Chandi, K.: Proofs of networks of processes. IEEE Transactions of Software Engineering (1981)
23. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
24. Müller-Schloer, C., von der Malsburg, C., Würtz, R.P.: Organic computing. Informatik Spektrum 27(4), 332–336 (2004)
25. 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)
26. Seebach, H., Nafz, F., Steghöfer, J.-P., Reif, W.: A software engineering guideline for self-organizing resource-flow systems. In: Proceedings of IEEE SASO 2010, IEEE Computer Society Press, Los Alamitos (2010)
27. Smith, G., Sanders, J.W.: Formal development of self-organising systems, pp. 90–104. Springer, Heidelberg (2009)
28. Solanki, M., Cau, A., Zedan, H.: Augmenting semantic web service descriptions with compositional specification. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proc. of 13th int. conference on World Wide Web, pp. 544–552. ACM, New York (2004)
29. Vassev, E., Paquet, J.: Assl - autonomic system specification language. In: Software Engineering Workshop, Annual IEEE/NASA Goddard, pp. 300–309 (2007)
30. Wooldridge, M., Dunne, P.E.: The computational complexity of agent verification (2001)