Synthesis of Observers for Autonomic Evolutionary Systems from Requirements Models

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

***Abstract*—Monitoring the behaviour of autonomous evolutionary systems is a necessity to control their behaviour at runtime and react to undesired changes or developments. We propose an approach that derives observers from system requirements annotated with OCL-constraints in a model-driven way that can easily be integrated into iterative incremental design processes.**

***Keywords*—Autonomous agents; Adaptive Systems; Software Engineering; Fault Detection**

I. Monitoring in Autonomic Evolving Systems

Autonomic 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. Therefore, it is even more important to make credible assertions about the system behaviour at design-time than for more conventional, non-autonomous systems.

Alas, traditional verification techniques such as theorem proving and model checking are hardly applicable for large-scale autonomous evolving systems due to their complexity, the associated state-space explosion, and the need for a global system view that is inherent in these techniques. Therefore, some approaches exist that shift verification to runtime as well, where all information necessary is available and a system structure and configuration has already evolved. Most prominently, runtime verification [1] is concerned with techniques to monitor system properties. In combination with runtime reflection, a reaction to violations of a specification becomes possible. Quantitative verification at runtime [2] 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 temporal logic, and complex global system models to specify and verify system behaviour.

This paper presents a work in progress that tries to integrate the creation of suitable observers into the design process of the system by defining a transformation from constraints that capture system requirements to these observers. The constraints are formulated in OCL (Object Constraint Language)—a language arguably more accessible to system designers than the aforementioned temporal logics—and defined on a domain model. The transformation can easily be integrated in a design process and allows a model-driven development of monitoring capabilities. Sect. II first describes the Restore Invariant Approach, the formal basis for the framework proposed here. The basic transformation process is introduced in Sect. III, while Sect. IV to Sect. VI give a more detailed view of the transformation steps. The paper concludes with a discussion of the approach and an outlook on future work.

II. The Restore Invariant Approach

We employ the Restore Invariant Approach (RIA) [3] 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 as verified at design time. When it leaves the corridor, a reconfiguration takes place. The corridor is constituted by an invariant INV that is basically the conjunction of all constraints $\phi \in \Phi$. Thus, if one of the constraints is violated, INV doesn't hold any longer and the system has to react accordingly. 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 exemplary 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 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 [4].

Since the invariant is the conjunction of all system constraints, the violation of one constraint implies the violation of the invariant. It is therefore necessary to monitor the individual constraints and react to their individual violation. The monitoring is performed by an observer that is coupled with a controller, as defined in the Observer/Controller architecture, [5], a variant of the classic MAPE cycle, and an operationalisation of a supervisory feedback control loop as, e.g., used in discrete event systems [6]. We assume that both monitoring and analysis take place in the observer, while the

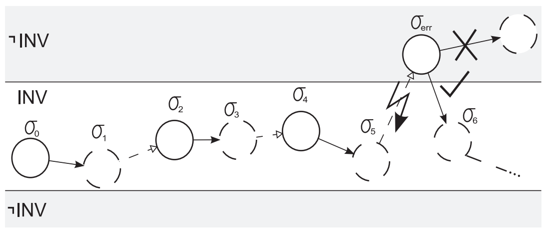

Fig. 1. The behavioural corridor of an autonomous evolving system as defined by the RIA invariant.

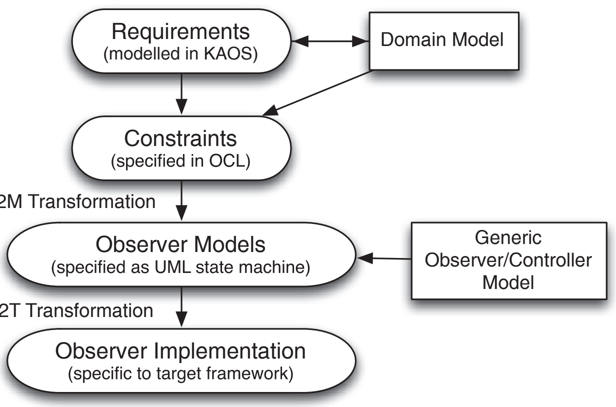

Fig. 2. The transformation process, starting from requirements modelled in KAOS that are successively formally described as the requirements become clearer and a domain model is elaborated, to the abstract observers expressed as UML state machines backed by a UML model of the observer/controller, to the final implemented observers for the target platform.

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 future course of the system. Such decisions can include triggering a self-organisation process or changes in the system parameters. The observer/controller thus constitutes a feedback and control loop that can adapt the system to counter unwanted behaviour in an evolving system.

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

III. The Transformation Process – From Requirements to Observers

Fig. 2 gives an overview of the transformation process. The first step, the formal description of the system goals and constraints happens during the iterative process of requirements analysis. During this process, a domain model is created that can be used to express OCL-constraints (see Sect. IV) that formally describe some of the requirements. These OCL-constraints define the correct states of the system and are transformed in a model-to-model transformation to abstract observers (see Sect. V) when moving from the analysis phase to the design phase in each iteration. The state-based evaluation of the OCL-constraints conforms well with the semantics of the transition system RIA is based on as described above. The behaviour of the abstract observers is modelled with UML sequence diagrams. These diagrams describe the interactions of an observer/controller model, described as a UML class diagram. For each OCL-constraint, a new abstract constraint class is created. Finally, the abstract observers and constraints are 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 phase. This last transformation is highly specific to the target system.

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.

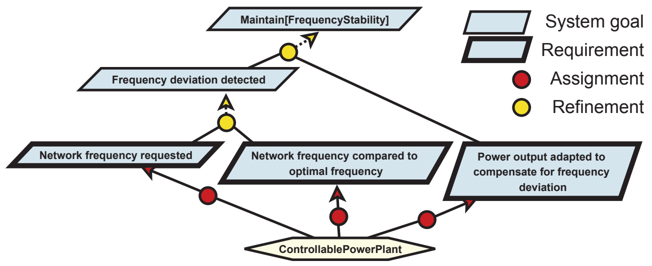
Maintain[FrequencyStability]
Frequency deviation detected
Network frequency requested
Network frequency compared to optimal frequency
Power output adapted to compensate for frequency deviation
ControllablePowerPlant
System goal
Requirement
Assignment
Refinement

Fig. 3. The KAOS model for the goal “Maintain[FrequencyStability]”

IV. Step 1 – From Requirements to Constraints

Cheng et al. propose an extension of the goal-oriented requirements specification methodology KAOS [7] that allows the expression of requirements for adaptive systems by incorporating uncertainty factors the system is supposed to adapt to [8]. These uncertainties are identified with the help of a conceptual domain model. Their existence can lead to a reformulation of requirements, introduction of new requirements, or a change in existing ones. The extended KAOS approach can be integrated easily in an iterative incremental software engineering process since it is based on progressive refinements from system goals to individual requirements of the agents. As part of this refinement process, we propose to augment requirements with semi-formal specification of system goals and constraints in OCL. These OCL-constraints will be monitored by the observers. The constraints 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 on the controllable power plant as shown in Fig. 3. The power plant needs to measure the frequency and compare it to the optimal frequency. It reacts to deviations by adapting its output. The last two requirements capture the control loop: the change of the output has a direct effect on the network frequency, thus providing feedback. The constraint that needs to be observed corresponds to the formal description of the requirement “Network frequency compared to optimal frequency”. In OCL it can be expressed as:

```
context ControllablePowerPlant inv noFrequencyDeviations:
 abs(currFrequency - optimalFrequency) < 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 autonomous system.

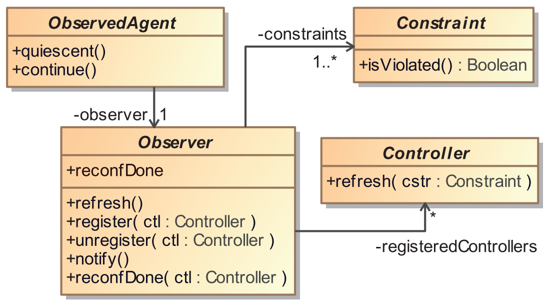

Fig. 4. Simplified generic Observer/Controller model as used in the transformation, specified as a UML class diagram.

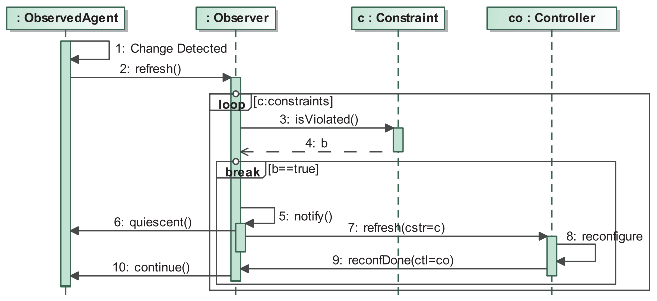

Fig. 5. A simplified sequence diagram showing the interactions between the elements of the generic Observer/Controller model.

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 observer models that are the basis of concrete observer implementations. The transformation is defined in QVT [9] 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. 4, that provides the structure for the observers models and is based on the publish/subscribe pattern.

The relevant interaction between the classes is depicted in Fig. 5. Whenever an ObservedAgent registers a change (basically, a transition in SYS), it informs its Observer. The Observer then updates its internal model of the agent and evaluates all Constraints. If one of them evaluates to false, the Observer sends the ObservedAgent the order to go into the quiescent state. It then informs all Controllers of the constraint violation. Each Controller can then decide whether or not to enact changes in the system. After any changes have been performed, the Observer is notified that the reconfiguration is done. When all Controllers reported back, the Observer tells the ObservedAgent to leave the quiescent state and continue with its productive work. Please note that at this point it is not specified how the information about the state change is communicated to the Observer. This decision is delegated to a later point (see Sect. VI).

Each agent from the domain model is transformed into a new agent class that is derived from ObservedAgent and from Agent, the latter class specifying general capabilities such as communication. It is important to note that each agent from the requirements needs to have a corresponding class in the domain model with the same name. For each OCL-constraint from the requirements, a new class is created that is derived from the abstract Constraint class. If temporal constraints are to be observed, e.g., to limit the number of violations within a sliding time window, specialisations of Constraint that record a history can be used. The OCL-constraint itself is used as the guard of the transitions of the UML activity diagram that describes the isViolated() method of the new constraint class. There is one observer per agent, but there are several constraints per observer.

Fig. 6 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 packaged with 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.

VI. Step 3 – From Observer Models to Observer Implementations

The final transformation to actual concrete 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. However, some of the basic principles remain the same, regardless of the transformation target.

The classes and sequence diagrams have to be translated into the target programming language and the target platform, i.e., the multi-agent platform or middleware the system will run on. Sequence diagrams become implementations of the methods of the classes. The OCL-constraints are parsed to conditional statements within Constraint.isViolated() for the corresponding constraint classes. Note that the multiple inheritance in the class diagram can be implemented differently, depending on the target language. In Java, an interface can be used, especially if the methods quiescent() and continue() are abstract methods anyways.

An important decision has to be made with regard to the flow of information at this point. Changes in the ObservedAgents cause updates to an internal model of the Observer on which the constraints will be evaluated. Alternatively, the observer or the implementations of the Constraints can request the required information from the ObservedAgents directly. Which choice is best depends on many properties of the system, including whether message-based communication is used, how often the information changes and how complex an internal model would be.

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

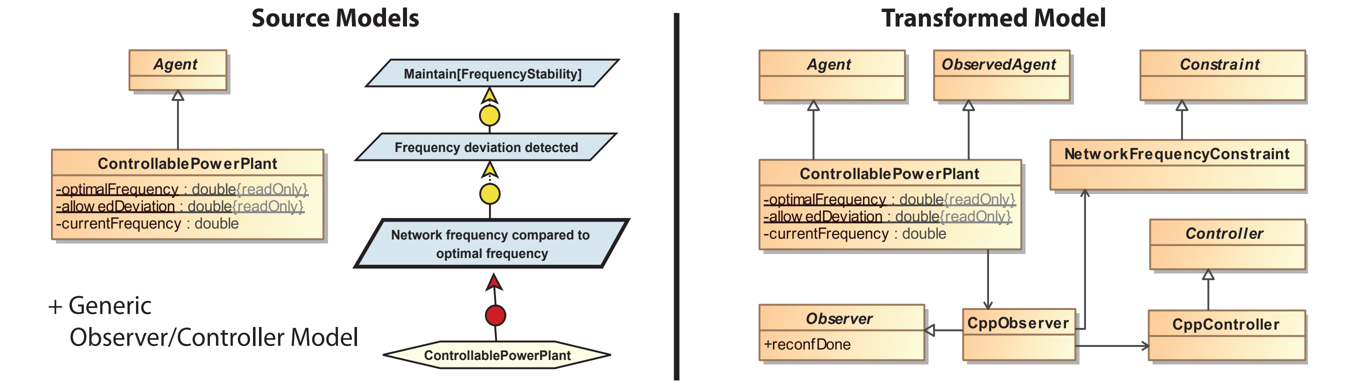
Source Models
Agent
ControllablePowerPlant
-optimalFrequency : double{readOnly}
-allowedDeviation : double{readOnly}
-currentFrequency : double
Maintain[FrequencyStability]
Frequency deviation detected
Network frequency compared to optimal frequency
ControllablePowerPlant
+ Generic Observer/Controller Model
Transformed Model
Agent
ObservedAgent
Constraint
ControllablePowerPlant
-optimalFrequency : double{readOnly}
-allowedDeviation : double{readOnly}
-currentFrequency : double
NetworkFrequencyConstraint
Controller
Observer
+reconfDone
CppObserver
CppController

Fig. 6. The sources of the transformation (simplified representation) and the resulting simplified class diagram for the example given in Sect. IV.

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.

A concrete observer implementation will have to be coupled with a controller that adapts the system accordingly. For the example given, there are a number of decentralised approaches that change the output of power plants to stabilise the network frequency [10].

VII. Discussion and Outlook

In this paper, we presented an approach to transform formally specified requirements into observers that are able to monitor an autonomous evolving system at runtime. Even though not all details of the approach have been elaborated yet, we believe that this approach will provide system engineers with an accessible way to integrate online monitoring into such systems. 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.

There are some limitation to the current status of the approach that have to be noted. First of all, the observed agent has to provide information about its internal state voluntarily. The assumption that an agent would always do this does not hold in open heterogeneous multi-agent systems. In general, it is desirable to only monitor externally observable properties, but this limits the applicability of any monitoring approach severely. Second of all, observing constraints that are defined over the state of several agents is not possible with the 1:1 relation between agent and observer that we showed here. Instead, a a regional view of the system would be helpful where an observer can monitor several agents and obtain a picture of a compartment of the system.

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. Finally, an interesting side effect of the way we propose to specify constraints in the requirements model is that it should be relatively straightforward to generate rely/guarantees from the requirements models for use as a formal system specification in verification [4].

Acknowledgement

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

References

[1] 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.

[2] 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.

[3] 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 Basel, 2011, pp. 79–93.

[4] 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. Lecture Notes in Computer Science. Springer Basel, 2013, vol. 7740.

[5] 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.

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

[7] 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.

[8] 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. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5795, pp. 468–483.

[9] 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

[10] 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 *2012 Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE Computer Society, Washington, D.C., 2012, pp. 29–38.