Correct by Prognosis: Methodology for a Contract-based Refinement of Evolution Models *

Christoph Etzien and Tayfun Gezgin

OFFIS, Escherweg 2,
26121 Oldenburg, Germany,
{christoph.etzien,tayfun.gezgin}@offis.de

Abstract. The scope of this paper is collaborative, distributed safety critical systems which build up a larger scale system of systems (SoS). Systems in this context are independently designed and can operate autonomously following both global SoS goals and individual goals. A major aspect of SoSs is the evolution over time, i.e. the change of its architecture as a result of changes in the context of the SoS or the changes of individual or global goals. The aim of this paper is to define a modeling concept for evolution specifying all possible changes of the SoS over time. This evolution model is used to generate and analyze future architectures enabling the prediction of future violations of static specifications. We derive so called *dynamicity contracts* and restrict the evolution model in such a manner, that *false* architectures are not reached.

1 Introduction

In recent years the co-operations and inter-connections between individual, geographically distributed systems heavily increased, leading to a new paradigm called *Systems of Systems* (SoSs). Also in safety critical areas the significance of these topics increased. As an example, much effort has been invested in the development of *Car-to-Car* communications with the aim to increase the safety in traffic and optimize traffic flows. Another example is the dynamic partitioning of the airspace with respect to time investigated in the SESAR (Single European Sky ATM Research) program. The recent partitioning of the airspace is performed in a statical manner with respect to time, i.e. the trajectories are not changed during the whole landing approach and the take off. The shift to a *dynamic* partitioning, which is called 4D-trajectories, involves a much more intensive co-operation between the tower and each airplane.

To distinguish between complex systems and SoSs, Mark Maier defined a set of characteristics [1], like the *geographical distribution* or the *operational independence*. The more a system exhibits these characteristics, the more it is an

* This work was supported in part by European Commission for funding the Large-scale integrating project (IP) proposal under the ICT Call 7 (FP7-ICT-2011-7) "Designing for Adaptability and evolutioN in System of systems Engineering (DANSE)" (No. 287716).

SoS. The main characteristic we are interest in is the *evolutionary development*, i.e. the change of the architecture of an SoS during its lifetime. A model for the evolutionary development can be created based on prognosis on possible future evolutions of the SoS. As an example, statistical data could be used to do a prognosis on the future traffic density in a district of a city. We propose *graph grammars* to model the possible evolutions of an SoS. These transformations could also be specified via temporal logics as we proposed in [2]. However, the specification with graph grammars is more intuitive than temporal logic.

Graph grammars describe the adaption to a context change in form of transformation rules. With these transformation rules the inter-connections of the constituent systems and thereby their roles and interaction protocols are changed. The trigger of such transformation rules are the constituent systems itself: When systems adapt or change their local goals and thus affect their local behavior, change their services offered to the environment, or need some services from their local environment, a request to change some parts of the SoS architecture are triggered from the corresponding constituent systems. In [2] we already discussed the initiations for transformation rules from constituent systems.

Besides the evolution model, we will consider static specifications of SoSs by contracts defining invariants and constraints on the architecture of the SoS. An example for a static contract is that systems applying inconsistent roles should not co-operate. The set of static contracts of an SoS defines all legal architectures of this SoS. Beginning with an initial SoS architecture, the evolution model successively generates a set of successor architectures. Transformations are applied locally resulting in sequences of transformations which could lead to an architecture violating the static contracts. In this paper, we derive so called *dynamicity contracts* which restrict the dynamics of the evolution model of the SoS to prevent the SoS entering an architecture which violates its static specification. We extend this approach by tolerating a finite set of intermediate architectures, which violate the static specifications. These intermediate architectures have to be left finally and a *safe* architecture has to be reached within a specified number of changes. The idea to allow temporarily intermediate faulty architectures is inspired by the *fault tolerance time intervals* defined in the ISO 26262 [3]. After the occurrence of a fault, a safe system state has to be reached within a defined time interval. If this interval exceeded, an hazardous event could occur.

To model the static architecural part of an SoS we use the UPDM framework [4]. UPDM is a unified Profile for DoDAF (Department of Defence Architecture Framework) and MODAF (Ministry of Defence Architectural Framework). It supports the capabilities to model architectures of complex systems, system of systems, and service oriented architectures. Beside milestones, no dynamicity aspects of systems of systems were considered in this framework.

1.1 Related work

In [5] a method for modeling and analyzing the dynamicity for multi-hop ad hoc networks was presented. Statistical estimation theory was applied to model the so called *configuration* of a multi-hop wireless network. In [6] a supporting model

called *dynamicity aware graph relabeling system* is introduced. This model is used for ad-hoc networks to take mobility into account. Ultra large scale systems are the topic of [7], where the main characteristics are captured and specified, e.g. decentralized control, conflicting requirements, and continuous evolution. In [8] some major issues in self-coordinating systems are depicted. The main statement is that a tight integration of all disciplines in the development process of such large scale self-coordinating systems has to be established. An approach for the design and analysis of multi-agent systems was presented in [9]. Agents are able to sense and manipulate specific aspects of the environment. Sets of agents form community types, which interact in the modeled environment with some interaction specifications. In [10] self-adaptive systems were presented. Initially a system architecture with defined components, their interfaces, and a set of coordination pattern is given. Coordination pattern define protocols between components via roles. Reconfigurations are defined via graph transformation rules and are initiated by environmental changes.

Automatic verification of the real-time behavior including the reconfiguration is supported by CHARON [11], Masaccio [12], and *Mechatronic UML* [13]. There are some approaches for modeling the structural aspects of adaptive systems [14, 15] or the behavioral aspects [16, 17] but none of them consider both aspects.

1.2 Outline

The following section introduces the fundamentals of our work, i.e. the considered modeling formalism called UPDM, the contract-based specification formalism, and the formalisms needed to express transformations. Section 3 illustrates our approach to derivate dynamicity contracts in order to prevent the SoS to evolve in architectures which violate its static specification. In Section 4 we illustrate our implementation and give some example scenarios. Finally, Section 5 concludes the paper and discusses some further work.

2 Fundamentals

The basic modeling elements of our approach are components as structural elements, and graph grammars. Our components are enriched by so called contracts, specifying the allowed context of a component and its guaranteed behaviour. Components and contracts are detailed in the following section.

2.1 Contract-based Modelling

We use *Heterogenous Rich Components* (HRCs) [18, 19] to model systems and its artifacts in a black box manner. The dynamics of an HRC can be specified by, e.g., an external behavior model. For each HRC a set of specifications in terms of contracts [20] is defined. A contract is a pair consisting of an assumption (A) and a guarantee (G). The assumption specifies how the context of the component, i.e. the environment from the point of view of the component, should behave. Only if the assumption holds, then the component will behave as guaranteed.

The system decomposition can be verified with respect to contracts without the knowledge of the concrete implementation. The specification of both assumptions and guarantees can be provided based on a pattern based language like introduced in [21].

Having a formal specification for the component and its sub-components the so called *Virtual Integration Test* (VIT) [22] can be performed. It is called virtual because no implementation for the sub-components or any testbed is required. This analysis is performed based on the specifications, the interfaces, the connections, and the structure of the composition. This test checks if the composition of the sub-component contracts implies the contracts of the surrounding component. In this work we assume that the components are implemented according to their contracts and call an architecture *valid* iff the VIT is successful.

2.2 Rewriting Rules

An *architecture* of an SoS is a composition of CSs at a specific time, where *roles* and inter-connections of all systems are specified. Changes of an architecture of the SoS are defined by a set of rewriting rules. Rewriting rules consist of a left hand side and a right hand side corresponding to architectures of the SoS. In this work, rewriting rules restructure the architecture of an SoS by composing or separating system instances in a well-defined way, and applying the right roles to the corresponding systems. So, a single transformation affects a subset of participating system instances, their inter-connections, roles, and modes. In the following, we will formalize the concept of graphs and rewriting rules.

A graph is defined as a tuple $G = (V, E, s, t)$ where V is a set of vertices, E is a set of edges and s, t are a source and a target function defined as $\{s, t\} : E \to V$. Let L, R be two graphs. A rewriting rule $r : \ L \to R$ is defined in such a way, that whenever an instance of L, called *match*, is found in a graph G, this instance can be replaced through an instance of R leading to the transformed graph G'. For two graphs H, G let $h : H \to G$ be a *graph homomorphism*, mapping nodes and edges of H to G. The homomorphism consists of two functions $h_V : V_H \to V_G$ and $h_E : E_H \to E_G$, such that $\mu_G \circ h_E = h_V \circ \mu_H$, with $\mu = \{s, t\}$. A rule $r : L \to R$ can be applied to a graph G leading to a changed graph G', in short $d : G \to_r G'$, if there exist two homomorphisms h_1, h_2, such that $h_1 : L \to G$ and $h_2 : R \to G'$. In Section 3 we will apply a set of rewriting rules specifying the dynamic behaviour of an SoS.

2.3 Modeling the SoS

System of systems (SoS) consist of several constituent systems (CS) which are instances of systems or even SoSs themselves. To distinguish between complex systems and SoSs Maier proposed five criteria for the "SoS-ness" of a complex system, which are introduced in the following [1]:

- *Operational independence of elements*: The CSs can operate independently.
- *Managerial independence of the elements*: The CSs are separately acquired by different managerial entities.

- *Evolutionary development*: An SoS evolves over time, developing its capabilities as the CSs are changed, added, or removed.
- *Emergent behavior*: The SoS itself offers additional services beyond the capabilities of the CSs including unexpected and potentially damaging behaviors.
- *Geographic distribution*: The geographical extent of the CSs could be "large".

We focus on the *evolutionary development* aspect of SoS and therefore concentrate not only on the architecture at a specific time but also on the evolution of the CSs and their re-configurations. We distinguish two levels of behavior, i.e. *system dynamics*, and *evolution*. System dynamics deal with the question, how systems exchange data via their inter-connections. The topic *evolution* poses the question, how systems and their inter-connections are changed over time.

System dynamics are covered by the UML/SysML behavioral models and diagrams, e.g. state charts. We use contracts to specify the assumed and guaranteed behavior of each CS.

We address the evolutionary development and extend the milestone-based representation of SoSs in UPDM. A milestone represents an architecture of the SoS at a specific point in time. We will focus on the system view which basically represents *systems* itself, their *resource roles* and inter-connections. The SV-1 allows to characterize the inter-connections of the CSs for a single architecture. The milestone plan (AV-2) is the planed evolution of the SoS taking the entire life-cycle of the CS into account. This plan is created manually and explicitly since each milestone consists of an entire SoS architecture.

The problem we address is that the owners or managers of the CSs follow their own goals, and change or influence changes of their CSs independently from a central authority. We define *goal* as an optimization metric which represents how good (or bad) a CS (or an SoS) performs. These values can be statically computed for an architecture of the SoS, or depend on the system dynamics and are measured during execution. The owners of a CS are assumed to monitor this values and decide to change the behavior or connectivity of their CS to improve their goals. This change might take place on the CS level by switching into another mode or on the SoS level by changing the inter-connections to other CSs. In the first case this behavior is part of the CS specification and covered by the contracts of the CS. In the second case the change is beyond the system borders of the CS and therefore not in the scope of the specification of the system dynamics. The evolution behavior of the SoS is based on changes which might impact the dynamics of the SoS.

3 From Evolution Model to Contracts

A model for an SoS consists of an initial architecture, a static specification and an evolution behaviour. As said before, evolution behaviors define the possibility of re-configuring a given architecture as a result of e. g. changing environmental conditions, or some adaption of cooperations between a set of systems. We will apply graph grammars to model such a behaviour. The benefit of the usage

of evolution models is the possibility to generate and analyze future architectures enabling the prediction of future violations of specifications. Changes of a given SoS can be explored before they occur in *reality* in order to prevent invalid architectures of the SoS. Typically an infinite number of architectures will be generated by graph grammars. In our concept, we will apply the concept of bounded model-checking, i.e. we will only consider a finite number of architectures reached by a grammar specification. This also has a practical relevance, as in general the evolution model shall only predict the possible behaviour for a finite time frame instead of an infinite time frame. To obtain a finite set of architectures, we could apply abstraction techniques like the *Partner Abstraction* introduced in [23].

3.1 Derivation of Evolution Contracts

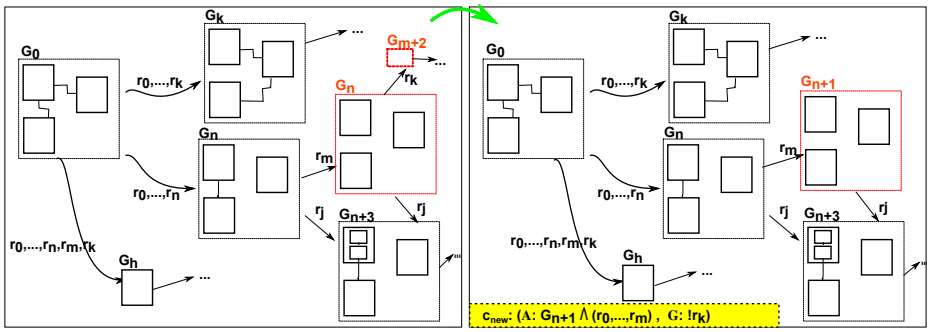


Fig. 1: Example scenario for derivation of evolution contracts – Left: Initial situation for a given SoS model; Right: Derivation of new dynamicity contracts.

The concept of dynamicity contracts complements the static contracts for the SoS and constituent systems. The static contracts restrict the allowed behaviour of the overall SoS and each system, whereas the dynamicity contracts restrict the dynamics of the evolution model of the SoS.

Starting from an initial architecture each reached architecture is analyzed if it is valid. If invalid architectures are reached, a dynamicity contract is derived in such a way, that the evolution model is prevented to generate this architecture. Thereby, the assumption part of a dynamicity contract encapsulates the architecture, from which a violating one can be reached by the application of a rule defined in the evolution model. The guarantee part then consists of the negation of the corresponding identifier of the rule. Further, we extend this approach by allowing intermediate architectures, which violate the static specifications, if a valid architecture is reached after "some time". In this work, we require that a specified *number* of successive invalid architectures may be tolerated, and after this number a valid architecture has to be reached. In future work, we will extend this approach by specifying some allowed *time frames*.

Consider the example of Figure 1, where the initial architecture G_0 can evolve to different future architectures by applying an evolution model consisting of a set of rewriting rules r. Assume that we allow that during the evolution maximal a single architecture may be reached which violates the static contracts. On the left part of the figure the initial situation is depicted, where no restrictions exist so far for our evolution model. If the sequence of rules $r_0, ...r_n, r_m$ is applied, we can reach the architecture G_{n+1} which violates the static specification. If we would now apply rule r_k we would again get an architecture violating the static specification. In order to restrict our evolution model we derive the contract illustrated in the right part of Figure 1. The contract states, that whenever we are in an architecture isomorph to G_{n+1} and we previously applied the sequence of rules $r_0, ..., r_m$, the rule r_k will not be applied. Note, that we need the architecture within the assumption part, as rewriting rules are non-deterministic. The sequence of rules $r_0, ..., r_m$ can also lead to some architectures not violating the static contracts as illustrated in Figure 1.

Next, we define our applied graph grammar formalism, and formalize the derivation of dynamicity contracts.

Graph Grammars Let $w = r_0, ..., r_n, ...$ be a word over an alphabet Σ, $pre(w, n) = r_0, ..., r_n$ be its prefix consisting of $n+1$ symbols, and $w(n) = r_n \in \Sigma$ the $(n+1)-$th symbol. A *dynamicity contract* is a contract talking about graphs and prefixed of words: The assumption (`A`) part of a dynamicity contract c consists of a (possibly empty) finite prefix of a word w and a graph G, its guarantee (`G`) consists of a symbol in Σ, in short $c:\ (\texttt{A}:\ pre(w, i-1) \wedge G_i,\ \texttt{G}:\ !\sigma)$ for some $i \in \mathbb{N}$ with $\sigma \in \Sigma$. The intuition is that whenever a finite sequence of symbols $pre(w, i-1)$ is received and the graph G_i is reached, the next symbol shall not be σ. With these dynamicity contracts we will restrict graph grammars in such a way, that through the successive application of rules it always holds, that no graph can be reached violating some static specifications.

A graph grammar is a tuple $\mathcal{G} = (G_0, R, C_D)$ where G_0 is a start graph, $R = \{r_0, ..., r_k\}$ is the set of rewriting rules (each with an unique identifier), and C_D is the set of dynamicity contracts, which may be empty at design time. In the next section we detail the iterative extension of this set. A graph grammar can be translated to a finite $\omega-$automaton $T_E = (S, s_o, \Sigma, \rightarrow)$, where S is a set of graphs corresponding to the set of states, s_0 the initial state, Σ an alphabet consisting of the identifiers of the rules in R, and $\rightarrow \subseteq S \times \Sigma \times S$ the transition relation. All states are considered to be accepting ones.

A run ρ of T_E over an infinite word $w = r_0, ..., r_n, ...$ is an infinite sequence of graphs $G_0 \rightarrow_{r_0} ... \rightarrow_{r_n} G_n, \rightarrow_{r_{n+1}} ...$ such that G_0 is the initial graph and $(G_i, r_i, G_{i+1}) \in \rightarrow$ for all $i, j \in \mathbb{N}$, for which holds that $(\texttt{A}:\ pre(w, i-1) \wedge G_i,\ \texttt{G}:\ !w(i)) \notin C_D$. The language of a graph grammar is defined as the set of words accepted by its finite automaton.

Derivation of Dynamicity Contracts A specification for an SoS is given by the tuple $SoS = (\mathcal{G}, C_s)$ where $\mathcal{G}$ is a graph grammar specifying the evolution model, and C_s a set of static contracts defining the allowed SoS behaviour. In

general, the evolution model specified through the concept of graph grammars is not initially consistent with the static specification specified as a set of contracts, as rewriting rules are applied locally resulting in sequences of rules which could lead to a graph violating the static contracts. This can happen because the application of a rule does not check whether the reached graph harms a static contract. In order to make the evolution model consistent with respect to the contract specification, such paths have to be removed from the evolution model. For this we derive new dynamicity contracts from these paths.

The easiest case is given, when a direct application of a rule violates a static contract and no intermediate architectures violating contracts are allowed. For such cases we can derive a dynamicity contract consisting of the current architecture G as the assumption part, and the negation of the identifier of the corresponding rule for the guarantee part. That is, we extend our dynamicity contract set C_D of $\mathcal{G}$ with the contract $\{(\texttt{A}:\ G,\ \texttt{G}:\ \sigma)\}$, if there exists a rule $\sigma : L \to R$ in $\mathcal{G}$, and $G \to_\sigma G'$ could be applied, such that $G' \not\models C_s$.

With this extension the graph grammar will be prevented by firing rule σ when an isomorphic graph to G is present. If violating graphs are accepted temporarily, e.g. a finite amount of time, or a finite number of violating graphs, we need to extend such dynamicity contracts with the history which lead to a corresponding architecture. In this work, we will only consider the maximal successive number of incorrect intermediate architectures, i.e. architectures violating the static specification.

Let $\xi \in \mathbb{N}$ be the maximal number of successive graphs violating the static specification, which is defined to be tolerable. Let $pre(w, n) = r_0, ..., r_n$ be a prefix of a word, for which there exists a run $\rho = G_0 \to_{r_0} ... \to_{r_i} G_i ... \to_{r_n} G_n$ of the automaton of $\mathcal{G}$, such that $G_i, ..., G_n \not\models C_s$ and $|\{G_i, ..., G_n\}| > \xi$. Then we derive the following dynamicity contract and extend the set C_D as follows:

$$C_D \cup \{(\texttt{A}:\ G_{n-1} \wedge pre(w, n-1),\ \texttt{G}:\ !w(n))\}. \tag{1}$$

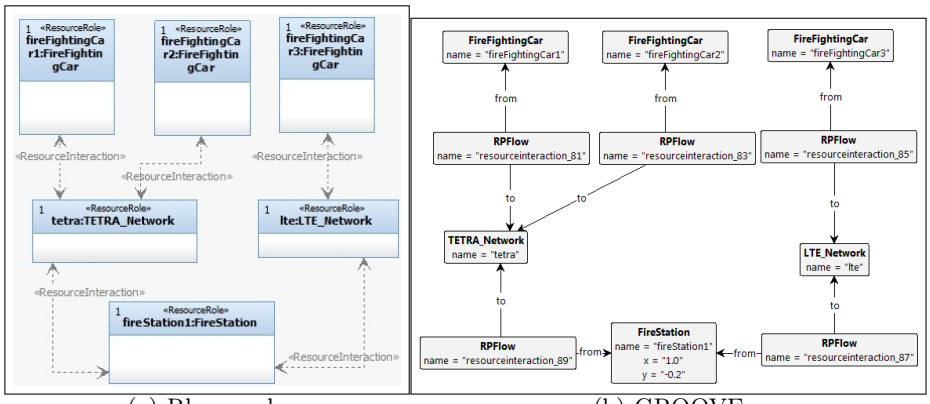
Note that a word w could result in a set of runs instead a single run. In this case our new dynamicity contracts are *correct* in the sense, that no *legal* evolutions resulting in graphs which all fulfill the static contracts are excluded. This is because the assumption part exactly states, that a rule shall not be applied if a specific architecture is given.

4 Application of Methodology

To illustrate our approach we consider an emergency response scenario, consisting of a set of constituent systems like fire stations and fire brigades. All CSs participating in this SoS shall behave cooperative in order to minimize the needed time for an operation in case of an emergency.

We use a new custom diagram via an additional profile which allows to model rewriting rules graphically in IBM Rational Rhapsody©. These diagrams allow to add placeholders which refer to model elements of the Rhapsody UPDM model. This reference mechanism ensures that the model itself and the rewriting

rules are clearly separated. The rules contain four different kinds of graphical elements for each CSs and their inter-connections, i.e. *Reader*, *Creator*, *Eraser* and *Embargo*. *Reader* elements are unchanged elements of a corresponding rule. *Creator* elements represent newly generated elements on the right hand side of the rule. *Eraser* elements address elements of the left hand side which are removed via the rule application. *Embargo* elements restrict the applicability of the rule if the match can be extended by these elements. The Rhapsody model including its rules are exported to GXL[24] files which are the input language of the GROOVE[25] tool. GROOVE is used for the generation of architecture alternatives and is also able to perform the isomorphism check of the generated architectures. After applying GROOVE we get a set of architectures, and the corresponding network representing the applied rules. As an example consider

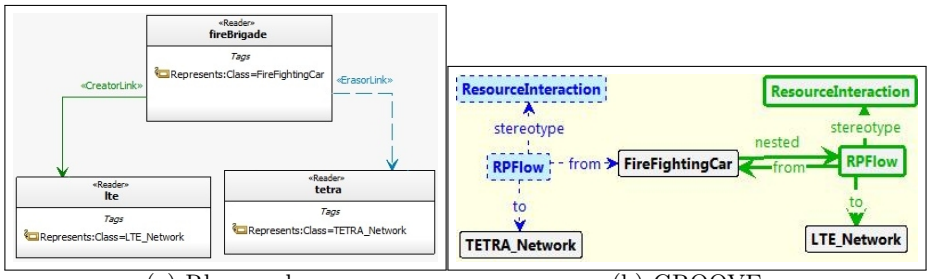


(a) Rhaposody (b) GROOVE

Fig. 2: Excerpt of the Emergency Response System

Figure 2 and 3. The purpose of the fire service is to delete fire at any location within a city and to save the involved people. The time between the incident harms people and the treatment begins is critical for the recovery of the injured. Therefore the goal of the fire service is to minimize the time between the notification and the arrival of the right amount of units to treat the injured people at the incident location. Increasing traffic density typically extends this time frame and might require to send units from locations with a larger geographical distance but lower distance in travel time. To improve this, one option is to increase the number of units like fire brigades but this is only partially possible. Another option is to increase the awareness of the fire head quarter about the required number (and kind) of units at the location. This can be achieved by improving the communication technology, in this scenario the change from the current TETRA [1]) to the LTE[2]) communication technology. The application of such a rule leading to an architectural change is illustrated in Figure 3(b). In

[1] TETRA: *Terrestrial Trunked Radio*, ETSI EN 300 392-2 v3.2.1
[2] LTE: *Long-Term Evolution*



(a) Rhaposody (b) GROOVE

Fig. 3: Rule Translation: Rules in Rhapsody (a) are automatically translated into rules in GROOVE (b)

this example, the evolution model contains only a very small set of architectures because the rule is only applicable once per fire brigade and the number of fire brigades is low. If one would add a rule adding fire brigades to the model the number of architectures would be infinite. In the complete model several fire stations are coordinated by one head quarter and also the number of fire brigades is higher. Since the fire brigades are coordinated by the different fire stations and must cooperate during operation it is essential that those brigades use the same communication network. If each brigade is updated to the new technology individually, invalid architectures are possible which can be characterized as (at least) two brigades coordinated by the same fire station using different networks. The evolution must be restricted to avoid those architectures. Typically not all those constraints can be derived from reasoning about architectural pattern only but the reachable architectures have to be analyzed including the system dynamics. This can be done via simulation or static analysis (e.g. timing analysis as proposed in [2]). The results of the analysis are annotated to the reachable architectures and support the identification of contracts for the evolution itself.

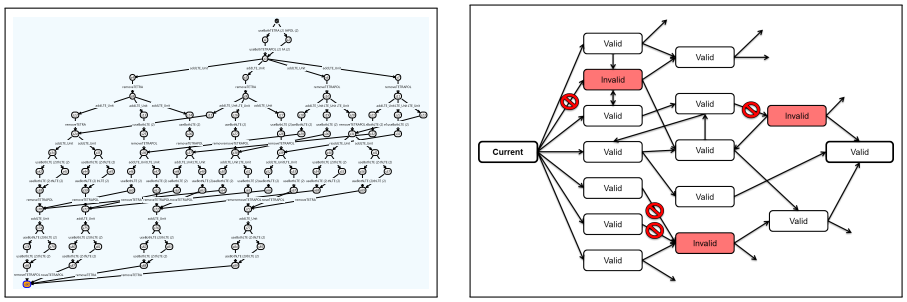


Fig. 4: Left: network of architecture alternatives; Right: annotated network.

In the left part of Figure 4 a network of reachable architectures is illustrated. In the right part of Figure 4 a (simplified) network of architectures is presented.

For this network the invalid architectures are marked in red. From this network global constraints are derived which restrict the application of rules. These conditions are the previous architectures of any edge ending in an invalid architecture. The evolution contract takes this condition as assumption and the negated invalid architecture as guarantee.

5 Conclusion

We presented a modeling concept for evolution specifying all possible changes of the SoS over time as an extension of the UPDM framework. We introduced a novel approach for deriving dynamicity contracts restricting such evolution models in order to prevent reaching invalid architectures with respect to the static specification of an SoS. Our prototype implementation offers so far an export mechanism from UPDM models created with Rhapsody to GROOVE, and feed back the generated architecture alternatives to Rhapsody. For the generated models we can apply our previously introduced virtual integration checker [26] and manually derive dynamicity contracts. Currently, we aim to close this loop, i.e. the generation of architectures and calling the verification back end to automatically generate dynamicity contracts. In future work we also plan to include the notion of time for the evolution models to enable reasoning about timing constraints for the evolution.

References

1. W.Maier, M.: Architecting principles for systems-of-systems. In: Inc. Systems Engineering. Volume 1. (1998) 267–284
2. Etzien, C., Gezgin, T., Fröschle, S., Henkler, S., Rettberg, A.: Contracts for evolving systems. In: SORT – The Fourth IEEE Workshop on Self-Organizing Real-Time Systems. (06 2013)
3. ISO26262: Road vehicles – functional safety (2011)
4. Group, O.M. In: Unified Profile for DoDAF and MODAF. (2008)
5. Hamlili, A., Morocco, R.: A common computational approach analyzing dynamicity and connectivity for reliable communications in multihop wireless networks. In: Int. Conf. on Models of Information and Communication Systems. ICST Alliance
6. Casteigts, A., Chaumette, S.: Dynamicity aware graph relabeling systems (da-grs), a local computation based model to describe manet algorithms. In: IASTED PDCS. In proceeding of: International Conference on Parallel and Distributed Computing Systems, PDCS, Phoenix, AZ, USA (2005) 231 – 236
7. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems - The Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon (June 2006)
8. Schäfer, W., Birattari, M., Blömer, J., Dorigo, M., Engels, G., O'Grady, R., Platzner, M., Rammig, F., Reif, W., Trächtler, A.: Engineering self-coordinating software intensive systems. In: Proceedings of the Foundations of Software Engineering (FSE) and NITRD/SPD Working Conference on the Future of Software Engineering Research (FoSER 2010). (2010)

9. Giese, H., Klein, F.: Systematic verification of multi-agent systems based on rigorous executable specifications. Int. J. Agent-Oriented Softw. Eng. **1** (2007) 28–62
10. Henkler, S., Hirsch, M., Priesterjahn, C., Schäfer, W.: Modeling and verifying dynamic communication structures based on graph transformations. In: GI Software Engineering. (2010)
11. Ivancic, F.: Modeling and Analysis of Hybrid Systems. PhD thesis, University of Pennsylvania (2003)
12. Henzinger, T.A.: Masaccio: A formal model for embedded components. In: IFIP International Conference on Theoretical Computer Science (TCS), LNCS1872, Springer, 549-563. (2000)
13. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Assmann, U., Rensink, A., Aksit, M., eds.: Model Driven Architecture: Foundations and Applications. LNCS, Springer Verlag (2005) 1–15
14. Métayer, D.L.: Software architecture styles as graph grammars. In: SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, New York, NY, USA, ACM (1996) 15–23
15. Kramer, J., Magee, J., Sloman, M.: Configuring distributed systems. In: EW 5: Proceedings of the 5th workshop on ACM SIGOPS European workshop, New York, NY, USA, ACM (1992) 1–5
16. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. Lecture Notes in Computer Science **1382** (1998) 21–36
17. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study. In: CDS '98: Proceedings of the International Conference on Configurable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1998) 91
18. Hungar, H.: Compositionality with strong assumptions, Mälardalen Real–Time Research Center (11 2011) 11–13
19. Baumgart, A., Böde, E., Büker, M., Damm, W., Ehmen, G., Gezgin, T., Henkler, S., Hungar, H., Josko, B., Oertel, M., Peikenkamp, T., Reinkemeier, P., Stierand, I., Weber, R.: Architecture modeling. Technical report (03 2011)
20. Meyer, B.: Applying "design by contract". Computer **25**(10) (1992) 40–51
21. CESAR SP2 Partners: Definition and exemplification of requirements specification language and requirements meta model. CESAR_D_SP2_R2.2_M2_v1.000.pdf on http://www.cesarproject.eu/fileadmin/user_upload/ (2010)
22. Damm, W., Hungar, H., Josko, B., Peikenkamp, T., Stierand, I.: Using contract-based component specifications for virtual integration testing and architecture design. In: Design, Aut. and Test in Europe (DATE 2011. 1–6
23. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In Nielson, H.R., File, G., eds.: Static Analysis, Int. Symposium, SAS 2007. Volume 4634 of LNCS., Springer (2007) 249–264
24. Winter, A., Kullbach, B., Riediger, V.: An overview of the gxl graph exchange language. In: Revised Lectures on Software Visualization, International Seminar, London, UK, UK, Springer-Verlag (2002) 324–336
25. Rensink, A.: The groove simulator: A tool for state space generation. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE). Volume 3062 of Lecture Notes in Computer Science., Berlin, Springer Verlag (2004) 479–485
26. Gezgin, T., Henkler, S., Stierand, I., Rettberg, A.: Impact analysis for timing requirements on real-time systems. In: Int. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2014). (To be published)