Model-driven synthesis of monitoring infrastructure for reliable adaptive multi-agent systems

Benedikt Eberhardinger, Jan-Philipp Steghöfer, Florian Nafz, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Eberhardinger, Benedikt, Jan-Philipp Steghöfer, Florian Nafz, and Wolfgang Reif. 2013. "Model-driven synthesis of monitoring infrastructure for reliable adaptive multi-agent systems." In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 4-7 November 2013, Pasadena, CA, USA*, edited by Allen Nikora, Sunita Chulani, Myra B. Cohen, and Carol Smidts, 21–30. Piscataway, NJ: IEEE. https://doi.org/10.1109/issre.2013.6698901.

Model-driven Synthesis of Monitoring Infrastructure for Reliable Adaptive Multi-Agent Systems

Benedikt Eberhardinger, Jan-Philipp Steghöfer, Florian Nafz, and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
{benedikt.eberhardinger, steghoefer, nafz, reif}@informatik.uni-augsburg.de

***Abstract*—Knowledge about the current state of the system serves at least two purposes: it is the basis for decisions to act and adapt to ensure reliable operation and it can be used to verify the correctness of the system at runtime. Both purposes require that current information is available at runtime that can be evaluated. Thus, the system designers have to create a complex monitoring infrastructure that suits the purposes of the system.**

We propose a combination of proven techniques that can be used as the basis for such a monitoring infrastructure. We combine it with a model-driven approach that allows a model transformation of information contained in the requirements and design documents to implementations of observers and controllers that allow adaptation at runtime based on current information as well as runtime verification. The approach can be easily integrated into an iterative-incremental software engineering process and is illustrated with two complex case studies.

Keywords*—*Autonomous agents; Adaptive Systems; Software Engineering; Software Reliability

I. Monitoring and Control in Reliable Adaptive Multi-Agent Systems

Adaptive multi-agent systems (MAS) are composed of a usually large number of autonomous agents, social software entities that interact with each other and the environment proactively. Such systems have the ability to react quickly to changes in their structure and in their environment by adapting the behaviour of the individual agents and the collective. This ability can increase the reliability of such systems as outages can be compensated, performance bottlenecks can be alleviated, and attacks can be averted. Even some autonomic computing frameworks are built on MAS (see, e.g., [1]) to harness these properties.

A fundamental requirement of reliable adaptive MAS is that the agents need to be aware of changes that affect them negatively. This demand is shared with many other systems that implement feedback loops that are evaluated at runtime, such as reactive distributed systems [2], real-time systems [3], and distributed safety-critical real-time systems [4]. In all cases, the system behaviour is observed by *monitors*, components that aggregate information to form a system state and check this state for desired properties. These approaches have close ties with *runtime verification* [5]. Traditional software verification takes place at design-time, before the system is deployed. However, adaptive systems make decisions at runtime that can have consequences that were unforeseeable at design-time. This is due to the fact that evolutionary changes of the system and of the environment that also occur at runtime make it impossible to pre-determine every possible future system state at design time.

To ensure correctness and reliability, it is therefore desirable to shift verification to runtime as well, where all information necessary is available and a system structure and configuration has already evolved. Monitoring at runtime—combined with runtime reflection—allows a reaction to violations of a specification. Quantitative verification at runtime [6] uses monitoring to construct or update a global model that can then be verified to check if the specification is violated and can trigger a reaction based on the model. These and related approaches use formal specifications, often formulated in specialised variants of temporal logic ([7], [8]), and complex global system models to specify and verify system behaviour.

However, even with this shift from design-time to runtime, the system designers have to consider the foundation of both adaption and runtime verification: the *monitoring infrastructure*. In all cases, it is necessary to form a kind of situational awareness, to determine the state of the system and to compare it with a desired state to make decisions about the correctness of the system or of possible adaptations. Information have to be gathered, aggregated, and evaluated. A verdict has to be passed to a controller that enacts changes in the agents and thus in the entire system. As the monitoring and control infrastructure is so closely integrated with the system and its components it is essential that it is regarded at design-time and a robust, flexible design is created. Changes in the design of the agents should be reflected in the design of the monitoring infrastructure.

We therefore propose a model-driven approach for the synthesis of monitoring infrastructure for reliable adaptive multi-agent systems. The contributions of the paper are a) a versatile combination of techniques for monitoring and control in the system class, based on the Observer/Controller architectural pattern, the Restore Invariant Approach and hard and soft constraints, all detailed in Sect. II; b) a model transformation that can be integrated easily into an iterative-incremental software engineering process that allows the automatic transformation of requirements captured as constraints to a monitoring infrastructure for the system class, detailed in Sect. III to Sect. VI and explained with a running example taken from the power management domain. Specialised concepts that are helpful in the system class are introduced in Sect. VIII.

This paper is a substantial extension and revision of a previously published work-in-progress report [9]. In contrast to the original short paper, this contribution is much more detailed in the description of the fundamental concepts used in the monitoring infrastructure and of the transformation process and includes a full description of the final transformation step. We also added a second case study (cf. Sect. VII) to illustrate the universal applicability of the approach.

II. Techniques for Behaviour Specification and Runtime Monitoring

A monitoring infrastructure that is able to detect deviations from the system's correct behaviour requires two fundamental elements: an unambiguous way to specify what constitutes correct behaviour and a system architecture that allows the integration of an adaptation feedback loop.

The specification of correct behaviour is based on constraints $\phi \in \Phi$ whose conjunction yields the system invariant INV. The *Restore Invariant Approach* (RIA [10], cf. Sect. II-A) uses the invariant to define a corridor of correct behaviour. To express desired behaviour (a stronger notion than correct behaviour), soft constraints can be specified as detailed in Sect. II-B. An architectural pattern that allows the observation of constraints is the Observer/Controller [11], outlined in Sect. II-C, that is the structural basis of the monitoring infrastructure proposed in this paper.

A. The Restore Invariant Approach

We employ the Restore Invariant Approach (RIA) [10] that defines a corridor of correct behaviour. The system is continuously checked against this corridor. While the system remains within the corridor, it works correctly. This property can even be verified at design time, see [12]. When it leaves the corridor, a reconfiguration takes place. More formally, the system can be regarded as a transition system $SYS = (S, \rightarrow, I, AP, L)$ with S a 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 $L : S \rightarrow 2^{AP}$ a labelling function. Within such a system, a trace π is a sequence of states $\sigma_i \in S$, related via $\rightarrow$ and starting from an initial state $\sigma_0 \in I$.

An example of a trace of an abstract transition system SYS is shown in Fig. 1. The corridor distinguishes states in which the invariant holds and those in which it does not. The corridor is constituted by an invariant INV. Thus, if INV doesn't hold any longer, the system has to react accordingly. The figure also shows that a system reaction to the violation of INV brings the system back into the corridor. To achieve this, the reaction has to be designed appropriately. The systems we consider are composed of individual autonomous agents with potentially complex interactions with other agents and the environment. To ensure that the agents participating in a system reaction do not interfere with it, they transition to a *quiescent state* in which they perform no critical actions [12].

So far, the use of RIA required the manual implementation of the monitors that checked whether or not the system is still in the corridor. With the transformation process presented in this paper, this step can now be automated and integrated into the software engineering process.

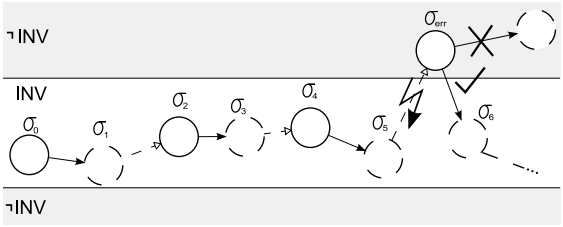


Fig. 1. The behavioural corridor of an autonomous evolving system as defined by the RIA invariant.

To be able to monitor a system appropriately it is necessary to evaluate the system invariant INV in every step $\sigma_i \in S$ of the system trace π. In the context of the RIA it is particularly interesting to know if INV does not hold in a specific step σ_i since an appropriate action has to be performed in this case. If INV holds, no action is required. Since the system invariant INV is composed of constraints $\phi \in \Phi$, their validity is fundamental for the validity of INV. The relation between INV and ϕ is as follows:

$$\forall \sigma_i \in S : \bigwedge_{\phi \in \Phi} \phi(\sigma_i) \rightarrow INV(\sigma_i) \tag{1}$$

where $\phi(\sigma_i)$ is the evaluation of the constraint ϕ in the state σ_i of a system trace π and $INV(\sigma_i)$ is the evaluation of the system invariant INV in the state σ_i of the trace π. Based on the conjunctive form of Eq. (1), the violation of INV is the consequence of a violation of one of the constraints:

$$\forall \sigma_i \in S : \exists \phi \in \Phi : \neg\phi(\sigma_i) \rightarrow \neg INV(\sigma_i) \tag{2}$$

Therefore, $\neg\phi(\sigma_i)$ implies that the invariant is violated and a reaction of the system is required. Thus, it is sufficient to monitor each constraint $\phi \in \Phi$ separately and react to any violation of a single constraint ϕ. This alleviates the need to gain a global view of the system to monitor a violation of INV in the RIA, a fact that not only makes monitoring much easier to implement but also leads to improvements in scalability and flexibility of the monitoring approach.

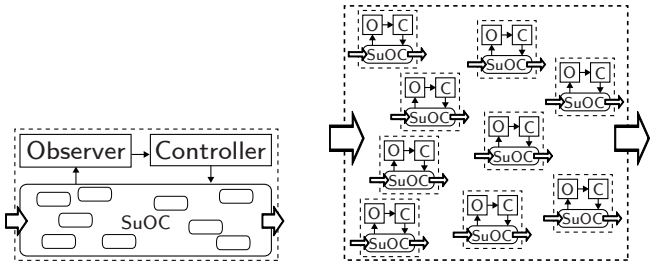
The constraints that are part of the system invariant are based on the functional and non-functional system requirements that constitute the *correct* behaviour of the system. If any of these constraints is violated, the system does not fulfill these requirements and an appropriate action has to be taken. Issues of functional correctness as well as reliability or performance can be formulated this way.

B. Hard and soft constraint

The constraints that are part of the system invariant INV must not be violated in the productive system. Therefore, these are *hard* constraints. A violation of those hard constraints leads to a reaction, where the productive state is interrupted, the system is transitioned into a quiescent state, and the system is reconfigured. For the violation of different constraints, different reactions can be appropriate.

It is often necessary, however, to describe *desired* system behaviour. If the system does not show this desired behaviour but still exhibits correct behaviour, it might not be necessary to react immediately. Therefore, requirements describing desired haviour must be expressed as constraints that are not part of the system invariant, the so-called *soft* constraints. They define a narrower "corridor within the corridor", that, e.g., expresses the optimal behaviour of the system (cf., e.g., [13]).

Soft constraints can be monitored with exactly the same monitoring infrastructure as hard constraints. Depending on the way they are formulated, they can serve different purposes. If a soft constraint (e.g., $x < 10$) has a stronger condition than a corresponding hard constraints (e.g., $x < 15$) they



(a) Central (b) Decentral

Fig. 2. Different realisations of Observer/Controller architectures [14].

can a) be considered optimization constraints but can also b) indicate that a system might approach the corridor of correct behaviour. If soft constraints are aggregated in hard constraints as in the example below, they are used to describe a desired state on two or more distinct variables with a hard restriction on their combination. If soft constraints indicate optimization criteria or a possible system development towards the corridor's boundaries, it might still be a good idea to react to their violation. In contrast to violations of hard constraints, however, such a reaction will not switch the system to a quiescent state in which the productive behaviour is suspended, but will work in the background.

Example: Brief constraint scenario. Consider an agent A with three internal numerical variables, namely x, y and z, describing its state. The value of x may be never greater than 100 or a failure will occur. Furthermore, the values of y and z should be less than 200 so that the agent works economically. Also, no system state is allowed where y and z are both greater than 200 at the same time. This textual description can be interpreted as follows:

context A **inv** c1: $x <= 100$ **context** A **soft** c2: $y < 200$
context A **soft** c3: $z < 200$ **context** A **inv** c4: c2 or c3

Thus, *c2* and *c3* are soft constraints, which do not affect the validity of the system invariant. These constraints can be monitored in order to optimize the system. Constraints *c1* and *c4* are hard constraints and must hold in every productive state and *c4* is a hard constraint over a set of two soft constraints.

C. Observer/Controller architecture

In order to implement the RIA, the system is integrated into an Observer/Controller (O/C) architecture. The monitoring is performed by an observer that is coupled with a controller, as defined in the O/C architecture, [14], a variant of the classic MAPE cycle, and an operationalisation of a supervisory feedback control loop as, e.g., used in discrete event systems [15]. We assume that monitoring and analysis take place in the observer, while the planning and execution steps of the MAPE cycle occur in the controller. The observer creates situation indicators from the data collected that is used by the controller to make decisions that will influence the system. Such decisions can include triggering a self-organization process or changes in the system parameters. The O/C thus constitutes a feedback and control loop that adapts the system to counter unwanted behaviour in an evolving system.

The concrete realization of the O/C depends mainly on the definition of the system under observation and control (SuOC). Basically the SuOC could encompass the entire system, thus forming a centralized approach as shown in Fig. 2a with one Observer/Controller. In contrast to this centralized approach a decentralized approach—as shown in Fig. 2b—is feasible, where the SuOC comprises a single system unit, e.g., a software agent. By using a decentralized O/C it is possible to gain a scalable and flexible systems. Based on Eq. (2) we are able to employ the RIA in a decentralized O/C.

III. The Transformation Process – From Requirements to Observers

The synthesis of the monitoring infrastructure follows a model-driven process, outlined in Fig. 3, and is divided into three major steps which can easily be integrated into an iterative-incremental design process. The process can be repeated when requirements or the domain model change in a model-driven design (MDD) approach. Changed parts of system models and implementation will be re-generated while existing models and code are preserved.

Step 1: System goals and constraints are described formally during the iterative process of requirements analysis, shown in Fig. 4. During this process, a domain model is created that can be used to express constraints in OCL (Object Constraint Language)—a language arguably more accessible to system designers than the aforementioned temporal logics—that formally describe the requirements (see Sect. IV). These OCL-constraints $\phi \in \Phi$ define the correct states of the system, as shown in Eq. (1), and are therefore transformed in a model-to-model transformation to abstract observers (see Sect. V) when moving from the analysis state to the design stage in each iteration. The state-based evaluation of the OCL-constraints, i.e., to check for all $\phi \in \Phi$ whether $\phi(\sigma_i)$ with $\sigma_i \in S$ holds, conforms well with the semantics of the transition system RIA is based on as described above.

Step 2: The underlying structure of the observer model, which is created during the transformation process described in Sect. V, contains the refined elements from the domain model and the generic Observer/Controller model. In addition, for each agent that has to be monitored, an observer class is created and for each OCL-constraint, a new abstract constraint

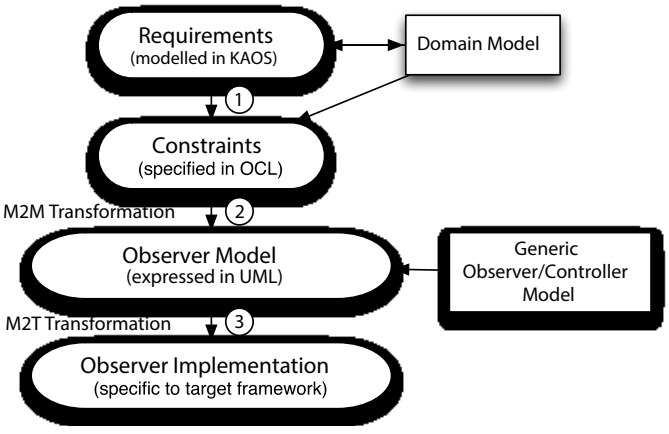


Fig. 3. The transformation process, starting from requirements modelled in KAOS (cf. Sect. IV) that are formally described while the requirements become clear and a domain model is elaborated, to abstract observers expressed as UML class and activity diagrams backed by a model of the Observer/Controller, to the final implemented observers for the target platform.

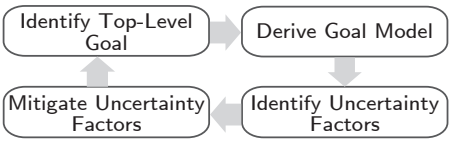


Fig. 4. The requirements engineering process, according to [18], starting with the identification of the top-level goals of the system, which are refined in the next step. The resulting goal model is afterwards examined according to find uncertainty factors. This could be mitigated, e.g., by introducing new goals. This process is iteratively executed until all uncertainties are eliminated.

class is created. Furthermore, the validity check for each constraint $\phi(\sigma_i)$ is modelled as a UML activity diagram in the constraint classes.

Step 3: The observer model is transformed into code from which the actual distributed observers for the agent system can be compiled (see Sect. VI) when moving from the design to the implementation stage. This last transformation is highly specific to the target system.

If the generic observation model proposed here is used, the software engineer has four responsibilities left: (1) formally describe the constraints based on a domain model; (2) create platform-specific transformation rules for the target platform; (3) create code for the controller and (4) for the state update. The rest of the infrastructure is created by the transformations.

IV. Step 1 – From Requirements to Constraints

To specify the requirements we use the goal-oriented requirements specification methodology KAOS, suggested by von Lamsweerde and Letier [16]. Basically the KAOS methodology consists of a graphical and a textual, formal description of the desired system. The graphical notation describes and connects goals, requirements, agents and obstacles. The connections specify refinements, assignments, or obstructions. These elements are used to model a goal refinement graph with specific system requirements at its leafs (cf. Fig. 5). To build this graph we are using only a subset of the methodology, i.e., the refinement consists only of logical *and*s, implying that all requirements have to be achieved to fulfil the global system goal. We use OCL instead of linear temporal logic (LTL) proposed by van Lamsweerde and Letier for the formal description of these requirements. The requirements are modelled in Objectiver [17] and exported in Eclipse XMI format for direct integration into the model-driven process.

Cheng et al. [18] propose an extension of the KAOS methodology that allows to express requirements for adaptive systems by incorporating uncertainty factors the system is supposed to adapt to. Their existence can lead to a reformulation of requirements, introduction of new requirements, or a change in existing ones, effectively introducing requirements for system adaptivity. The proposed process, shown in Fig. 4, is structured into four parts, which are peformed iteratively. Since it uses progressive refinements from system goals to individual requirements of the agents, it can be easily integrated in an iterative-incremental software engineering process,

Identify Top-Level Goal: The refinement process starts with a global goal for the intended system. Based on this global goal the top-level goals are derived which are necessary to fulfil the global goal.

Derive the Goal Model: These top-level goals are the basis for the complete goal model, which is developed by refining the goals until clear and achievable requirements can be formulated for the fulfillment of the goals.

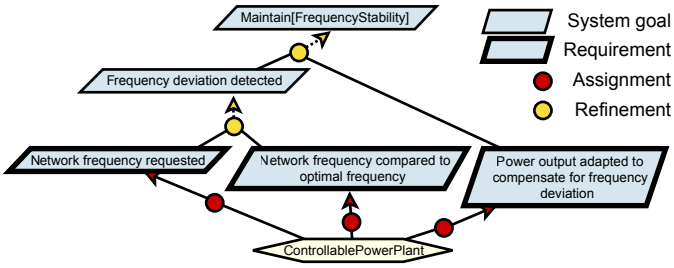


Fig. 5. The goal refinement graph in KAOS' graphical notation for the goal "Maintain[FrequencyStability]".

Identify Uncertainty Factors: This goal model has to be examined to identify uncertainty factors, i.e., obstacles that might prevent the system from reaching the goals.

Mitigate Uncertainty Factors: These obstacles can be mitigated by reformulating the goal model, introducing new goals, or changing existing goals.

After mitigating the uncertainties, the process has to be repeated, until a sufficient model is developed. As part of this refinement process, we propose to augment requirements with formal specification of system goals and constraints in OCL. These OCL-constraints will be monitored by the observers and are formulated on the concepts defined in the domain model.

Example: Constraint to observe the network frequency in power grids. In autonomous power grids, scheduling of controllable power sources is performed based on predictions of output and demand. These predictions are based on a number of uncertain factors. Therefore, even the best scheduling algorithms will never be able to approximate the required demand and the so called "residual load", i.e., the power that needs to be produced when all production by non-controllable power plants (solar, wind, residential heat-and-power) has been factored in. However, the power grid is very sensitive to deviations between production and demand and therefore, there needs to be an adaptive mechanism that can quickly react to such deviations. Since deviations alter the power grids internal frequency, all power plants can monitor this frequency and react to deviations from the optimal frequency autonomously.

The necessity of adapting the power plants' output based on the network frequency is captured in the goal "Maintain[FrequencyStability]". It can be refined to concrete requirements for controllable power plants as shown in Fig. 5. They need to measure the frequency and compare it to the optimal frequency, reacting to deviations by adapting their output. These requirements capture the control loop: changing the output has a direct effect on the network frequency, thus providing feedback. The constraint that needs to be observed corresponds to the requirement "Network frequency compared to optimal frequency". In OCL it can be expressed as:

context ControllablePowerPlant **inv** noFrequencyDeviations:
(currFrequency - optimalFrequency).abs() $<$ allowedDeviation

While this constraint may seem simplistic, it is a good example for a property that has to be monitored as part of a feedback loop in an adaptive system.

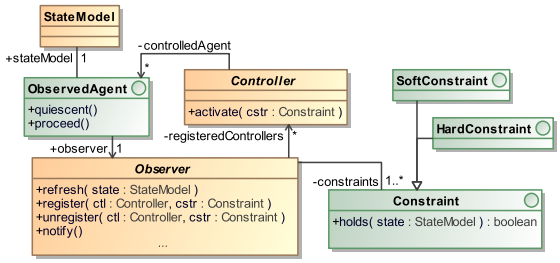


Fig. 6. Simplified generic Observer/Controller model as used in the transformation, specified as a UML class diagram.

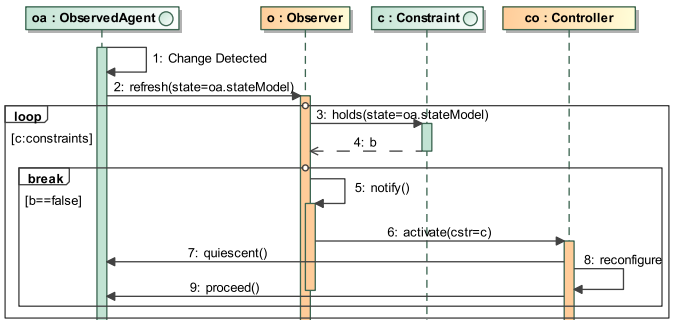


Fig. 7. A simplified sequence diagram showing the interactions between the elements of the generic Observer/Controller model.

V. Step 2 – From Constraints to Observer Models

After the relevant requirements have been formally defined with OCL-constraints, the resulting requirements model and the domain model are transformed into an observer model, which is the basis of a specific observer implementation. The transformation is defined in QVT [19], which uses *Queries* on the source models to *Transform* them into target models, the *Views*. Part of the views can be pre-specified. We use this feature to specify a generic Observer/Controller model, as depicted in Fig. 6, that provides the structure for the observers models and is based on the publish/subscribe pattern [20].

The relevant interaction between the classes is depicted in Fig. 7. Whenever an `ObservedAgent` registers a change (basically, a transition in SYS from σ_i to σ_{i+1}), it informs its `Observer` by sending the state model with the updated information. The `Observer` then updates its state model for σ_{i+1} of the agent and evaluates all `Constraints` $\phi \in \Phi$. If one of them evaluates to false, i.e., $\neg\phi(\sigma_{i+1})$, the `Observer` informs all `Controllers` which are registered for this constraint. Each `Controller` decides whether or not to enact changes in the system. In order to do this, it interrupts the productive state of the `ObservedAgent` by calling the `quiescent()` method. After any changes have been performed, the `Controller` sends a signal to the `ObservedAgent` to return to the productive state, by calling the `proceed()` method. Depending on the reaction to the constraint violation, the `Controller` might reconfigure several agents or ask other controllers to participate in the reaction. In such a case, the global quiescent state is left when all `Controllers` have told the `ObservedAgents` to leave the quiescent state and to continue with their productive work.

The requirements model and the domain model are transformed into a new observer model, which is described by a UML class diagram and activity diagrams. These diagrams are supplemented by the generic sequence diagram in Fig. 7 which characterises the interaction in the class diagram. The activity diagrams embed the OCL-constraints in the `holds()` methods of the implementations of the `Constraint` interfaces.

Identify Observed Object: First, to generate the observer model it is necessary to identify the classes which should be monitored. These are all classes from the domain model which have an according agent in the requirements model that has a set of semi-formally defined requirements to be monitored. Each class identified in this manner has to implement the `ObservedAgent` interface in the observer model.

Generate Monitoring Structure: Furthermore, a specific state model for each of these classes is created by generating a simple class containing all attributes of the agent class which represents the state of the observed class. Additionally an observer derived from the `Observer` class for this agent is created. Thus, a relationship between the object that should be observed and the concrete observer is generated. Another relationship is generated between the specific observer object and a generated concrete controller class derived from `Controller`. To complete the control-loop a connection between the generated controller class and the agent class is established. Most importantly, the specific constraint class is added into the model. For this purpose, the type of constraint is identified, i.e., if it is a hard or a soft constraint by evaluating the keywords **inv** and **soft** in the OCL statement which indicates the type of the constraint. Once the constraint type is determined the specific class is generated, which implements either `SoftConstraint` or `HardConstraint`.

Generate Dynamic Model: The final step in the creation of the observer model is to parse the OCL statement and put it into the guards of the activity diagram that describes the functionality of the `holds()` method of this constraint class. The generic activity diagram for the constraint type is used by copying it and replacing the wildcard with the specific extracted guard.

This procedure is repeated for every constrained agent. After the completed transformation there is one observer per agent, but there are several constraints per observer. Furthermore, a controller is able to register for a specific constraint by a specific observer. This enables using specialized controllers for optimization and for maintaining correct behaviour according to INV. Fig. 8 depicts the elements that are used in this transformation process as well as a simplified version of the resulting class diagram. The transformation creates class diagrams for all agents, as well as the diagrams that model the respective methods for the new constraint classes and the observer. It also checks the domain model and the requirements model for inconsistencies, e.g., if the requirements model defines agents for which no class exists in the domain model. The result of the transformation process is a platform independent model of the monitoring infrastructure specified in the UML2 meta-model of the Eclipse Modeling Framework (EMF).

The `Controller` class is only a stub at this point, with no functionality. Its specification can be much more complex than the observers' and is usually a part of the design documents. However, the template provided by the transformation detailed here can be used as a starting point for the modelling and the implementation of the controller.

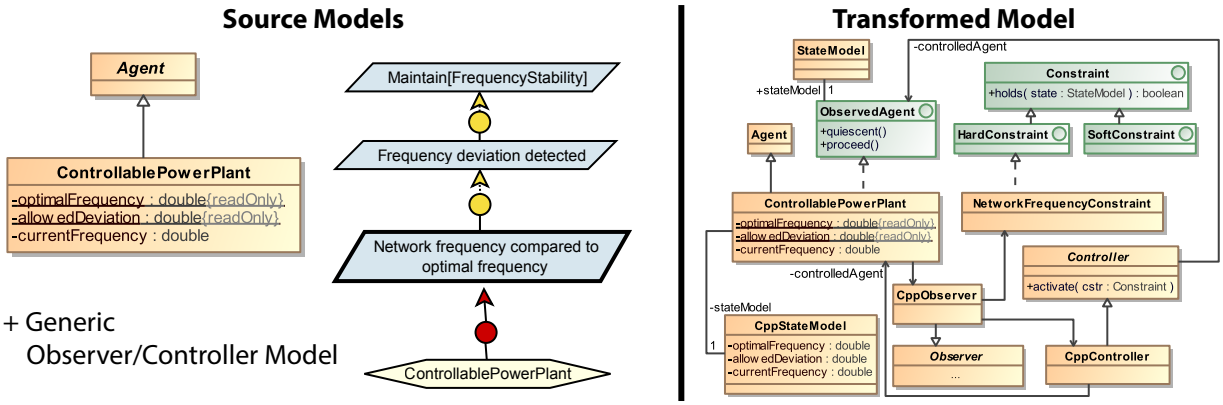


Fig. 8. The sources of the transformation (simplified representation) and the resulting simplified class diagram for the example given in Sect. IV.

VI. Step 3 – From Observer Models to Observer Implementations

The final transformation to actual observer implementations contains many platform specific choices, e.g., whether or not observers and controllers are independent agents or become part of the agents defined in the domain model, whether properties of the agents can be accessed directly or only via message passing, etc. It will therefore have to be adapted to each target platform and target system. An example of a rather non-standard transformation target is detailed in Sect. VII. However, some of the basic principles remain the same, regardless of the transformation target.

We developed a template which can be adapted for the use in a specific target system. This template is defined in the language Xpand, which is a part of the Eclipse Modelling Framework (EMF) and enables model-to-text transformations. The template contains static parts, written in the target programming language that do not change and only have to be copied into the source code files. In addition, it contains dynamic parts that depend on the elements in the observer model and are evaluated when the transformation is performed. This final transformation consists of two steps:

Structural Transformation: First, stubs of the classes contained in the class diagram of the observer model are generated. As defined in the static part of the transformation template, these stubs are integrated into the target platform, i.e., the multi-agent platform or middleware the system will run on and can contain additional initialisation or housekeeping code. The implementation of the generic sequence diagram depicted in Fig. 7 is also part of the static part and thus has to be included in the template. This step transforms the entire class diagram into system specific source code.

Functional Transformation: Afterwards, the activity diagrams are transformed into implementations of the `holds()` methods in the abstract specialisations of `Constraint`. For this purpose, the OCL statements in the transition-guards have to be translated into conditional statements expressed in the language of the target system. The dynamic part of the template parses the OCL statement into an abstract syntax tree (AST), including the constrained attributes, the OCL functions, logic operators, etc. The AST can be transformed into code for the specific target system. We use the Eclipse OCL grammar as a basis and employ a custom ANTLR parser that supports all standard OCL constructs. In comparison to using a standard parser such as the one provided with Eclipse OCL or Dresden OCL directly, this gives us more flexibility with regard to the target language and thus the transformation target.

In concrete terms for the observer model of the autonomous power grid, shown in Fig. 8, the output of this last step is as follows: The target system is the multi-agent framework Repast Simphony, which is implemented in the programming language Java. So to generate the outline of the class diagram for each interface and class of the observer model, a file containing the basic class declarations, including attributes, method stubs, etc. has to be generated. Next, the `holds()` method of the class `NetworkFrequencyConstraint` is translated into working Java code. For this purpose, the AST created by the OCL parser is translated into Java syntax, e.g., an OCL "`and`" into a Java "`&&`" and OCL methods such as "`includesAll`" into corresponding Java constructs.

Instead of parsing the constraints, it would also be possible to use the faculties of tools like Dresden OCL [21] to check the constraints at runtime. However, a parser like the one used here can be used to create code for target systems other than those based on a Java Virtual Machine which Dresden OCL is limited to as it uses Bytecode weaving. In principle, tools like Dresden OCL do not provide data gathering facilities or infrastructure for interactions with controllers. They can thus be used for constraint checking within the monitoring infrastructure but not as a replacement for the concepts proposed here.

A concrete observer implementation will have to be coupled with a controller that adapts the system accordingly. The transformation creates stubs for the appropriate classes and methods such as `activate()`. For the example given, there are a number of decentralised approaches that change the output of power plants to stabilise the network frequency [22]. An interesting issue is the creation of appropriate bootstrapping code to start observers and controllers along with the respective agents. During the bootstrapping phase, instances have to be created and associations set accordingly. After this initialisation, the controllers will have to register with the observers. Therefore, a phased system boot is usually advised.

VII. Case Study: Adaptive Production Cell

In *self-organising resource-flow systems* [23], resources are processed according to a task by independent agents. Each agent has a number of capabilities it can apply to the resource.

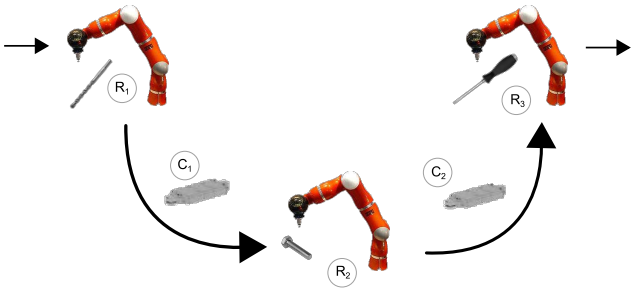


Fig. 9. A simple adaptive production cell with three robots and two carts.

Agents can exchange resources with other agents as, e.g., given by the layout of a shop floor. To fulfil the task for a resource (i.e., to apply the correct capabilities in the correct order), a resource-flow is established that determines how the resource is moved through the system and processed on the way.

An example of such a self-organising resource-flow system is an adaptive production cell. Resources are workpieces that are transported by carts between robots that have several processing capabilities. The cell depicted in Fig. 9 consists of three robots and two autonomous transport units (carts) connecting them. Each 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 has to reconfigure the cell and re-assign 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.

A. From Requirements to Constraints

The main system goal of the adaptive production cell is to process the workpieces according to their task. During requirements analysis this is refined to the system goal visible on the right-hand side of Fig. 10: the resource has to be processed according to a role allocation that defines which agent performs which capability and how the carts transport the resources. However, an obstacle has been identified as well: the role allocation can become invalid due to environmental changes such as a capability that is no longer available. This circumstance is captured in a hierarchy of obstacles, shown as red rhombuses.

To counter this uncertainty, a new goal is introduced to change a role allocation if necessary. Refinement of this goal leads to a requirement for the agent: the capability it has to apply must be available to it. Formally, this can be captured in OCL as:

```
context Agent inv capabilityConsistency:
  self.availableCapabilities
    → includesAll(self.allocatedRoles.capabilitiesToApply)
```

The basis for this formulation is the domain model of a self-organising resource-flow system depicted in Fig. 11 [23]. It is not only applicable to adaptive production cells, but to a variety of systems with characteristics as described above. Therefore, the requirements sketched here can also be more universally applied in resource-flow systems. For the sake of argument, however, we will focus on the adaptive production cell as described above in the following.

The *Capability-Consistency* constraint is by far not the only one that needs to be observed. The requirement "Agent communicates the loss of a capability it is configured to apply" could be further refined to yield a requirement that observes neighbouring agents by sending them heartbeat messages. If an agent does not answer anymore, the *I/O-Consistency* constraint is violated that states that agents resources are exchanged with have to be reachable. This constraint as well is part of the invariant and should be observed accordingly.

B. From Constraints to Observer Models

Based on the constraints specified in the requirements analysis and the generic Observer/Controller model introduced earlier and depicted in Fig. 6, the transformation yields the class diagram shown in Fig. 12. The transformation is similar to the one described in Sect. V, and one observer as well as one controller per agent type is created.

C. From Observer Models to Observer Implementations

The case study's target platform is the Organic Runtime Environment (ORE, [24]), based on Jadex [25], a multi-agent system using the Belief-Desire-Intention (BDI) agent architecture. The procedural reasoning approach behind BDI allows agents to dynamically adapt new desires (long-term *goals*) due to changes in the environment or in their *beliefs* (their internal data) and select appropriate intentions (short-term *plans*) to pursue these desires. Jadex allows the specification of these goals in a declarative form in an agent definition file, whereas plans are sequences of actions specified as Java classes.

Furthermore, Jadex allows goals to be adopted based on the fulfillment of a condition that is continuously evaluated by the agent. The condition can be a relatively simple boolean formula as well as a more complex construct, expressed in a derivative of standard Java expressions. As soon as the condition holds, the goal is adopted. The transformation process creates a number of code fragments:

Observer Goal: Each observer class from the transformed model is converted into a goal definition in the respective agent definition file. The constraints this observer has to monitor are converted into expressions that trigger the adoption of the goal if they are violated and cause the goal to be dropped as soon as they hold again.

Publish/Subscribe Infrastructure: Each agent that has an observer goal is able to receive messages from controllers that register to be informed about violations. If a controller sends a subscribe-message, it is added to the belief that stores all registered controllers. In addition, a plan that is executed

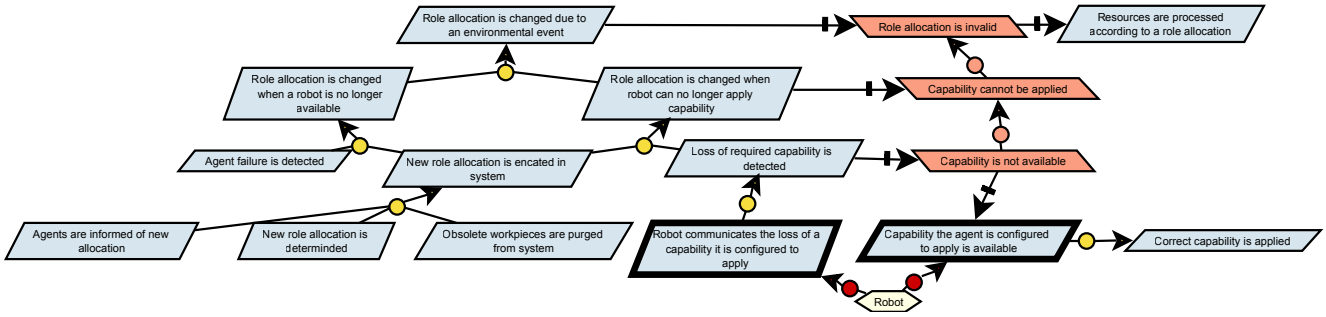


Fig. 10. Excerpt from the requirement model of the Adaptive Production Cell that deals with changing role allocations.

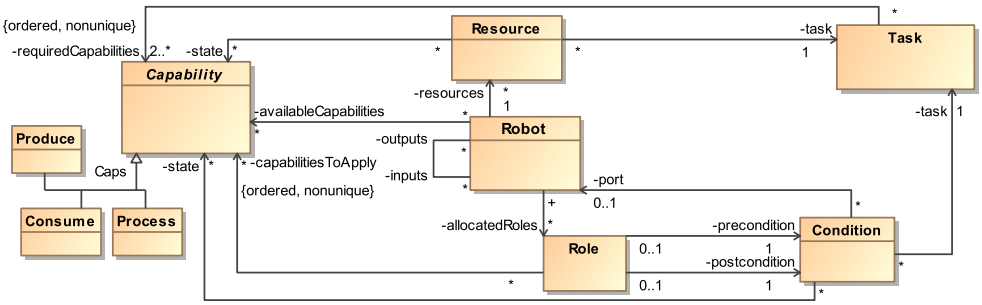


Fig. 11. Domain model of the Adaptive Production Cell. `Robot` is of type `Agent` and works on `Resources` by applying `Capabilities`.

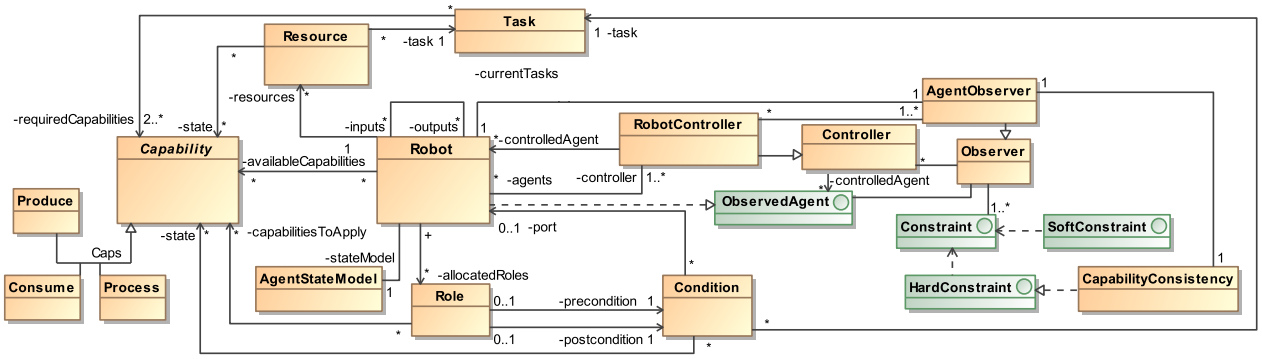


Fig. 12. Transformed model of the Adaptive Production Cell, combining the generic Observer/Controller model (Fig. 6) and the domain model.

when the observer goal is adopted is created that uses this information to inform the appropriate controllers. The message sent to the controllers also includes a model of the current agent state, derived from the agent's beliefs.

Controller Stubs: The controllers are themselves agents that become active as soon as a message about a constraint violation is received. The transformation creates the appropriate agent definition files and the infrastructure to handle messages. The plans that reconfigure the system are however left as stubs so the actual reconfiguration algorithm can be plugged in.

At runtime, the agent will continuously evaluate the goal condition based on the current state of its beliefs. If the goal condition is violated, the appropriate plan is executed which will inform all registered controllers. The controllers can then run the reconfiguration algorithm that restores the invariant and thus the goal's condition. As several goals can be active at the same time, it is possible to run reconfigurations for different violated constraints in parallel.

VIII. Specialised Concepts for Open, Heterogeneous Systems

So far, we have only considered systems in which the designer controls the architecture and implementation of the agents and has full control over their behaviour. Thus, agents always behave benevolently and disclose the required information. It can also be assumed that the information is correct to the best of the agents knowledge. If we consider open, heterogeneous systems however, in which agents can enter and leave the system arbitrarily, are not under control by the designer and can behave arbitrarily, matters of trustworthiness come into play. Monitoring in such an environment is a much more challenging task. Agents might not disclose truthful information, either intentionally to exploit the system or unintentionally due to errors or other circumstances [26]. Thus, information provided by these agents about their internal state has to be taken with a grain of salt.

On the other hand, these agents might not reveal any information at all. If that is the case, the internal state must

be approximated by the externally observable behaviour. This becomes especially difficult as such information can usually only be determined from the interactions with others and the environment. Instead of monitoring single agents, groups of agents now have to be observed. Such a situation can also occur in the adaptive MAS we considered so far: depending on the constraint, different views of the system might have to be considered. A regional view of the system could be helpful where an observer can monitor several agents and obtain a picture of a compartment of the system. This is due to the fact that, e.g., emergent behaviour only arises through interaction of different agents. To constrain certain behaviour, the collective behaviour of groups of agents has to be monitored. In distributed open systems, such information is, however, not readily available locally. Aggregation of the required data is therefore one open issue [2].

These challenges constitute current limitations of our approach. The former challenge can be dealt with by the introduction of *black box/white box monitoring*. To deal with the latter challenge, we plan to exploit *hierarchies*.

A. Black-Box/White-Box Monitoring Concept

The proposed monitoring techniques assume that all required data is provided to the monitor. In certain situations, an agent will provide this information about its internal state voluntarily. The assumption that an agent would always do this does not hold in open heterogeneous multi-agent systems. That is the case if the monitor is integrated into the internal feedback-loop of the agents which enables adaptivity and thus reliability. Thus, the monitor and the observed object are coupled very closely and therefore the agent can be considered a *white box*. In such a situation the information is highly reliable. In general, it is desirable to only monitor externally observable properties, but this limits the applicability of any monitoring approach severely. In case of a selfish agent, who is trying to maximize its private benefit, its information is not reliable. In addition, it might have no interest adhering to constraints that are only useful for other agents of the system but not for him. Consequently, to avoid an external interference by a controller it could tamper the information about its state. Therefore, the state of an agent has to be derived by observation of the behaviour to gain reliable information and must be considered a *black box*.

Thus, there are agents which could be considered a white-box and agents which have to be treated as a black-box. The realization of white-box monitoring is straightforward, because all needed information is trustworthily provided by the agent. All constraints could consequently be evaluated on this provided information. The evaluation of constraints in the black-box monitoring concept is much more complex. As the information about the current state of the observed agent is not directly available, it is necessary to model its behaviour. Based on this model the monitor is able to state whether or not the current state fulfils its constraints. The required model depends on the specific agent and its domain. Techniques to approximate stochastic processes such as the one in [26] allow the formulation of such complex models from the observed behaviour. Appropriate infrastructure to populate them would have to be created by the transformation rules.

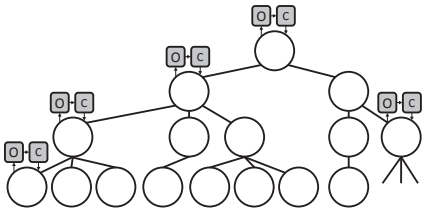


Fig. 13. Hierarchical system structure with multiple Observer/Controllers on different levels of the system.

B. Hierarchical Monitoring

To deal with the latter limitation, we plan to exploit hierarchies in our future work. As many systems are structured hierarchically, we want to use the hierarchical structure and observe constraints within individual parts of the hierarchy. Additionally, we are looking into using soft constraints to optimise the system on the fly. Not all constraint violations make it necessary to reconfigure the system, some can also trigger a self-optimisation process that does not require the agent to go into a quiescent state. Such a staged reaction can also be used to, e.g., escalate reconfiguration attempts if less aggressive methods do not yield the desired result.

Hierarchies ensure the scalability of the system by providing information about the different parts of the system at different aggregation levels [27]. Thus, state models can be aggregated on a higher level and monitoring can use this aggregated information to observe the system behaviour. The integration of the monitor synthesis into a hierarchical system is feasible, because of its one-to-one relationship between an agent and an observer. Fig. 13 shows the integration of the Observer/Controller concept into a hierarchical system. The principle is that a node, i.e., an agent, aggregates all information about its children. Therefore, behaviours within a group of agents could be monitored by collecting information from one agent, which is the parent node in the hierarchy. The allocation between an observer and a hierarchical level could be obtained from the requirement document and the corresponding domain model.

IX. Discussion and Outlook

In this paper, we presented an approach to transform semi-formally specified requirements into observers that are able to monitor adaptive multi-agent systems at runtime. We believe that this approach will provide system engineers with an accessible way to integrate online monitoring into such systems and benefit from the possibilities an extensive, easily to maintain monitoring infrastructure offers with regard to reliability and adaptability. Integration into the software engineering process is a great advantage, as is the universality of the process. Only the last transformation step is system-specific, at least as long as the generic Observer/Controller model can be used. The most important benefit, however, is that constraints can be easily expressed in OCL, a language that is arguably much better understood by system engineers than temporal logics or other formal specification paradigms. Temporal logics can, however, express statements about past and future events. While some authors resort to past-time temporal logic (e.g., [28]) to allow the former and prevent the latter, our future work will include handling such statements in OCL as well.

There are some limitation to the current status of the approach that have to be noted. The incorporation of hierarchies and the development of black-box monitoring was already discussed in Sect. VIII. As mentioned above, the consideration of *timed constraints* is an important topic for future work. While it is relatively easy to express constraints such as "$x < 10$ in 5 of the last 8 time steps", the systems we regard usually do not have a synchronised clock. It is thus necessary to express such constraints with actual time periods. However, such constraints can not be monitored in a straight-forward fashion due to issues with the minimal sampling rate required for such checks and the inherent delay within the monitoring infrastructure. They also require a history of states that can be evaluated.

As an interesting side effect of the way we propose to specify constraints in the requirements model, it is relatively straightforward to generate rely/guarantees from the requirements models for use as a formal system specification in verification [12]. This enables the co-design of models for verification and design models and opens up possibilities to keep formal models up-to-date during the entire development effort. It is therefore possible, to continually verify the correctness of the current design and check the requirements specification for inconsistencies. This early error detection in turn increases reliability in the final system.

ACKNOWLEDGEMENT

This research is partly sponsored by the German Research Foundation (DFG) in the project "OC-Trust" (FOR 1085).

REFERENCES

[1] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *Proc. 3rd Int. Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, ser. AAMAS '04. IEEE Computer Society, 2004, pp. 464–471.

[2] Y. Bai, J. Brandt, and K. Schneider, "Monitoring distributed reactive systems," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, nov. 2012, pp. 84 –91.

[3] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Real-Time Systems, 1999. Proc. 11th Euromicro Conference on*, 1999, pp. 114 –122.

[4] A. Goodloe and L. Pike, "Monitoring Distributed Real-Time Systems: A Survey and Future Directions," NASA Langley Research Center, Tech. Rep. NASA/CR-2010-216724, July 2010.

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[6] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[7] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[8] ——, *Temporal verification of reactive systems: safety*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[9] J.-P. Steghöfer, B. Eberhardinger, F. Nafz, and W. Reif, "Synthesis of Observers for Autonomic Evolutionary Systems from Requirements Models," in *Proc. 6th Int. Workshop on Distributed Autonomous Network Management Systems*. IEEE Computer Society, 2013.

[10] F. Nafz, H. Seebach, J.-P. Steghöfer, G. Anders, and W. Reif, "Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach," in *Organic Computing — A Paradigm Shift for Complex Systems*, ser. Autonomic Systems, C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Springer, 2011, pp. 79–93.

[11] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a generic observer/controller architecture for Organic Computing," in *Informatik 2006 - Informatik für Menschen, Band 1, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, ser. LNI, vol. 93. GI, 2006, pp. 112–119.

[12] F. Nafz, J.-P. Steghöfer, H. Seebach, and W. Reif, "Formal Modeling and Verification of Self-* Systems Based on Observer/Controller-Architectures," in *Assurances for Self-Adaptive Systems*, ser. LNCS. Springer, 2013, vol. 7740.

[13] S. Quinton and R. Ernst, "Generalized weakly-hard constraints," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, ser. LNCS, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2012, vol. 7610, pp. 96–110.

[14] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, "Adaptivity and self-organization in organic computing systems," *ACM Trans. Auton. Adapt. Syst*, vol. 5, no. 3, pp. 10:1–10:32, 2010.

[15] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer US, 2007, 2nd Edition.

[16] A. Lamsweerde and E. Letier, "From object orientation to goal orientation: A paradigm shift for requirements engineering," in *Radical Innovations of Software and Systems Engineering in the Future*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 2941, pp. 325–340.

[17] RespectIT, "Objectiver homepage," May 2013. [Online]. Available: http://www.objectiver.com/index.php?id=4

[18] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Model Driven Engineering Languages and Systems*, ser. LNCS, A. Schürr and B. Selic, Eds. Springer, 2009, vol. 5795, pp. 468–483.

[19] Object Management Group (OMG), "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," 2005. [Online]. Available: http://www.omg.org/spec/QVT/1.0/PDF

[20] E. Gamma, D. Riehle, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

[21] B. Demuth and C. Wilke, "Model and Object Verification by Using Dresden OCL," in *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice*, 2009, pp. 81–90.

[22] G. Anders, C. Hinrichs, F. Siefert, P. Behrmann, W. Reif, and M. Sonnenschein, "On the influence of inter-agent variation on multi-agent algorithms solving a dynamic task allocation problem under uncertainty," in *Proc. 6h IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) 2012*. IEEE Computer Society, 2012, pp. 29–38.

[23] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif, "How to design and implement self-organising resource-flow systems," in *Organic Computing – A Paradigm Shift for Complex Systems*, ser. Autonomic Systems, C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Springer, 2011, vol. 1, pp. 145–161.

[24] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif, "A generic software framework for role-based Organic Computing systems," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*. IEEE, 2009, pp. 96–105.

[25] L. Braubach, A. Pokahr, and W. Lamersdorf, "Jadex: A BDI Agent System Combining Middleware and Reasoning," in *Software Agent-Based Applications, Platforms and Development Kits*, M. K. R. Unland, M. Calisti, Ed. Birkhäuser-Verlag, 9 2005, pp. 143–168.

[26] G. Anders, F. Siefert, J.-P. Steghöfer, and W. Reif, "Trust-Based Scenarios – Predicting Future Agent Behavior in Open Self-Organizing Systems," in *Proc. of the 7th International Workshop on Self-Organizing Systems (IWSOS 2013)*, May 2013.

[27] J.-P. Steghöfer, P. Behrmann, G. Anders, F. Siefert, and W. Reif, "HiSPADA: Self-Organising Hierarchies for Large-Scale Multi-Agent Systems," in *Proc. 9th Int. Conference on Autonomic and Autonomous Systems (ICAS)*. IARIA, 2013.

[28] G. Roşu, F. Chen, and T. Ball, "Synthesizing monitors for safety properties: This time with calls and returns," in *Runtime Verification*, ser. LNCS, M. Leucker, Ed. Springer, 2008, vol. 5289, pp. 51–68.