DOI 10.1007/s00165-015-0346-y
BCS © 2015
Formal Aspects of Computing (2016) 28: 21–43

Formal Aspects of Computing

Synthesizing structural and behavioral control for reconfigurations in component-based systems

Narges Khakpour[1,2], Farhad Arbab[2,3], Eric Rutten[4]

[1] Department of Computer Science, Linnaeus University, Växjö, Sweden
[2] Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
[3] CWI, Amsterdam, The Netherlands
[4] INRIA, Grenoble, France

Abstract. Correctness of the behavior of an adaptive system during dynamic adaptation is an important challenge to realize correct adaptive systems. Dynamic adaptation refers to changes to both the functionality of the computational entities that comprise a composite system, as well as the structure of their interconnections, in response to variations in the environment, e.g., the load of requests on a server system. In this research, we view the problem of correct structural adaptation as a supervisory control problem and synthesize a reconfiguration controller that guides the behavior of a system during adaptation. The reconfiguration controller observes the system behavior during an adaptation and controls the system behavior by allowing/disallowing actions in a way to ensure that a given property is satisfied and a deadlock is avoided. The system during adaptation is modeled using a graph transition system and properties to be enforced are specified using a graph automaton. We adapt a classical theory of supervisory control for synthesizing a controller for controlling the behavior of a system modeled using graph transition systems. This theory is used to synthesize a controller that can impose both behavioral and structural constraints on the system during an adaptation. We apply a tool that we have implemented to support our approach on a case study involving https servers.

Keywords: Dynamic reconfiguration, Synthesis, Control theory, Adaptive systems, Correct-by-construction

1. Introduction

The next generation of software systems includes systems composed of a large number of distributed, autonomous, interacting and continually evolving components and subsystems. These software systems are subject to adaptation at runtime due to changes in their operational environments and user requirements. In general, dynamic adaptation techniques are classified into two broad categories: structural adaptation and behavioral adaptation. While structural adaptation aims to adapt the behavior by changing a system's architecture, behavioral adaptation focuses on modifying the functionalities of the computational entities.

Correspondence and offprint requests to: N. Khakpour, E-mail: narges.khakpour@lnu.se

In order to have a safe structural adaptation in component-based systems, we must address several challenges.

First, structural adaptation may involve simultaneous changes in several independent components. It is obvious that runtime changes do not occur instantaneously, and the system is likely to move through several invalid configurations before reaching a final valid configuration; consequently, it is likely that some safety properties are violated in transient states. For instance, to replace a broken component with a new one, we must follow a plan that specifies the steps to remove the old component, connect the new component to the same components that the old component was connected to, and then allow the new component to start its execution. These components may be distributed all over a network, which makes it impossible to carry out the reconfiguration operations instantaneously, e.g. there may be a state where the new component and the old component coexist in the system, some of the connections of the old component are connected to the new component while the rest are still attached to the old version. Therefore, *correct design of a system such that the system satisfies some specific properties during adaptation is an important challenge.*

Second, preserving consistency of the state of a component or system after adaptation is an important challenge. It is often assumed that a component starts its execution in a predefined initial state. However, when a component is replaced by another one and the system switches to a new configuration, it is required for the new state to be consistent with the previous states. *Thus, it is necessary to determine the correct start state of a component after adaptation.* Observe that the definition of state consistency is application-specific and is determined by the verification engineer. For instance, if a server handling requests is replaced by a new server, the information of the handled requests by the old server and its waiting requests should be transferred to the new server, otherwise the requests will be lost.

Third, there must not be a deadlock in the adaptation phase and the system must successfully finish the adaptation by converging to its final states. If a deadlock occurs during an adaptation, the whole or part of the system will stop working, which subsequently impacts the availability of the system.

Researchers have already proposed techniques to synthesize adaptors to control interactions where the system is built by composing several (black-box) components/services. In [AFI+06, AMNT08, TFGG07], the authors propose algorithms for synthesizing adaptors to assemble component-based systems wherein component interactions and desired behavior are modeled using labeled transition systems. In another similar work [GMW12], a control-theory based approach is proposed to synthesize behavioral adaptors that are responsible to adjust the communication between some given services such that a certain behavioral property holds in the composed system. In both approaches, the aim of reconfiguration control synthesis is controlling the interactions among the components/services. However they are not concerned about the correctness of the adaptation phase. An approach based on the notion of transitional-invariant lattice is presented in [BK08, KB04] to verify the correctness of adaptation. To the best of our knowledge, there is no research done on synthesizing controllers to control the behavior of a system during a reconfiguration phase, i.e., while the system is in a transient state, undergoing a structural reconfiguration, which may involve sequences of actions that affect different components of the system, as the unaffected parts of the system continue to run concurrently.

To address the above challenges, we proposed an approach in [KAR14] to synthesize a non-blocking controller that controls the reconfiguration process such that the desired behavior, defined by the verification engineer, is preserved during a reconfiguration. We imposed properties only on the behavior of the system described using an ordinary automaton. We modeled the behavior of the system during reconfiguration with a graph transition system. Graph transition systems are classical transition systems augmented with a function mapping states into graphs, and transitions into partial graph morphisms. The graph of a state represents the structure of the system in that state.

We target component-based systems that employ dynamic reconfigurations to adapt system behavior and focus on non-quantitative safety properties and deadlock avoidance. Other topics in self-adaptive computing systems, which lie beyond our concerns, involve more quantitative properties, such as managing the number of servers in a data center while ensuring response time properties, as well as the notions of stability, robustness and fast convergence. These systems and properties would be approached with quite different models and control techniques, e.g., hybrid modeling, differential equations, and non-linear controllers.

The Ramadge–Wonham (RW) framework [RW87] is a classical automaton-based framework used to synthesize a controller. In this framework, the system model, so called the plant, and the specification to be enforced are specified by automata, and a controller is synthesized to control the plant's behavior such that the property is satisfied. This approach is one of the common supevisory controller synthesis approaches that has been applied to computing systems, e.g. [GN12, GVNH11]. Our choice for discrete control is motivated by the fact that it is a technique that solves safety problems from a description of possible behaviors and declaration properties of

interest, and therefore can support safe design. The systems of concern in our work consist of component assemblies that must always satisfy predefined structural and behavioral constraints. Graphs are a common and suitable mechanism to represent the structure of a system. To represent both the behavior and the structure of a system, we use a combined formalism to model its structural (graph-based) and its behavioral (automata-based) aspects. Accordingly, we need to adapt the automaton-based RW framework to a transition system that is enriched with structural aspects in order to enforce structural and behavioral constraints on the desired system behavior and the interactions among its components.
This paper In this work, we extend and improve our work by proposing a framework that allows us to (1) impose properties on both the behavior and the structure of the system, (2) to construct the behavior of the whole system during an adaptation given the initial system structure and each component behavior, and (3) to automate the controller synthesis. Moreover, we prove the soundness of our approach and apply it on a case study to evaluate the framework. Our approach consists of three steps shown in Fig. 1:

Step 1 We model the behavior of the system during reconfiguration, from when a reconfiguration starts until it ends, with a graph transition system $\mathcal{G}$ in [KAR14]. This model captures both the behavior and the structure of the whole system during a reconfiguration. To consider the environment behavior and its interaction with the system, one can over-approximate its behavior and model it as a component of the system. It is a cumbersome and error-prone process to manually construct a model that describes the whole behavior of a system, because the changing structure of the system influences the components' interactions. To address this problem, we specify the behavior of a component using a state transition system and formalize a plan for performing a reconfiguration by an algebra. Given the initial system structure, we apply the reconfiguration plan to the components' behavior specified by the state transition systems and construct a graph transition system $\mathcal{G}$ that models both behavior and structure of the system during an adaptation. We refer to the system actions by events that are the labels of the graph transition systems. An event represents either a behavioral event, e.g. sending a message, or a reconfiguration event, e.g. adding a component. The RW framework is an automaton-based approach which motivates our choice of state transition systems, and an algebraic formalism for describing reconfiguration plans provides a compositional higher-level specification compared to the state transition systems.

Step 2 We specify a property to be enforced during the adaptation phase, using an automaton $\mathcal{A}$ defined both on the structure and the behavior of the system. For example, one can constrain a system to keep two components A and B connected, as long as a component C is in the system, or a message is finally sent to component A.

Step 3 We compute the product of the graph transition system $\mathcal{G}$ and the automaton A denoted by $\mathcal{AG} = \mathcal{G} \bowtie A$. The graph transition system $\mathcal{AG}$ indicates the part of $\mathcal{G}$ that preserves the property $\mathcal{A}$. We use the Ramadge and Wonham framework [RW87] to refine $\mathcal{AG}$ to remove all bad states, i.e. states that cause deadlock or wherein an uncontrollable event is forbidden. An event (the label of a transition) is uncontrollable if it cannot be prevented from occurring in a system, e.g. the event of a component crash is uncontrollable, as we cannot prevent it from happening. The result is then a reconfiguration controller used to control the behavior of the system during an adaptation, i.e. the controller runs in parallel with the system, monitors the system, and allows/disallows the controllable events to preserve $\mathcal{A}$. Furthermore, we improve the approach for the cases that we have to enforce a purely structural property during a reconfiguration. Note that there may not exist a non-blocking controller that enforces a property during an adaptation, e.g., the synthesized controller may be an empty controller, which blocks unconditionally. In general, in the RW framework, when the control problem is over-constrained and the property to be enforced is strong, the set of remaining behaviors allowed by the controller may become empty. In that case, either the property should be weakened, or the system must be redesigned in order to make the control that preserves the property possible. In our case, for instance, this happens if an adaptation does not start in a proper state, in which case, the verification engineer must update the reconfiguration plan, or change the adaptation logic to start the adaptation from a suitable state such that the property is satisfied.

In our work, a reconfiguration can be triggered by the system or the environment. We may obtain different controllers to enforce a property depending on the controllability of the system events, if such a controller exists at all. Furthermore, the states from which a reconfiguration starts are usually determined by the system adaptation logic. The verification engineer can synthesize different reconfiguration controllers for different starting states of the system, and evaluate them according to her needs to find suitable states to start a reconfiguration. The starting state of a reconfiguration is determined based on the initial system structure and the current states of the components.

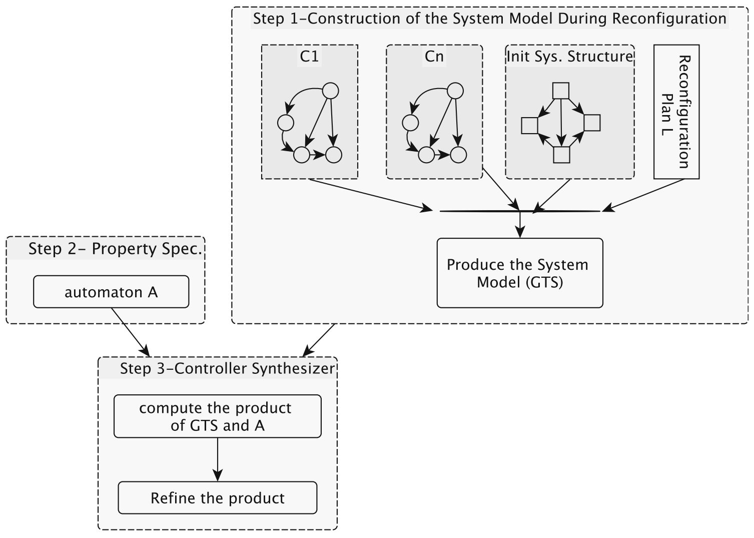

Step 1–Construction of the System Model During Reconfiguration
C1
Cn
Init Sys. Structure
Reconfiguration Plan L
Produce the System Model (GTS)
Step 2– Property Spec.
automaton A
Step 3–Controller Synthesizer
compute the product of GTS and A
Refine the product

Fig. 1. The outline of our approach

In this paper, we assume that (1) we know the behavior of every component, modeled as a transitions system, (2) some events in the system are controllable while some are uncontrollable, (3) the set of possible safe initial states for a new component is given by the verification engineer according to her needs, (4) the system states from which a reconfiguration can start and its current configuration are known, and (5) the verification engineer defines different plans for re-configuring the system. These assumptions may not always hold in all systems. For example in completely open systems components can come in whose behavior is not known, in which case there can be no guarantee about the safety of a reconfiguration, as we consider here.

The existing approaches proposed for safe adaptation have focused on controlling the interactions of components. In this paper, we propose a new approach to synthesize a controller for guiding the adaptation process safely. Our contributions are as follows:

- We customize the supervisory control problem for synthesis of systems modeled using a graph-based formalism. This customization enables us to consider both the structure and the behavior of a system for designing a correct controller.
- We use the extended supervisory control problem supporting structural modeling to design a correct non-blocking controller where the controller controls the adaptation process.
- We can enforce a range of properties defined on both the behavior and the structure of a system.
- Given the behavior of each component and the initial structure of a system, we use an algebraic method to construct a model of the system automatically. This model captures both the behavior and the structure of the system considering the components' interactions.
- We present a tailored method to synthesize a controller to enforce a purely structural property. This efficient method helps to reduce the complexity of the synthesis process by abstracting the internal behavior of the components.
- We implement a tool to support the proposed approach.
- We successfully use our tool in a case study to construct the system model during adaptation, and synthesize a controller to guide the adaptation process.

Structure of the paper This paper is organized as follows. We present a brief review of graphs and supervisory control in Sect. 2. Sections 3 and 4 concern modeling a system during the adaptation phase and property specification, respectively. We introduce our synthesis approach in Sect. 5. In Sect. 6, we present our implementation, and evaluate our approach by applying it on a case study. In Sect. 7, we discuss related work, and finally in Sect. 8, we conclude and discuss our plans for future work.

2. Preliminaries

2.1. Graphs overview

Consider an algebraic definition of labeled directed graphs. A labeled directed graph is a structure $G = \langle V, E, \sigma, \tau, lab\rangle$ where V denotes the set of vertices and $E \subseteq V \times V$ denotes the edge set of G. The functions $\sigma, \tau : E \rightarrow V$ are the source and target functions such that for $e = (x, y) \in E$, we have $\sigma(e) = x$ and $\tau(e) = y$. The edge labeling function $lab : E \rightarrow L$ maps every edge to a label in a fixed set of labels L.

Definition 1 (*Graph morphism*) Given graphs G_1 and G_2 with $G_i = (V_i, E_i, \sigma_i, \tau_i, lab_i)$ for $i = 1, 2$, a graph morphism $\psi : G_1 \rightarrow G_2$ is a pair of mappings $\psi = (\psi_V, \psi_E)$ where the functions $\psi_V : V_1 \rightarrow V_2$ and $\psi_E : E_1 \rightarrow E_2$ preserve the source and target functions, i.e. $\psi_V \circ \sigma_1 = \sigma_2 \circ \psi_E$, $\psi_V \circ \tau_1 = \tau_2 \circ \psi_E$ and $lab_1 = lab_2 \circ \psi_E$:

We say a graph morphism $\psi : G_1 \rightarrow G_2$ is injective, surjective or bijective if ψ_V and ψ_E are injective, surjective or bijective, respectively.

2.2. Supervisory control

The objective of the supervisory control problem is to synthesize a supervisor that constrains a system's behavior according to a given specification while ensuring controllability and coaccessibility. The coaccessibility property states that all states are coaccessible: a state is coaccessible if it is reachable from the initial state and from which a final state can be reached. Controllability refers to the fact that inhibition of controllable events makes it possible to enforce a property.

A solution to this problem is offered by the Ramadge–Wonham (RW) framework [RW87]. In this approach, both the possible behavior of a system and a specification for its desired behavior are given as finite automata. The system (also called plant) automaton (A) describes both wanted and unwanted behaviors, while the specification automaton is intended to restrict the possible actions to something useful. In this framework, events are categorized into two types: *controllable* and *uncontrollable*. Controllable events can be prevented from occurring in the system, while it is infeasible to prevent uncontrollable events from happening. In the RW framework, the aim is to design a nonblocking supervisor, denoted by S/A, that monitors the events and ensures that the system under supervision satisfies the specification by disabling/enabling the controllable events.

Let Σ represent an alphabet, $L \subseteq \Sigma^*$ be a language over Σ, and $\widehat{L}$ be the prefix-closure of L, i.e. $\widehat{L} = \{s \in \Sigma^* \mid \exists t \in \Sigma^*, st \in L\}$. L is called prefix-closed if $L = \widehat{L}$. In the RW framework, the plant is a discrete event system (DES) modeled by a finite automaton $A = \langle Q, \Sigma, \delta, q_0, Q_m\rangle$ where Q is the set of states, Σ is the finite set of events, $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function, q_0 is the initial state, and $Q_m \subseteq Q$ is the set of marked states.[1] The events are partitioned into two disjoint subsets, *controllable events* set Σ_c, and *uncontrollable events* set Σ_{uc}. We also introduce the extended transition function $\delta : Q \times \Sigma^* \rightarrow Q$ as usual. Let $L(A) = \{s \mid s \in \Sigma^*, \delta(q_0, s) \text{ is defined}\}$ and $L_m(A) = \{s \mid s \in L(A), \delta(q_0, s) \in Q_m\}$ be the closed and marked behaviors of A. The language $L(A)$ represents all possible paths of A from its initial state, while the marked language $L_m(A)$ represents only the accepting paths of A, i.e. the paths that end in a marked state. The DES A is blocking if $\widehat{L}_m(A) \subset L(A)$ and nonblocking if $\widehat{L}_m(A) = L(A)$, i.e. if all prefixes of $L(A)$ finally ends in a marked state, then A is non-blocking, otherwise, the system may follow a path in $L(A)\backslash\widehat{L}_m(A)$ and end in a state from which it cannot reach a marked state. Let S/A show a plant automaton A controlled by a supervisor automaton S, and define $L_m(S/A) := L_m(A) \cap L(S/A)$. The plant S/A can perform a controllable event, if it is allowed by S, and it can perform an uncontrollable event with no restriction from S. SCP (Supervisory Control Problem) is defined formally as follows:

Given a plant represented by an automaton A, a specification automaton with the language $E \subseteq L_m(A)$ representing the desired behavior of A under supervision, and a minimally acceptable behavior $A_{min} \subseteq E$, Supervisory Control Problem consists of finding a nonblocking supervisor S (i.e., $\widehat{L}_m(S/A) = L(S/A)$) such that $A_{min} \subseteq L_m(S/A) \subseteq E$.

[1] In the RW framework, marked states play the role of final states in an automaton, but they are not referred to as final states to avoid the connotation of termination. We also call them marked states here, to avoid the confusion with what we call the final states of reconfigurations, where we do require termination.

In the RW framework, first the product of the plant A and the specification E is computed. The standard automata product accepts exactly those paths of the plant that are consistent with the specification, i.e. $L(S \times A) = L(A) \cap L(S)$. Then, the result is refined to obtain a non-blocking controller.

3. Modeling dynamic reconfiguration

As the first step of our approach, in this section we concentrate on modeling the system behavior during a reconfiguration. The reconfiguration of a system is performed using *reconfiguration plans*: a reconfiguration plan describes different strategies (sequences of actions) to reconfigure a system structure to reach a target structure, for example, there can be two strategies to remove a component A and add a component B: first add B and then remove A, or the other way around. Given the initial and final configurations of a system, one can in principle obtain a reconfiguration plan automatically that contains all strategies to reconfigure the system. Although, automatic generation of reconfiguration plans can be convenient from a practical point of view, such synthesis may be very expensive, if possible at all. Hence, we assume that the verification engineer specifies a reconfiguration plan and removes unsuitable strategies according to her domain knowledge.

We specify the behavior of each component using deterministic state transition systems, and specify a reconfiguration plan using an algebra. Given the behavior of all components in a system and a reconfiguration plan, we obtain a graph transition system that models both behavior and structure of the system during reconfiguration. In our model, behavioral events arise from the internal behavior of components and their synchronous communications. Structural events, on the other hand, may be triggered by internal computation or in reaction to external changes in the environment, and require modifications to the graph that represents the structure of the interconnections of the components in the system. In this section, we explain how we build the whole system model, to accommodate the handling of both behavioral and structural events.

(A) Modeling the behavior of single components

As mentioned above, we specify the behavior of a component as an ordinary state transition system:

Definition 2 (*State transition system*) A state transition system is a tuple $\mathcal{T} = \langle S, \Sigma, \rightarrow, s_0, F \rangle$ where S is a set of states, Σ is a (finite) set of events, $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation, s_0 is the initial state, and $F \subseteq S$ is the set of final states.

An event $e \in \Sigma$ can be an input, an output or an internal event. An input event, denoted by $e?$ is used to receive a message, an output event, denoted by $e!$ is used to send a message, and an internal event e represents an internal computation. It is worth mentioning that a component has no final state.

Example 1 Figure 2 shows the behavior of four components A, B, C and D. The component A, first performs an internal action a, then either synchronizes on the event d, or synchronizes on the event sequences $c1$-d or $c1$-$c2$-d. Afterwards, the component A performs the internal event e. The component B synchronizes on $c1$-d, the component C synchronizes on $c1$-d-$c2$, and the component D synchronizes on d.

(B) The system structure modeling

We model the structure of a system as a graph, which represents the components and their interconnections. The components can interact with each other through synchronous communications, i.e., two active connected components c_i and c_j may synchronize on an action α. If there is no event issued by the other components to synchronize on, the component will block. Multi-party synchronization is not supported in our model and if a component is connected to multiple other communication enabled components, it can synchronize with only one of them at a time. Figure 3 is an example showing the structure of a system before and after a reconfiguration.

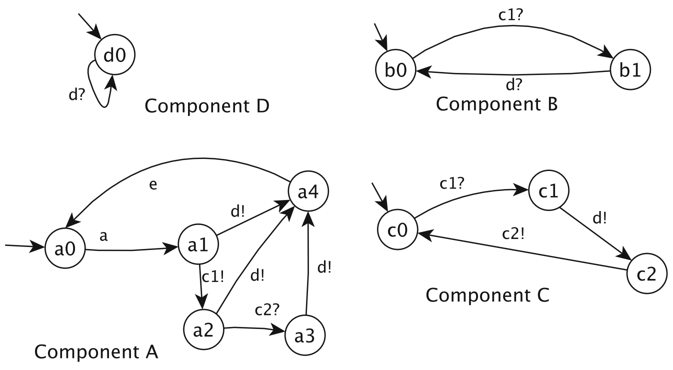


Fig. 2. The behavior of the components

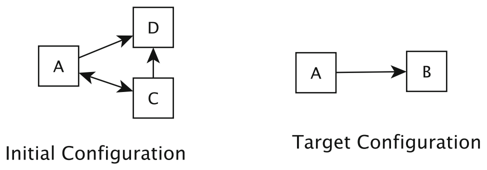


Fig. 3. The initial structure and the target structure

(C) Reconfiguration plans

A reconfiguration starts from an initial configuration and applies a set of atomic reconfiguration actions to obtain the target configuration. We use the following language to specify an algebra of reconfiguration plans in which α is a primitive reconfiguration action. The set of primitive structural actions includes $add(c)$, $del(c)$, $con(c, c')$, and $dis(c, c')$ for respectively adding a new component c, removing the component c, connecting c to c', and disconnecting c from c'.

$$a := a;\ a' \ \mid\ a \parallel a' \ \mid\ a + a' \ \mid\ \phi :\!\rightarrow a \ \mid\ \alpha$$
$$\alpha := add(c) \ \mid\ del(c) \ \mid\ con(c, c') \ \mid\ dis(c, c')$$

Thus a reconfiguration plan can be a sequential composition (;), a parallel composition ($\parallel$), an internal non-deterministic choice (+), or a conditional choice ($\phi :\!\rightarrow a$ where ϕ is a condition defined on the system structure). The non-deterministic choice operator is a source of choice points, which collectively form the domain of controllability. The standard operators $\parallel$ and + are commutative. We don't allow recursion in this language to ensure the termination of reconfiguration plans. Figure 4 shows the operational semantics of the reconfiguration algebra. SR1 represents the execution of a primitive action α. SR2 and SR3 define the semantics of non-deterministic choice and SR4-SR9 apply sequential, parallel compositions and conditional choice of actions. The symbol $\surd$ shows the termination of a computation.

We explain some of the rules. The rule SR2 states that if the plan a evolves to a'' by performing an action μ, then the non-deterministic choice between a and a' can evolve to a'' by performing μ as well. The rule SR4 states that if a performs an action μ and evolves to a'', then its parallel composition $a \ ||\ a'$ can evolve to $a'' \ ||\ a'$ by performing μ. The rule SR6 states that if a plan a evolves to a'' by performing an action μ, then $a;\ a'$ can first perform a'' and then proceed with a'.

Example 2 Figure 3 shows the structure of the system before and after a reconfiguration. We define the following reconfiguration plan to perform this reconfiguration. In this plan, the connection between the components C and D may be removed first, and then C and D are removed concurrently while the component B is added and then connected to A:

$$Pl = \begin{array}{l} ((del(\mathtt{C}) \ ||\ del(\mathtt{D})) \ +\ (dis(\mathtt{C,D});\ (del(\mathtt{C}) \ ||\ del(\mathtt{D})))) \\ \ ||\ (add(\mathtt{B});\ con(\mathtt{A,B})) \end{array}$$

Note that when a component is removed, all of its connecting edges are removed as well, leaving no dangling edge in the updated structure.

$$(\text{SR1})\frac{}{\alpha \xrightarrow{\alpha} \surd} \quad (\text{SR2})\frac{a \xrightarrow{\mu} a''}{a + a' \xrightarrow{\mu} a''} \quad (\text{SR3})\frac{a \xrightarrow{\mu} \surd}{a + a' \xrightarrow{\mu} \surd} \quad (\text{SR4})\frac{a \xrightarrow{\mu} a''}{a \parallel a' \xrightarrow{\mu} a'' \parallel a'}$$

$$(\text{SR5})\frac{a \xrightarrow{\mu} \surd}{a \parallel a' \xrightarrow{\mu} a'} \quad (\text{SR6})\frac{a \xrightarrow{\mu} a''}{a; a' \xrightarrow{\mu} a''; a'} \quad (\text{SR7})\frac{a \xrightarrow{\mu} \surd}{a; a' \xrightarrow{\mu} a'}$$

$$(\text{SR8})\frac{eval(\phi) \qquad a \xrightarrow{\mu} a'}{\phi :\rightarrow a \xrightarrow{\mu} a'} \quad (\text{SR9})\frac{eval(\phi) \qquad a \xrightarrow{\mu} \surd}{\phi :\rightarrow a \xrightarrow{\mu} \surd}$$

Fig. 4. Operational semantics of reconfiguration algebra

(D) Construction of the system model during an adaptation

Given a reconfiguration plan as an algebraic term, and the behavior of each component in terms of state transition systems, we construct a graph transition system $\mathcal{G}$ that models the system during an adaptation. A graph transition system is essentially a classical transition system augmented with a function mapping states into graphs and transitions into partial graph morphisms:

Definition 3 (*Graph transition system*) Let $\mathbb{G}$ be a universal set of graphs. A graph transition system is a pair $\mathcal{G} = \langle \mathcal{T}, g\rangle$ consisting of a state transition system $\mathcal{T} = \langle S, \Sigma, \rightarrow, \bar{s}_0, F\rangle$ and a pair $g = \langle g_1, g_2\rangle$ where $g_1 : S \rightarrow \mathbb{G}$ is a total function that associates a graph to every state in S, and $g_2(t)$ is an injective partial graph morphism for every transition $t : \bar{s} \xrightarrow{e} \bar{s}'$ of $\rightarrow$.

An accepting run of a graph transition system $\mathcal{G} = \langle \mathcal{T}, g\rangle$ is a sequence $\pi := \langle g_1(\bar{s}_0), e_0\rangle, \langle g_1(\bar{s}_1), e_1\rangle, \ldots, g_1(\bar{s}_n)$ where $\bar{s}_n \in F$, and $(\bar{s}_k, e_k, \bar{s}_{k+1}) \in\rightarrow$ for $0 \leq k < n - 1$. The language of $\mathcal{G}$, denoted by $L(\mathcal{G})$, consists of the set of its accepting runs.

We use a graph transition system $\mathcal{G} = \langle \mathcal{T}, \langle g_1, g_2\rangle\rangle$ to model a system of components c_i, $1 \leq i \leq n$ undergoing a reconfiguration plan Pl. We use a bar variable to show a vector, and s_i to denote the current state of component c_i, regardless of whether or not it is currently an active member of the global system. An active component is a running component that is a part of the system structure. Informally, a state in $\mathcal{G}$ is a pair $\langle \bar{x}, G\rangle \in S \times \mathbb{G}$ where the component vector $\bar{x} \in S$ denotes the computational state of $\mathcal{G}$, and the topology graph $G = g_1(\bar{x})$ represents the connections among the components in the system. The nodes of G represent the components in the system in state $\langle \bar{x}, G\rangle$, and a component c_i can send a message to a component c_j in state $\langle \bar{x}, G\rangle$, only if a directed edge from c_i to c_j exists in G, otherwise, it will be blocked. The significance of G is that it restricts the possibilities of communications among components to only those that are connected in G.

Thus, such a graph transition system, $\mathcal{G}$, specifies both behavior and structural evolution of a system during its reconfiguration phase. Structural modification actions execute interleavingly with behavioral actions of its components. Without loss of generality, we assume all components pre-exist in the system state, regardless of whether or not they are active members of the global system in a given global system state. If a component c_i is inactive in a state $\langle \bar{x}, G\rangle$, x_i shows its state but it should not belong to the G's nodes.

Reconfiguration actions change G and thus, the communication structure of the global system. Changes to the components connectivity graph follow the rules in Fig. 5. These changes result from the execution of the actions given in a reconfiguration plan, specified in the language of our reconfiguration algebra whose semantics appears in Fig. 4. The rules in Fig. 5 define the evolution of G under the execution of a reconfiguration plan, l. The construct $l \xrightarrow{\gamma} l'$ represents the transformation of the reconfiguration plan l into l' as a result of the execution of one of its actions, γ, according to the semantics in Fig. 4. Execution of a reconfiguration action γ changes the component connectivity graph, G, into G' represented by $G \xrightarrow{\gamma} G'$, according to the rules in Fig. 5. The rule $addSE$ transforms G into G' by the execution of an $add(c_i)$ reconfiguration action. Similarly, $delSE$, $conSE$ and $disSE$ each transforms G into G' by the execution of, respectively, a $del(c_i)$, $con(c_i, c_j)$, or $dis(c_i, c_j)$ reconfiguration action.

For $1 \leq i \leq n$, let the state transition system $\mathcal{T}_i = \langle S_i, \Sigma, \rightarrow_i, s_{0,i}, F_i\rangle$ denote the behavior of component c_i, where $\varepsilon \notin S_i$, and let $S_i^{\varepsilon} = S_i \cup \{\varepsilon\}$. Every component c_i behaves and changes states according to its own respective state transition system $\mathcal{T}_i$, regardless of whether or not it is an active member of the global system.

$$(addSE)\ \frac{l \xrightarrow{add(c_i)} l', G = (V, E, \sigma, \tau, lab), V' = V \cup \{c_i\}, G' = (V', E, \sigma, \tau, lab)}{G \xrightarrow{add(c_i)} G'}$$

$$(delSE)\ \frac{\begin{array}{c} l \xrightarrow{del(c_i)} l', G = (V, E, \sigma, \tau, lab), R = \{e \mid e \in E, \sigma(e) = c_i \vee \tau(e) = c_i\}, \\ V' = V \setminus \{c_i\}, E' = E \setminus R, G' = (V', E', \sigma, \tau, lab) \end{array}}{G \xrightarrow{del(c_i)} G'}$$

$$(conSE)\ \frac{\begin{array}{c} l \xrightarrow{con(c_i,c_j)} l', G = (V, E, \sigma, \tau, lab), c_i \in V, c_j \in V, \\ E' = E \cup \{(c_i, c_j)\}, G' = (V, E', \sigma, \tau, lab) \end{array}}{G \xrightarrow{con(c_i,c_j)} G'}$$

$$(disSE)\ \frac{\begin{array}{c} l \xrightarrow{dis(c_i,c_j)} l', G = (V, E, \sigma, \tau, lab), c_i \in V, c_j \in V, \\ E' = E \setminus \{(c_i, c_j)\}, G' = (V, E', \sigma, \tau, lab) \end{array}}{G \xrightarrow{dis(c_i,c_j)} G'}$$

Fig. 5. Structure evolution rules

$$\texttt{(ADD)}\ \frac{\begin{array}{c} \bar{x} = \langle x_1, ...x_{i-1}, \varepsilon, x_{i+1}, ...x_n\rangle, G \xrightarrow{add(c_i)} G', \\ s_i \in \texttt{init}_i(x), s_i \neq \varepsilon, \bar{x}' = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_n\rangle \end{array}}{\langle \bar{x}, G\rangle \xrightarrow{add(c_i)} \langle \bar{x}', G'\rangle}$$

$$\texttt{(REMOVE)}\ \frac{\begin{array}{c} \bar{x} = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_n\rangle, G \xrightarrow{del(c_i)} G', \\ s_i \in \texttt{removable}_i(x), s_i \neq \varepsilon, \bar{x}' = \langle x_1, ...x_{i-1}, \varepsilon, x_{i+1}, ...x_n\rangle \end{array}}{\langle \bar{x}, G\rangle \xrightarrow{del(c_i)} \langle \bar{x}', G'\rangle}$$

$$\texttt{(CONT)}\ \frac{\bar{x} = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_{j-1}, s_j, x_{j+1}, ...x_n\rangle, s_i \neq \varepsilon, s_j \neq \varepsilon, G \xrightarrow{con(c_i,c_j)} G'}{\langle \bar{x}, G\rangle \xrightarrow{con(c_i,c_j)} \langle \bar{x}, G'\rangle}$$

$$\texttt{(DISCONT)}\ \frac{\bar{x} = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_{j-1}, s_j, x_{j+1}, ...x_n\rangle, s_i \neq \varepsilon, s_j \neq \varepsilon, G \xrightarrow{dis(c_i,c_j)} G'}{\langle \bar{x}, G\rangle \xrightarrow{dis(c_i,c_j)} \langle \bar{x}, G'\rangle}$$

$$\texttt{(IACT)}\ \frac{\bar{x} = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_n\rangle, s_i \xrightarrow{\alpha}_i s_i', s_i \neq \varepsilon, s_i' \neq \varepsilon, \bar{x}' = \langle x_1, ...x_{i-1}, s_i', x_{i+1}, ...x_n\rangle}{\langle \bar{x}, G\rangle \xrightarrow{\alpha} \langle \bar{x}', G\rangle}$$

$$\texttt{(SACT)}\ \frac{\begin{array}{c} \bar{x} = \langle x_1, ...x_{i-1}, s_i, x_{i+1}, ...x_{j-1}, s_j, x_{j+1}, ...x_n\rangle, G = (V, E, \sigma, \tau, lab), (c_i, c_j) \in E, \\ s_i \xrightarrow{\alpha!}_i s_i', s_i \neq \varepsilon, s_i' \neq \varepsilon, s_j \xrightarrow{\alpha?}_j s_j', s_j \neq \varepsilon, s_j' \neq \varepsilon, \\ \bar{x}' = \langle x_1, ...x_{i-1}, s_i', x_{i+1}, ...x_{j-1}, s_j', x_{j+1}, ...x_n\rangle, i \neq j, \psi_{i,j}(\bar{x}, \alpha) \end{array}}{\langle \bar{x}, G\rangle \xrightarrow{\alpha} \langle \bar{x}', G\rangle}$$

Fig. 6. Operational semantics of reconfiguration actions

A component vector $\bar{x} \in \mathcal{X} = \prod_{i=1}^{n} S_i^{\varepsilon}$ denotes the current state of every c_i that is an active member of the the global system in global system state $\langle \bar{x}, G \rangle$. For $\bar{x} = \langle x_1, x_2, \ldots, x_n \rangle \in \mathcal{X}$, a component c_i is an active member of the global system in state $\langle \bar{x}, G \rangle$ if and only if $x_i \neq \varepsilon$, in which case $x_i = s_i \in S_i$ is the current state of c_i in the system state $\langle \bar{x}, G \rangle$. Similarly, $x_i = \varepsilon$ if and only if c_i is not an active member of the global system in state $\langle \bar{x}, G \rangle$.

Definition 4 (*Global transition system*) Let $\mathcal{G} = \langle \mathcal{T}, \langle g_1, g_2 \rangle \rangle$ be a graph transition system representing a system of components $c_i, 1 \leq i \leq n$, whose connections are modeled by a graph $G = (V, E, \sigma, \tau, lab)$, where $\mathcal{T} = \langle S, \Sigma, \rightarrow, s_0, F \rangle$. The set of states of $\mathcal{G}$ is the set of all $\langle \bar{x}, G \rangle$ where:

1. $\bar{x} \in \mathcal{X} \wedge G = g_1(\bar{x})$, and
2. for all $1 \leq i \leq n$, $x_i \neq \varepsilon \iff c_i \in V$.

The function g_2 returns a partial injective morphism $\psi = (\psi_V, \psi_E) : G \rightarrow G'$ implied by the rules in Fig. 5. □

Condition 2 in Definition 4 ensures that a component c_i is active in a global system state $\langle \bar{x}, G \rangle$ if and only if c_i belongs to the system structure in $\langle \bar{x}, G \rangle$, as determined by graph G. The rules in Fig. 6 define how the components connection graph changes as a result of the execution of actions in the reconfiguration plan.

The reconfiguration of the global system according to a reconfiguration plan, *Pl*, starts with $\mathcal{G}$ in an initial state $\langle \bar{x}_0, G_0 \rangle$ where $\bar{x}_0 = \langle x_{1,0}, \ldots, x_{n,0} \rangle$, $x_{i,0}$ denotes the status of component c_i at the start of the reconfiguration, and $G_0 = g_1(\bar{x}_0)$ denotes the initial structure of the components in the system before reconfiguration. The global state $\langle \bar{x}, G \rangle$ of the system changes either when (1) a component performs an internal computation action; (2) a pair of components perform a pair of complementary communication actions compatible with the topology of component connections in G; or (3) execution of the next reconfiguration action prescribed in *Pl* changes the system structure. The first two activities change the x-part and the third changes the G-part of the global system state (in addition to possibly changing its x-part, as well).

The rule `ADD` in Fig. 6 "adds" a pre-existing component c_i as a new active member to the system. The entry ε in position i in x ensures that c_i is currently not an active member of the system. Execution of the $add(c_i)$ reconfiguration action evolves the component connections graph from G to G' according to the rule *addSE* in Fig. 5. When a component c_i is added to the system, it must start from a state compatible with the current system state $\bar{x}$. The definition of *state compatibility* is application-specific and we assume that the verification engineer defines a function that determines the compatible states from which c_i can start in $\bar{x}$. If no such compatible state exists, it means that the current state is not a proper state to add the component. Furthermore, not all compatible states are safe to start c_i in $\bar{x}$ because it may lead to violating some safety properties or cause a deadlock, in which case the compatible state will be removed by the synthesized controller. Thus, to ensure a safe reconfiguration, c_i may start only from a subset of its compatible states. The user-defined function $\texttt{init}_i(\bar{x})$ returns the set of states of c_i that are compatible with c_i joining the system, given the membership status and the actual states of all system components, as specified by $\bar{x}$. If the current state of c_i, denoted by s_i, is in the set returned by $\texttt{init}_i(\bar{x})$, then adding c_i in its current state of s_i is actually compatible with the current states of the rest of the components. The new global state, therefore, replaces ε with s_i.

The component c_i is removed using the rule `REMOVE` in Fig. 6. The entry s_i in position i in $\bar{x}$ ensures that c_i is currently an active member of the system (by definition, $s_i \neq \varepsilon$). Execution of the $del(c_i)$ reconfiguration action evolves the component connections graph from G to G' according to the rule *delSE* in Fig. 5. The function $\texttt{removable}_i(\bar{x})$ returns the set of states of c_i that are compatible with c_i leaving the system, given the membership status and the actual states of all system components, as specified by $\bar{x}$. If the current state of c_i, denoted by s_i, is in the set returned by $\texttt{removable}_i(\bar{x})$, then removing c_i in its current state of s_i is actually compatible with the current states of the rest of the components. The new global state, therefore, replaces s_i with ε.

The `CONT` rule in Fig. 6 connects active components c_i and c_j, and the `DISCONT` rule disconnects them. The entries s_i and s_j (that, by definition, are different from ε) in $\bar{x}$ ensure that c_i and c_j are currently active members of the system.

When an internal action is executed by an active component c_i, the system evolves to a new state where the state of c_i is updated (rule `IACT`). If two active connected components c_i and c_j synchronize on action α, the system state is updated with the new states of c_i and c_j (rule `SACT`). The condition $\psi_{i,j}(\bar{x}, \alpha)$ checks if c_i can synchronize with c_j on α in state $\bar{x}$ and ensures that the system model remains deterministic. The RW framework is developed for deterministic systems and the determinisim of a constructed model is ensured by the condition $\psi_{i,j}(\bar{x}, \alpha)$. If the determinism condition does not hold, the transition cannot be taken and the user will be notified about this. The above rules express the case of $i < j$. The rules for the case of $j < i$ are defined similarly.

As, according to the rules in Fig. 4, execution of a reconfiguration plan eventually reduces it to $\surd$ for which no further transformation is possible, the execution of a reconfiguration plan eventually terminates. Termination of the execution of a reconfiguration plan, Pl, places $\mathcal{G}$ in a state $\langle \bar{x}, G\rangle$, $\bar{x} \in X_{\surd}$, $G \in G_{\surd}$ that is reachable from the initial, pre-reconfiguration state $\langle \bar{x}_0, G_0\rangle$. Observe that at this point $\mathcal{G}$ can no longer structurally evolve, however, some of its components may still be active, in which case the global state of the system may still evolve according to the `(IACT)` and `(SACT)` rules of Fig. 6. Some of these reachable states may have undesirable properties, for instance, the ones that lead to deadlock after an adaptation. Let $\langle \bar{x}, G\rangle$, $\bar{x} \in X_d$, $G \in G_d$ be the set of states of $\mathcal{G}$ that have no undesirable properties. We define the final states of $\mathcal{G}$, then, as a subset of its states reachable from $\langle \bar{x}_0, G_0\rangle$ by the reconfiguration plan Pl that have no undesired properties, i.e., $\langle \bar{x}, G\rangle$, $\bar{x} \in X_d \cap X_{\surd}$, $G \in G_d \cap G_{\surd}$ constitutes the final states of $\mathcal{G}$.

Lemma 1 *If $\langle \bar{x}, G\rangle$ is a state of $\mathcal{G}$ by Definition 4, then in Fig. 6, $\langle \bar{x}', G'\rangle$ derived in rules `ADD` and `REMOVE`, $\langle \bar{x}, G'\rangle$ derived in rules `CONT` and `DISCONT`, and $\langle \bar{x}', G\rangle$ derived in rules `IACT` and `SACT`, are also states of $\mathcal{G}$ by Definition 4.*

Proof From $x'_k \in S^{\varepsilon}_k$, $1 \le k \le n$ and the definition of g_1, item (1) is simply concluded. The rules `IACT` and `SACT` do not change the system structure, i.e. for $1 \le k \le n$,

$$G' \models \phi_k \Leftrightarrow G \models \phi_k. \tag{1}$$

1. If $k \neq i, j$ then $x'_k = x_k$ and $x'_k \neq \varepsilon \Leftrightarrow G' \models \phi_k$ is followed from (1) and the assumption $x_k \neq \varepsilon \Leftrightarrow G \models \phi_k$.
2. If $k \in \{i, j\}$, then $x_k, x'_k \in S_k$ and $x_k, x'_k \neq \varepsilon$. Since $x_k \neq \varepsilon$ then $G \models \phi_k$, and from $x'_k \neq \varepsilon$ and (1), we can conclude $x'_k \neq \varepsilon \Leftrightarrow G' \models \phi_k$.

Similarly, the rules `CONT` and `DISCONT` do not add or remove nodes, i.e., the formula (1) holds for $1 \le k \le n$. Since $x'_k = x_k$, then $x'_k \neq \varepsilon \Leftrightarrow G' \models \phi_k$ is followed consequently.

Since $\langle \bar{x}, G\rangle$ is a global system state, then $G \models \phi_i$ does not hold and the rule `ADD` adds the new component c_i by applying the rule $addSE$, i.e. $V' = V \cup \{c_i\}$ and $G' \models \phi_i$ holds. Furthermore, for $1 \le k \le n$, $k \neq i$, $x'_k = x_k$ and the formula (1) holds. If $k \neq i$, we can prove $x'_k \neq \varepsilon \Leftrightarrow G' \models \phi_k$ similar to the case 1 of the rules `CONT` and `DISCONT`. If $k = i$, then $x' = s_i \in S_i$ and $s_i \neq \varepsilon$, and subsequently, item (2) is concluded from $G' \models \phi_i$. Similarly, we can prove item (2) for the rule `Remove`. □

Example 3 Consider $del(C)\ add(B)\ conn(A, B)\ del(D)$ as one of the possible reconfiguration paths in our example. Figure 7 shows the behavior of the system during this reconfiguration. For the sake of simplicity and readability, we do not draw the graphs associated with each state in this figure, but we partition the system behavior into several parts (shown as the boxes in the figure), each of which shows the behavior of the system in the corresponding configuration obtained using the rules `IACT` and `SACT` in Fig. 6. In other words, the states in each box have the same structure and the transition system in a lined box describes the behavior of the system between structural actions. The corresponding structure of each box, showing the changes of the system structure, is depicted in Fig. 8. The label of a box in Fig. 7 (e.g., p0, p1, ..., p4) represents its corresponding configuration label in Fig. 8.

The label of a system state is the concatenation of the state labels of each component followed by the label of the current configuration. For example, the state a4c3d0b2p1 shows that the component A is in the state a4, C is in the ε-state c3, D is in the state d0, B is in the ε-state b2, and the system structure is the configuration $p1$. The dashed arrows show the structural events (`add`, `del`, `con`, and `dis`) connecting the states of two configurations, and solid arrows are the behavioral events done by the components in a configuration. The gray states are bad states that we will introduce and consider in Sect. 5. All states of the final configuration comprise the final states.

4. Property specification

We specify properties to be preserved during the adaptation of a system in terms of the structure of the system in addition to its behavior. We use an automaton called *graph automaton* over an alphabet of both ordinary event variables and atomic graph constraints to specify a property. A graph constraint is defined as follows:

Definition 5 (*Graph constraints*) An atomic graph constraint is a graph morphism. A graph constraint is a Boolean formula over atomic graph constraints:

1. $True$, $False$, and every atomic graph constraint are graph constraints;
2. if c_1 and c_2 are two graph constraints, then $c_1 \vee c_2$, $c_1 \wedge c_2$ and $\neg c_1$ are graph constraints;
3. if c is a graph constraint, then (c) is a graph constraint.

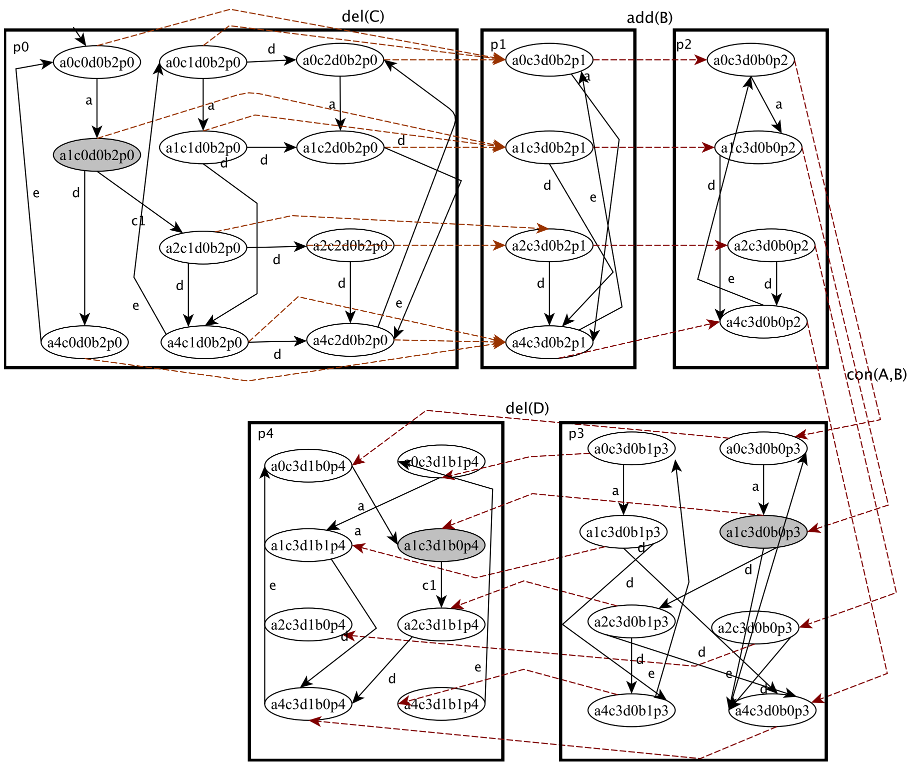


Fig. 7. The system behavior during the reconfiguration

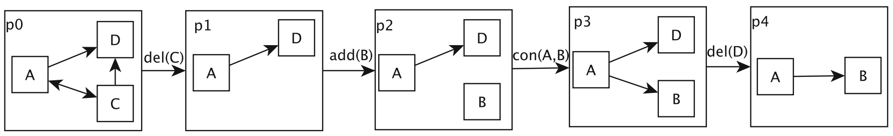


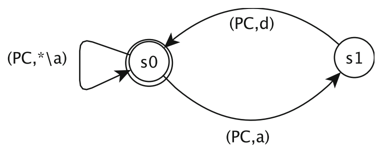
Fig. 8. The system structure during the reconfiguration

Definition 6 (*Graph constraint satisfaction*) A graph G satisfies an atomic graph constraint $\alpha : P \rightarrow C$ if for every injective morphism $p : P \rightarrow G$, there exists an injective morphism $q : C \rightarrow G$ with $q \circ \alpha = p$.
A graph G satisfies a graph constraint of the form:

1. $c_1 \vee c_2$, if G satisfies either c_1 or c_2, or both;
2. $c_1 \wedge c_2$, if G satisfies both c_1 and c_2;
3. $\neg c$, if G does not satisfy c;
4. (c), if G satisfies c.

Definition 7 A graph automaton is defined as $\mathcal{A} = (Q, \Sigma, GC, \delta, q_0, Q_m)$ where Q is a finite set of states, Σ is a finite set of alphabet symbols, GC is a set of graph constraints as defined in Definition 5, $\delta : Q \times (GC \times \Sigma) \rightarrow Q$ is a partial transition function, q_0 is the initial state, and $Q_m \subseteq Q$ is the set of marked states.

In general, an automaton runs on a given sequence of inputs and its language is defined as the set of observables of all of its runs. The observable of a run is the sequence of the observables of its transition instances, and an observable of a transition instance is a witness that makes its label satisfiable. In a graph automaton, a witness of a transition $\langle X, G\rangle \xrightarrow{(\phi, e)} \langle X', G'\rangle$ is a pair (G, e) where $G \models \phi$ (i.e., G satisfies the graph constraint ϕ), and



$$PC = \emptyset \rightarrow \boxed{A} \rightarrow \boxed{B} \;\vee\; \emptyset \rightarrow \boxed{A} \rightarrow \boxed{C} \;\vee\; \emptyset \rightarrow \boxed{A} \rightarrow \boxed{D}$$

Fig. 9. A property

$e \in \Sigma$ is an event. Thus, the observable of a run of a graph automaton $\mathcal{A}$ is a sequence of graph/event pairs $\pi \in (\mathbb{G} \times \Sigma)^*$, where for some set of X_i's and e_i's, its i-th element is $\pi_i = (G_i, e_i)$; $q_0 = \langle \bar{x}_0, G_0 \rangle$ is the initial, pre-reconfiguration state of the automaton; $\delta(\langle X_i, G_i \rangle, (\phi, e_i)) = \langle X_{i+1}, G_{i+1} \rangle$; and $G_i \models \phi_i$. The language $L(\mathcal{A})$ of a graph automaton $\mathcal{A}$ is the set of all such sequences π. The marked language of a graph automaton is defined similarly to that of an ordinary automaton (See Sect. 2.2).

The product of two graph automata is defined as follows:

Definition 8 The product of two graph automata $\mathcal{A}_1 = (Q_1, \Sigma, GC, \delta_1, q_{0,1}, Q_{m,1})$ and $\mathcal{A}_2 = (Q_2, \Sigma, GC, \delta_2, q_{0,2}, Q_{m,2})$, denoted by $\mathcal{A}_1 \times \mathcal{A}_2$, is a graph automaton $\mathcal{A} = (Q, \Sigma, GC, \delta, q_0, Q_m)$ where $Q = Q_1 \times Q_2$, $q_0 = (q_{0,1}, q_{0,2})$, $Q_m = Q_{m,1} \times Q_{m,2}$, and $((s_1, s_2), (t_1 \wedge t_2, a), (s'_1, s'_2)) \in \delta$ where $(s_1, (t_1, a), s'_1) \in \delta_1$ and $(s_2, (t_2, a), s'_2) \in \delta_2$.

Lemma 2 $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$.

Proof Induction on the length of the accepting words. □

Example 4 The property stating that the component A must always remain connected to another component, and an interaction d must always happen after an occurrence of a is shown in Fig. 9, where PC is a graph constraint stating that A is connected to C, or D, or B, $\emptyset$ shows an empty graph, and $\emptyset \rightarrow G$ is a basic graph constraint (See Definition 5).

5. Reconfiguration controller synthesis

We need to make sure that a reconfiguration plan is safe, i.e., it does not cause a deadlock during the reconfiguration process, nor does it violate any user-defined correctness properties. Control consists in imposing interleaving of structural actions and behavioral actions in such a way that they are complying with the required constraints, e.g., structural actions can wait for behavioral actions to be in a state ready for reconfiguration, and reciprocally behavioral actions can be inhibited or delayed until the adaptation reaches a state where they can be executed correctly. As the third step of our approach, in this section we deal with the design of safe structural adaptation plans for adaptive systems. We build a non-blocking controller to control the reconfiguration process such that a specific property remains satisfied. As mentioned before, a property can be specified in terms of the system structure as well as its behavior. First, we deal with the synthesis of controllers for enforcing properties specified in terms of both structure and behavior. Then to reduce the complexity of computing the controller, we improve the approach for the case of pure structural properties.

5.1. General synthesis method

The aim of the supervisory control problem is to design a controller that restricts the system behavior by allowing/disallowing events in a proper way such that a specific property is preserved. We approach our goal as a controller synthesis problem. Given a reconfiguration plan Pl, we generate all possible strategies for reconfiguring a structure, and build a system model containing all possible reconfiguration strategies. Then, we restrict the reconfiguration/component operations to preserve the desired property in question. The synthesized controller allows behavioral and structural actions be interleaved in such a way that they can progress as far as possible without jeopardizing safety requirements. Inhibiting events corresponds to a control that is delaying some actions in order to execute them only in states where they can safely be executed. Delay is to be taken in the sense that transitions with these actions are inhibited or removed except from appropriate states. This delaying is a reciprocal game between structural and behavioral parts of the system, which have to wait for each other in order to avoid undesirable behavior, e.g., attempting communication when the other party is not ready yet.

In our problem domain, a controllable event in the system can be a reconfiguration event described in an adaptation strategy, or a visible event describing an internal computation or an interaction of the components. It is worth mentioning that many components (such as web-services) are presented as black-boxes, and it is impossible to observe and modify their internal behavior. However, one can usually observe and coordinate the interactions of components. Uncontrollable events can be invisible internal events of components, as well as unplanned reconfiguration events, e.g., break-down of a component or a fault in the connections among components. We assume that a controller has full observation, i.e. all events are observable by the controller. We do not deal with partial observations in this paper, and if an event is invisible for the controller in the implementation, one can consider it as an uncontrollable event.

In contrast to the Ramadge–Wonham framework, we do not use ordinary automata to model a system and a specification. We adapt the automata theoretic approach of the RW framework for designing a controller to control the reconfiguration phase.

We first compute the product of a system model specified as a graph transition system and the graph automaton of a specification. We prove that the product returns those paths of the system model that are consistent with the specification. Then, we follow the RW framework approach to obtain the final non-blocking controller. Synthesis of a controller for enforcing a property during a reconfiguration phase consists of three main steps as the classical RW framework. The product that we use in the first step is specific to our problem, the second step is the same as in the RW framework, and we use our own algorithm in the third step. This algorithm is based on the ordinary state space exploration algorithms.

Step A—Computing the synchronous product of the system model and the graph automaton of the property

Given the graph transition system of a reconfiguration, we compute the product of the graph transition system and the graph automaton of the property as follows:

Definition 9 Let $\mathcal{G} = \langle \mathcal{T}, g\rangle$, $\mathcal{T} = \langle S, \Sigma, \rightarrow, \bar{s}_0, F\rangle$ be a graph transition system indicating the evolution of a system structure during a dynamic reconfiguration, and $\mathcal{A} = (Q, \Sigma, GC, \delta, q_0, Q_m)$, denote the graph automaton of the property to be enforced. The product of $\mathcal{G}$ and $\mathcal{A}$, written as $\mathcal{G} \bowtie \mathcal{A}$, is a graph transition system $\mathcal{AG} = \langle \mathcal{T}', g'\rangle$ where $g' = \langle g_1', g_2'\rangle$ and $\mathcal{T}' = \langle S', \Sigma, \rightarrow', \bar{s}_0', F'\rangle$ is defined as follows:

- $S' = S \times Q$,
- $\bar{s}_0' = (\bar{s}_0, q_0)$,
- $F' = F \times Q_m$,
- $g_1'((\bar{s}, q)) = g_1(\bar{s})$ for all $(\bar{s}, q) \in S'$,
- $g_2'(((\bar{s}, q), a, (\bar{s}', q'))) = g_2((\bar{s}, a, \bar{s}'))$ for all $((\bar{s}, q), a, (\bar{s}', q')) \in\rightarrow'$, and

$$\frac{(q, (t, a), q') \in \delta \quad (\bar{s}, a, \bar{s}') \in\rightarrow \quad g_1(\bar{s}) \models t}{((\bar{s}, q), a, (\bar{s}', q')) \in\rightarrow'}.$$

The goal of synchronous product is to consider all accepting runs of $\mathcal{G}$ that are also the accepting runs of $\mathcal{A}$. An accepting run on $\mathcal{G}$ which matches a run in $\mathcal{A}$ describes an execution sequence satisfying the property expressed by $\mathcal{A}$.

Theorem 1 *For any graph transition system $\mathcal{G}$ and a graph automaton $\mathcal{A}$, $L(\mathcal{G} \bowtie \mathcal{A}) = L(\mathcal{G}) \cap L(\mathcal{A})$*

Proof In order to prove this theorem, we transform the system model $\mathcal{G} = \langle \mathcal{T}, g\rangle$, $\mathcal{T} = \langle S, \Sigma, \rightarrow, s_0, F\rangle$ into an automaton defined as $\Psi(\mathcal{G}) = \langle S, \Sigma, \delta, s_0, F\rangle$ where $(s, (t, a), s') \in \delta$ if $(s, a, s') \in\rightarrow$ and the graph constraint t is $\emptyset \rightarrow g_1(s)$. Note that $\emptyset$ represents an empty graph. In this automaton, we move the graph label of a state to all its outgoing transitions. Observe that the label of the states with no outgoing transitions is not considered in the product. It is trivial to show that the product of $\mathcal{G}$ and the property automaton equals to the ordinary product of the transformation of $\mathcal{G}$ and the property automaton (by induction on the length of the accepting words), i.e. $\mathcal{G} \bowtie \mathcal{A} = \Psi(\mathcal{G}) \times \mathcal{A}$. It follows from Lemma 2 that $L(\Psi(\mathcal{G}) \times \mathcal{A}) = L(\Psi(\mathcal{G})) \cap L(\mathcal{A})$, and subsequently $L(\mathcal{G} \bowtie \mathcal{A}) = L(\mathcal{G}) \cap L(\mathcal{A})$. □

Step B—Removing bad states from $\mathcal{AG}$

The product of a graph transition system $\mathcal{G}$ modeling the system behavior, and a graph automaton $\mathcal{A}$ modeling the desired property, is a controller that restricts atomic reconfigurations to those paths which preserve the property A during the adaptation phase by disallowing some events. However, not all events are controllable, and for some technical reasons, they may not be preventable either. For instance, only the interactions among the components, but not the internal behavior of a component may be observable and controllable by the controller. However, the controller $\mathcal{AG}$ prevents all forbidden uncontrollable events from occurring while it is impossible to prevent undesirable uncontrollable events. To rectify this situation, we compare $\mathcal{G}$ and $\mathcal{AG}$ to find all bad states of $\mathcal{AG}$ where a forbidden uncontrollable reconfiguration occurs, according to the RW framework.

Let $\Sigma \supseteq \Sigma_r = \Sigma_r^u \cup \Sigma_r^c$ denote the set of all actions applied during a reconfiguration phase where Σ_r^c and Σ_r^u are the sets of controllable events and uncontrollable events, respectively. A state such as (s, q) of $\mathcal{G} \bowtie \mathcal{A}$ is considered a bad state if it fails to satisfy the following condition:

$$act_{\mathcal{G}}(s) \cap \Sigma_r^u \subseteq act_{\mathcal{AG}}((s, q)) \tag{2}$$

where $act_{\mathcal{G}}(s)$ and $act_{\mathcal{AG}}((s, q))$ denote all activated events of $\mathcal{G}$ and $\mathcal{AG}$, in states s and (s, q) respectively, i.e., $act_{\mathcal{G}}(s) = \{\sigma \in \Sigma \mid \exists s' \in S_1 , (s, \sigma, s') \in\rightarrow\}$. In other words, a state is good when uncontrollable actions from the original system are still included in actions of the controlled system, i.e. they have not been abusively inhibited. Having all bad states of $\mathcal{AG}$, we remove bad states and the transitions to and from them from $\mathcal{AG}$. We use a depth-first algorithm to traverse the state space and remove the bad states.

Step C—Removing non-coaccessible states

The resulting graph transition system of Step B may include non-coaccessible states, i.e. states that are either not reachable from an initial state or from which a final state is not reachable.

Moreover, the system can lead to a *final deadlock* state after adaptation, i.e. the adaptation is performed successfully, but the system after adaptation contains deadlock states. Therefore, the deadlock states must be avoided. We trim this graph transition system to those states that lie on a path from an initial state to a final state. The resulting graph transition system is a non-blocking controller that enforces the desired property during the reconfiguration phase.

Algorithm 1 is our algorithm to remove the non-coaccessible states. It takes an input graph transition system $\mathcal{AG}$, and returns a graph transition system (*res*). This algorithm, first returns in *rgts*, the part of $\mathcal{AG}$ that is accessible from its initial state using a depth first search algorithm (the function compute_reachable_GTS_from). Now we must trim *rgts* to those states from which a final state can be reached. So, for each final state s in *rgts*, the algorithm computes the part of *rgts* from which s can be reached. The variable *res* shows a subset of coaccessible states/transitions computed so far. We use a (backward) depth first algorithm to compute the states from which a final state can be reached. This search starts from a final state, traverses the incoming transitions, adds the visited source states/transitions to the result in a recursive manner until it reaches a coaccessible state, i.e. a state of *res*

Theorem 2 *Algorithm 1 is sound.*

Proof Let $s \overset{*}{\rightarrow} s'$ denote that state s' is reachable from s, i.e. $\overset{*}{\rightarrow}$ is the transitive closure of $\rightarrow$. To show the correctness, we need to show that for all states s_k in *res*, (i) $s_0 \overset{*}{\rightarrow} s_k$, and (ii) there exists a state s_n in the final states of *res* such that $s_k \overset{*}{\rightarrow} s_n$. The graph transition system *rgts* contains all the states that are reachable from the initial state. Since the state/transition set of *res* is a subset of *rgts*'s states/transitions, we conclude the states of *res* are reachable from the initial state [obligation (i)]. The obligation (ii) is proved by induction on the number of final states of *rgts*. □

There is no unique solution to the basic controller synthesis problem, i.e. there may be several controllers that can enforce a property. A maximal solution is a controller $\mathcal{AG}$ that restricts the behavior of the system the least.

Definition 10 Given a system model $\mathcal{G}$ and a property $\mathcal{A}$, a non-blocking controller, $\mathcal{AG}$, enforcing $\mathcal{A}$ in $\mathcal{G}$ is maximally permissive if for any other non-blocking controller $\mathcal{AG}'$ that enforces $\mathcal{A}$ on $\mathcal{G}$, $L(\mathcal{AG}') \subseteq L(\mathcal{AG})$.

Algorithm 1: Computing Coaccessible Automaton

```
    Input: Graph Transition System $\mathcal{AG}$.
    Output: res (Trimmed Graph Transition System $\mathcal{AG}'$).
    // $s_0$ is the initial state of $\mathcal{AG}$
1   res = ∅;
2   stack = ∅;
    // compute the reachable part of $\mathcal{AG}$ from $s_0$
3   rgts = compute_reachable_GTS_from($s_0$, $\mathcal{AG}$);
4   for each final state $s$ in rgts do
5       if $s \notin$ res.states then
6           stack.push($s$);
7           visited=∅;
8           repeat
9               curstate = stack.pop();
10              if curstate $\notin$ res.states and curstate $\notin$ visited then
                    // gets all transitions of rgts with the target state of "curstate"
11                  intrans = IncomingTransof(curstate,rgts);
12                  for each $t \in$ intrans do
13                      res.states = $\{t$.trg$\}$ ∪ res.states
14                      res.transitions = $\{t\}$ ∪ res.transitions
15                      stack.push($t$.src);
16                  end
17                  visited = visited ∪ {curstate};
18              end
19          until stack $\neq$ ∅;
20      end
21  end
```

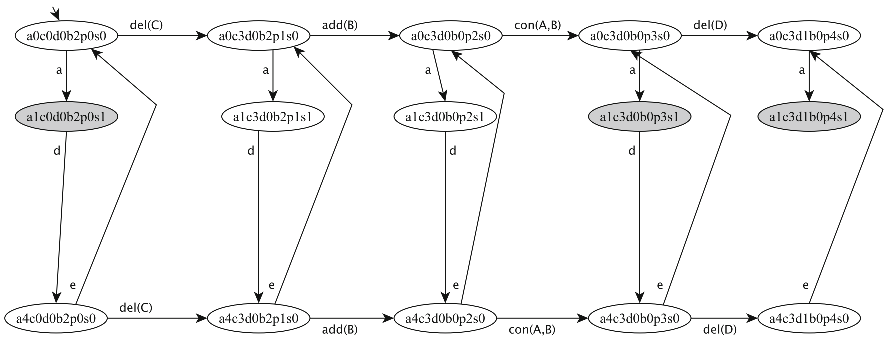


Fig. 10. The product

Theorem 3 *A controller $\mathcal{AG}'$ synthesized using our approach is the maximally-permissive controller.*

Proof A graph transition system can be transformed into an equivalent automaton as explained above. Then, we can prove this theorem similarly to the corresponding theorem of the RW framework [RW87]. □

Example 5 An excerpt of the product of the system behavior and the property automaton of our running example is shown in Fig. 10, if the adaptation is initiated in the system state a0c0d0b2p0 (Fig. 7). If all events are controllable, then the product is the final controller (or the controlled system).

Now suppose a case where the event c1 is uncontrollable, while the events a, d, e and c2 are controllable, i.e., the controller monitors internal behavior of the components and their interactions, except the interactions on c1. In this case, the states in {a1c0d0b2p0s1,a1c3d0b0p3s1,a1c3d1b0p4s1} are bad, because in the product, the event c1 in the corresponding states of the plant model (i.e., the state set {a1c0d0b2p0,a1c3d0b0p3,a1c3d1b0p4})

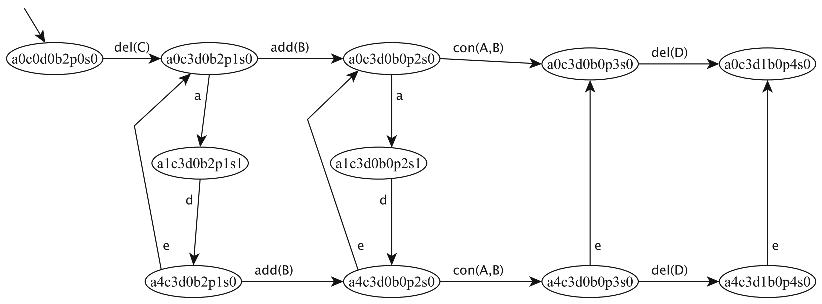


Fig. 11. The controller

is forbidden while this event is not controllable and cannot be disallowed. After removing the bad states, the state a4c0d0b2p0s0 becomes non-coaccessible, because it is not reachable from the initial state a0c0d0b2p0s0 and is removed from the product. Figure 11 shows the controller after removing the bad and non-coaccessible states.

While a synthesized controller is nonblocking during the adaptation, it is not guaranteed that the system has no deadlock state after an adaptation as well. The reason is that we defined the termination of the adaptation phase as execution of the last structural event. As such, we considered all the states of the final mode reached by a structural event as a valid final state. To prevent such blocking, we need to define the termination of an adaptation more carefully. Note that the state a0c3d1b0p4 is not a blocking state, because after the adaptation, the system does not run under the control of the controller anymore, but continues its normal execution. From this state, the system can perform a and evolve to the state a1c3d1b0p4.

A possible solution to this problem is to find the deadlock states of the system after an adaptation, remove them from the set of final states of the system and perform Step C for the updated model.

Complexity The complexity of the approach is $O(m \times n)$ where m and n are respectively, the sizes of the property and that of the model, i.e., the numbers of transitions. The complexity of computing the product is $O(m \times n)$. Step B traverses the whole state space of the product, i.e. the complexity of this step is $O(m \times n)$. To obtain the accessible states from the initial state, in the worst case, we traverse the whole state space. Similarly, computing the states from which a final state is reachable also needs $m \times n$ steps in the worst case. Note that for each final state, we do not traverse the whole state space to reach the initial state, but we update the list of coaccessible states and the traversing process stops when a coaccessible state is reached. Therefore, a transition/state is visited once, and the complexity of Step C is $O(m \times n)$.

5.2. Synthesis for pure structural properties

To reduce the complexity of synthesis in the special case of preserving a pure structural property, we construct an abstract graph transition system which specifies only the structural changes of the system. In other words, we don't use the rules `IACT` and `SACT` in Fig. 6 to build this transition system and use a ghost state s to denote a (behavioral) state. Given a pure structural property and the abstract graph transition system, we obtain an abstract controller $\mathcal{AG}$ using the method introduced in Sect. 5.1. This controller only restricts the reconfiguration actions such that the property is preserved; however, it does not ensure termination of the adaptation phase. This is because of the possible existence of deadlocks in the system when it runs with a fixed structure. For instance, assume a case where while a component is waiting to receive a message from another component, the sender is removed which may cause a deadlock in the system. In other words, since we do not model the behavior of the components, it is impossible to detect a (behavioral) deadlock using the abstract model. Hence, we need to model the behavior of the system as well to be able to detect deadlocks.

The abstract synthesized controller, $\mathcal{AG} = \langle \mathcal{T}, g \rangle$, specifies the potential safe reconfiguration strategies to adapt the system behavior. To remove non-coaccessible states, we use $\mathcal{T}$ as the input reconfiguration plan at the semantics level and construct the system model, as explained in Sect. 3. Afterwards, we proceed with step C to remove non-coaccessible states of the constructed model. The result is a non-blocking controller enforcing the structural property.

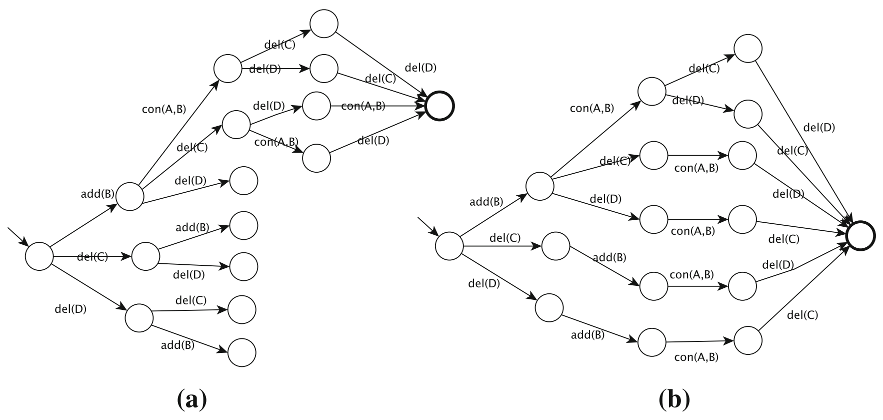


Fig. 12. The pure graph transition system of the example 5.2

In other words, in the extended approach for synthesis of pure structural properties, we do not consider the internal behavior of the components in constructing the whole system model. Instead, we build a simple model describing only the structural evolution and synthesize a controller based on this model. Afterwards, we add the behavior of components to the synthesized controller and refine it. While in the approach described in Sect. 5.1, we build a complex model describing both the system behavior and its structural evolution, synthesize a controller based on that model and refine the synthesized controller.

Example 6 In our running example, assume the property to be enforced is that at all time the component A should be connected to another component. The pure graph transition of our reconfiguration plan is partially shown in Fig. 12a, and the abstract synthesized controller to enforce the reconfiguration actions is shown in Fig. 12b. The reconfiguration plan corresponds to the abstract synthesized controller in Fig. 12b is the following:

$$((\mathit{add}(\mathtt{B});\ \mathit{con}(\mathtt{A,B}))\ ||\ \mathit{del}(\mathtt{C}));\ \mathit{del}(\mathtt{D})\ +\ ((\mathit{add}(\mathtt{B});\ \mathit{con}(\mathtt{A,B}))\ ||\ \mathit{del}(\mathtt{D}));\ \mathit{del}(\mathtt{C})$$

We use this reconfiguration plan to build a graph transition system and remove the deadlock states.

6. Implementation and evaluation

Implementation We have implemented a tool in Java to support our approach. This tool has two main components: (1) an engine to construct the plant model that takes the model of each component as a state transition system, the initial structure of the system and a reconfiguration plan, and produces the system model during reconfiguration automatically, and (2) the synthesizer that takes the output of the model construction engine, the set of controllable events and a property, and synthesizes a controller to enforce that property. The property to be enforced is provided to the tool as an automaton in the .dot format. The output of the tool is a controller (a graph transition system) stored in a .dot file. We used the tool to produce the results of the illustrating example in this paper that are available.

Case study To evaluate our approach, we use an extension of the example in [BCL+06] to illustrate our approach. In this example, three application servers provide services of types A and B to the clients where their required data are provided by the data servers *DBA* and *DBB*. The *cache handler* is used to determine the best server for handling a request considering the quality of service constraints, and the *logger* monitors the incoming requests. The *request analyzer* analyzes the requests and transmits them to the *request dispatcher*. The latter forwards the request to the proper application server(s). The application servers *server 1* and *server 3* can handle services of both types A and B, *server 2* can handle tasks of type B but delegates tasks of type A to server 3 for performance reasons. When a request is processed, it gets forwarded to the *result aggregator* component. This component is responsible to combine the results received from the servers and sends the result back to the requester and/or the cache handler. According to the load of the system, two different configurations, light-load and heavy-load configurations, are used (Fig. 13). When the number of requests is high, the cache handler is activated, which gets replaced by the logger in the light-load configuration. Application server 2 is active only in the heavy-load configuration, and server 1 handles requests of both types A and B in the light-load configuration.

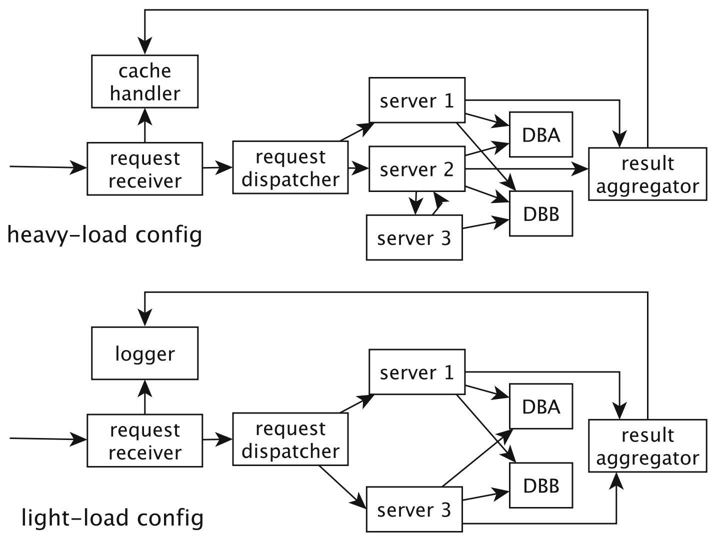


Fig. 13. The reconfiguration of the case study

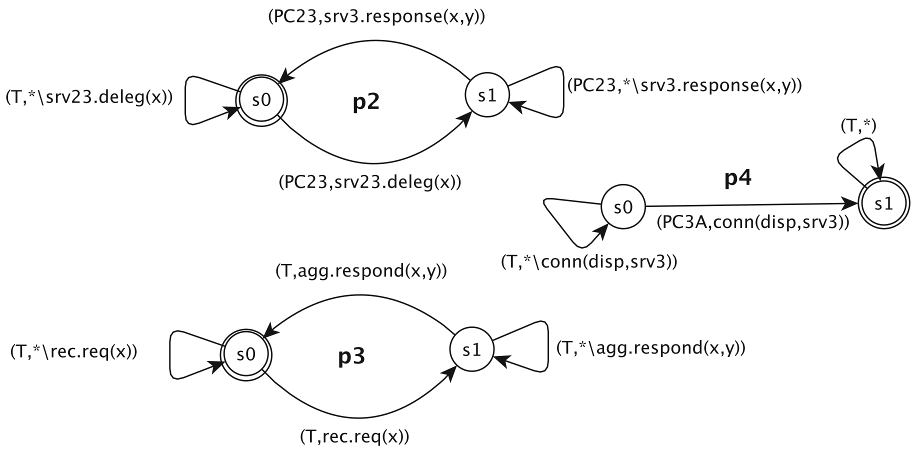


Fig. 14. The properties of the cluster Https servers

Results In our example, different constraints must be enforced to have a correct reconfiguration. Figure 14 shows some of the properties that we checked. A category of properties contains the properties stating that the caller should be connected to the called component. When we construct a model of the system during a reconfiguration, we only allow synchronization on the connected components. Therefore, this category of properties are guaranteed by construction. However, if there is no component to synchronize with, this can cause a deadlock. Deadlock freedom is one of the basic properties that we check (the property **p1**). The property **p2** states that the requests delegated to Server 3 by Server 2 should be responded to before shutting down Server 2, otherwise the responses will be lost. The formula `PC23` is a graph constraint stating that Server 2 and Server 3 are connected. The property **p3** asserts that a request should finally be handled. The property **p4** asserts that Server 3 should be connected to the data server A before connecting it to the dispatcher, otherwise, the new requests cannot be handled correctly. The graph constraints `PC3A` states that Server 3 and the data server A are connected.

Table 1 shows the results of our analysis. The columns of this table, from left to right, show the plan used to reconfigure the system, the number of states/transitions of the system model during reconfiguration, the time for constructing the system model during reconfiguration, the consumed memory, the property to be enforced, the set of events that are uncontrollable by the synthesized controller, the number of states/transitions of the synthesized controller, and the synthesis time. In this table, plans Pl_1 and Pl_2, below, are used to reconfigure the system from the heavy-load to the light-load configuration, and $Pl3$ reconfigures the system from the light-load to the heavy-load configuration:

Table 1. Statistics

Plan	System's states/ transitions	Model const. time (s)	Memory (MB)	Prop.	Uncont. events	Controller's states/ transitions	Synth. time (s)
Pl1	$1.7 \times 10^4/5.8 \times 10^4$	$\simeq$11.7	18	p1	el2	$9.6 \times 10^3/3.4 \times 10^4$	197
				p2	∅	$7.8 \times 10^3/2.7 \times 10^4$	78
					el2	$7.8 \times 10^4/2.7 \times 10^4$	108
				p3	el1	$1.0 \times 10^4/3.6 \times 10^4$	103
				p4	el2	0/0	40
Pl2	$2.4 \times 10^4/8.7 \times 10^4$	$\simeq$15.8	26	p1	el1	$7.7 \times 10^3/2.7 \times 10^4$	12
				p2	∅	$1.2 \times 10^4/4.6 \times 10^4$	572
					el1	$1.2 \times 10^4/4.6 \times 10^4$	576
					el2	$1.2 \times 10^4/4.6 \times 10^4$	579
				p3	el2	$1.3 \times 10^4/4.9 \times 10^4$	496
				p4	el1	$1.1 \times 10^4/4.5 \times 10^4$	425
Pl3	$2.6 \times 10^4/1.0 \times 10^5$	$\simeq$35	31	p1	el1	$2.6 \times 10^4/1.0 \times 10^5$	1460
				p2	∅	$9.8 \times 10^3/3.8 \times 10^4$	422
				p3	el1	$2.3 \times 10^4/7.8 \times 10^4$	2017
				p4	el1	0/0	2996
					el2	0/0	950

$$\begin{aligned} Pl_1 = {} & \texttt{del(srv2)} \mid\mid \texttt{del(ch)};\ \texttt{add(lgr)};\ \texttt{conn(rec, lgr)};\ \texttt{conn(agg, lgr)};\\ & \texttt{conn(disp, srv3)};\ \texttt{conn(srv3, DBA)} \mid\mid \texttt{conn(srv3, agg)} \end{aligned}$$

$$\begin{aligned} Pl_2 = {} & \texttt{dis(rec, ch)};\ \texttt{dis(srv3, srv2)};\ \texttt{dis(disp, srv2)};\ \texttt{add(lgr)};\ \texttt{del(srv2)} \mid\mid \texttt{del(ch)};\\ & \texttt{conn(rec, lgr)};\ \texttt{conn(agg, lgr)};\ \texttt{conn(srv3, DBA)} \mid\mid \texttt{conn(srv3, agg)};\ \texttt{conn(disp, srv3)} \end{aligned}$$

$$\begin{aligned} Pl_3 = {} & \texttt{add(srv2)} \mid\mid \texttt{add(ch)};\ \texttt{del(lgr)};\ \texttt{conn(rec, ch)};\ \texttt{conn(srv2, srv3)};\\ & \texttt{conn(srv3, srv2)};\ \texttt{conn(disp, srv2)};\ \texttt{conn(srv2, DBB)};\ \texttt{conn(srv2, agg)};\\ & \texttt{dis(disp, srv3)};\ \texttt{dis(srv3, DBA)} \mid\mid \texttt{dis(srv3, agg)} \end{aligned}$$

We consider three cases for partitioning the events into controllable and uncontrollable events. In the first case (el1), all events are controllable. This case is suitable when there are facilities provided to disable an event, e.g., all components can control their internal computations and an external entity coordinates the interactions between the components by disallowing interactions. In the second case (el2), receiving a request from outside is uncontrollable; it is not always possible to prevent the requester from issuing a request. If the components are designed as black boxes, we cannot observe their internal behavior and disallow the events associated with their internal computations. Furthermore, the servers altogether can be provided as a (web)-service making their internal interactions invisible from outside. We consider the third case (el3) to model these scenarios, i.e., where the internal computations of components and the interaction between the servers and the data servers are uncontrollable.

We carried out our experiments on an Intel Core i7 2.8 GHz CPU with 16 Gbyte memory running OS X 10.9.5. The size of the system model and the time for constructing the system model depend on the number and the size of components involved in the reconfiguration, as well as the complexity of the reconfiguration plan. For instance, consider the plan $Pl3$ which has more reconfiguration steps compared to $Pl1$ and $Pl2$. The size of its system model is larger than those of $Pl1$ and $Pl2$ and it is more time-consuming to build its system model. The memory used for synthesizing a property depends on the system model size and the property. Since the properties have similar complexities in this case study, the memory usage mainly depend on the system model size, i.e., different properties enforced in the same plan require almost the same amount of memory.

Besides the size of the system model and the property to be enforced, the number of uncontrolled events affects the synthesis time too. Consider the plan $Pl1$ and the property $p2$. The controllers synthesized for the empty uncontrolled event set and the set $e2$ are the same, but it takes less time to synthesize a controller for the event set ∅. The difference is due to the time consumed to check if a transition label is in the uncontrollable event list $e2$. For some cases, the tool could not find a controller to enforce the property. For instance, there is no controller to enforce the property **p4** in the plan $Pl1$. This is because Server 3 is not connected to the dispatcher [holds after executing the event conn(srv3, DBA)] when the event conn(disp, srv3) occurs. Due to the complexity of the synthesized controllers, it is impossible to show them in the paper; however, the synthesized controllers are available. We use Graphviz [GRP] to draw transition systems/automata in dot or gv format.

We believe that the controller synthesis is expensive, in particular for real-life applications. However, since the synthesis process is performed offline, it does not have a high run-time cost. Furthermore, our experiments show that the tool can synthesize controllers for systems with large state spaces that are much more difficult to obtain manually. We tested the validity of the results for some smaller examples. The output of the tool can be used for further analysis during the software development cycle. Once we have obtained a controlled system, we can continue working on it in the design phase, for instance through model checking to verify properties other than those already enforced by the controller, that may be relevant for the complete system. Automated generation of executable code for the synthesized controller is yet another important step in the implementation phase of a system under design. Adding this step completes our methodology by integrating controller synthesis into a complete design flow, as done in [DRM13]. Such an executable can be used for simulation or for actual run-time execution of the controlled system.

7. Related work

The aim of reconfiguration control synthesis in [AFI$^+$06, TFGG07, AMNT08] is to coordinate the interaction behavior of the components in order to avoid undesirable behavior such as deadlocks. The interaction behavior of the system and the desirable behavior are modeled using labeled transitions systems. Then, algorithms are proposed to synthesize (distributed) adaptors to coordinate the components' interactions. Furthermore, supervisory control theory is used to synthesize behavioral adaptors to adjust the communication between services such that a certain behavioral property holds in the composed system [GMW12]. The main difference between our approach and the above approaches is that they are mainly concerned with synthesizing the behavioral adaptors to coordinate the interactions of components/services, while we are interested in correct-by-construction design of controllers to guarantee the safe structural adaptations. We model both the behavior and the structure of the system, while the above approaches deal with only behavioral modeling of the component interactions. We specify properties involving both structure and behavior of a system, but only behavioral properties are taken into account in [AFI$^+$06, TFGG07, AMNT08, GMW12]. Compared to [AFI$^+$06, AMNT08] we can synthesize a centralized adaptor and we do not deal with real-time properties such as latency, performance, etc., as done in [TFGG07].

The authors of [BK08, KB04] proposed an approach based on the concept of proof lattice to verify if a system is in a correct state during and after adaptation in terms of satisfying the transitional-invariants. In this approach, the behavior of a system during adaptation is specified using an adaptation lattice in which a node is an automaton denoting the behavior of a possible intermediate program. Although verification identifies undesirable behavior, one has to fix errors manually while using synthesis techniques, one can generate a controller to control the adaptation phase. In this work, the properties to be verified are only about the behavior of the system while we consider both structural and behavioral properties.

Discrete Control Theory has recently been applied to computing systems. We restrict ourselves to present the works done in the area of dynamic adaptation and component-based system. In [GN12, GVNH11], the authors consider run-time exceptions raised by programs and not handled by the code. Supervisory control is used to modify and adapt programs in such a way that the un-handled exceptions will be inhibited. In terms of autonomic computing, this corresponds to a form of self-healing of the system.

Applying control theory to design self-adaptive systems has recently received attentions. In [FHM14], the authors propose a methodology for automatically constructing a model of the system dynamically and synthesize a continuous controller to enforce non-functional requirements at runtime, and tune the variables to achieve the goal in the presence of unpredictable disturbances. In this work, the model is updated dynamically at runtime. In [GGM12], the aim is introducing safe and automatic self-adaptation mechanisms driven by the requirement and environment changes. In this research, the system is controlled by a controller that enforces requirements and environment assumptions. If the specification changes, the approach attempts to find a safe state to update the system behavior to satisfy the new specification, and then drive a new controller from the old controller to implement the changed specifications. A multi-tier framework is proposed in [DBK$^+$14] for designing the planning layer of a self-adaptive system. In this framework, each tier is associated with its own set of goals to be achieved and the environment model. Then for each tier, a control problem is solved to synthesize a controller to achieve the goal of that tier given the environment model. This approach allows to provide graceful degradation and progressive functional enhancement at runtime based on the fact that the assumptions of the tiers hold. The focus of [FHM14, GGM12, DBK+14], in contrast to ours, is not on synthesizing a controller to provide safe adaptation during an adaptation, but their goal is to provide techniques to design self-adaptive systems.

In an approach related to reactive systems and synchronous programming, discrete controller synthesis, as defined and implemented in the tool Sigali, is integrated in a programming language compiler BZR [DRM13], used in component-based software [BSDR11].

Furthermore, interface synthesis [CdAHM02] is also related to Discrete Controller Synthesis, and consists of generating interfacing wrappers for components, in order to adapt them for the composition into given component assemblies, with respect to the communication protocols between them.

There are several works on integrating graph-based formalisms with the behavioral modeling approaches to model an adaptive system. In [WGT14], the authors combine Modal Sequence Diagrams and graph transformation techniques to model message-based interaction behavior and structural reconfigurations. Similar to our work, they use synchronous message passing. In this work, the events are implemented using graph transformation rules, while we only implement structural reconfiguration events using (simple) graph evolutions. In another work, [HH11] integrates real-time state-charts and time graph transformation systems where the adaptation behavior is specified by graph transformation rules and the remaining behavior is modeled in state-charts. The authors in [HHG08] present an approach to model collaborations among several participants (roles) where the components can join and leave collaborations dynamically. They use state charts to describe the behavior of each role and graph transformation to express the structural changes. The authors of [HH11, WGT14, HHG08] do not model the behavior of the system during an adaptation and assume that a reconfiguration is performed in one step, specified by graph transformation rules, while we are concerned about the system behavior during an adaptation. Furthermore, they can specify behavioral adaptations, and we only focus on structural adaptations in this paper.

Koenig [Koe04] investigates graph-transformation techniques for modeling and formal analysis of adaptive systems. In this thesis, the focus is on both behavioral and structural adaptations, while we only concentrate on structural adaptation. As a part of the RAPIDware project, Zhang and Cheng [ZC06] proposed a model-driven approach for developing adaptive systems. In this approach, different contexts in which an adaptive program may run are determined according to high-level requirements specified by a formalism like temporal logic. The local properties of the program in each context are described formally. Then, a state-based model of the program in each context, as well as the adaptation models for the adaptations of the program from one context to another, are built. Different behavioral variants of a program are modeled as Petri Nets in [ZC06]. They specify the system behavior during an adaptation, as we do in this paper. However, they only deal with behavioral adaptation, while we consider structural adaptations. In contrast to our work, they do not deal with correct-by-construction adaptations and their focus is on formal verification of adaptive systems.

8. Conclusions

We proposed an approach for synthesizing a controller to control dynamic reconfigurations in adaptive systems. We modeled the problem of reconfiguration control synthesis as a supervisory control problem. To this end, we adapted the supervisory control problem to support reasoning about the structure of system. The property to be applied is specified using a graph automaton and the system is modeled using a graph transition system. We have implemented a tool to support our approach and have applied it on a case study in the area of clustered https-servers.

There is much more research to pursue in the area of reconfiguration control synthesis. Convergence of system state toward a stable state during adaptation is an important issue. It is often the case that an adaptation causes another adaptation which in turn leads to another, and so on. If this cycle continues without reaching a stable state, we say the system is in an unstable state. Extending our approach to avoid instability during reconfiguration is a future work. Finding the best strategy for performing an adaptation with minimum transient states is another issue which we did not consider in this research. There are different approaches to solve the SCP problem. In this paper, we chose and adapted the RW framework to solve our problem, however we are interested in investigating other advanced approaches, particularly game-theoretic approaches (e.g. [Job07]) to optimize the synthesis process. An adaptive system evolves dynamically, and both the system model and the specification may change over time. In order to take into account the system and the specification dynamics, we will focus on online controller synthesis in the future.

References

[AFI+06] Autili M, Flammini M, Inverardi P, Navarra A, Tivoli M (2006) Synthesis of concurrent and distributed adaptors for component-based systems. In: Third European workshop on software architecture, LNCS, vol 4344. Springer, pp 17–32

[AMNT08] Autili M, Mostarda L, Navarra A, Tivoli M (2008) Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. J Syst Softw 81(12):2210–2236

[BCL+06] Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani J-B (2006) The fractal component model and its support in java. Softw Pract Exp 36(11-12):1257–1284

[BK08] Biyani KN, Kulkarni SS (2008) Assurance of dynamic adaptation in distributed systems. J Parallel Distrib Comput 68(8):1097–1112

[BSDR11] Bouhadiba T, Sabah Q, Delaval G, Rutten É (2011) Synchronous control of reconfiguration in fractal component-based systems—a case study. In: Proceedings of the ACM conference on embedded software, EMSOFT, Taiwan, Oct 2011

[CdAHM02] Chakrabarti A, de Alfaro L, Henzinger TA, Mang FYC (2002) Synchronous and bidirectional component interfaces. In: Computer aided verification, LNCS, Copenhagen, Denmark, vol 2404, pp 414–427

[DBK+14] D'Ippolito N, Braberman V A, Kramer J, Magee J, Sykes D, Uchitel S (2014) Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: 36th international conference on software engineering, ICSE '14, pp 688–699

[DRM13] Delaval G, Rutten E, Marchand H (2013) Integrating discrete controller synthesis into a reactive programming language compiler. Discrete Event Dyn Syst 23(4):385–418

[FHM14] Filieri A, Hoffmann H, Maggio M (2014) Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 299–310

[GGM12] Ghezzi C, Greenyer J, Manna VPL (2012) Synthesizing dynamically updating controllers from changes in scenario-based specifications. In: 2012 ICSE workshop on Software engineering for adaptive and self-managing systems (SEAMS), pp 145–154

[GMW12] Gierds C, Mooij AJ, Wolf K (2012) Reducing adapter synthesis to controller synthesis. IEEE Trans Serv Comput 5(1):72–85

[GN12] Gaudin B, Nixon P (2012) Supervisory control for software runtime exception avoidance. In: Proceedings of the fifth international C* conference on computer science and software engineering, C3S2E '12. ACM, New York, pp 109–112

[GRP] Graphviz-graph visualization software. http://www.graphviz.org/

[GVNH11] Gaudin B, Vassev E I, Nixon P, Hinchey M (2011) A control theory based approach for self-healing of un-handled runtime exceptions. In: Proceedings of the 8th ACM international conference on Autonomic computing, ICAC '11. ACM, New York, pp 217–220

[HH11] Heinzemann C, Henkler S (2011) Reusing dynamic communication protocols in self-adaptive embedded component architectures. In: Proceedings of the 14th international ACM sigsoft symposium on component based software engineering, CBSE '11. ACM, New York, pp 109–118

[HHG08] Hirsch M, Henkler S, Giese H (2008) Modeling collaborations with dynamic structural adaptation in mechatronic uml. In: Proceedings of the 2008 international workshop on software engineering for adaptive and self-managing systems, SEAMS '08. ACM, New York, pp 33–40

[Job07] Jobstmann B (2007) Applications and Optimizations for LTL Synthesis. Ph.D. thesis, IST—Institute for Software Technology, TU Graz

[KAR14] Khakpour N, Arbab F, Rutten E (2014) Supervisory controller synthesis for safe software adaptation. In: Proceedings of the 12th IFAC workshop on discrete event systems

[KB04] Kulkarni SS, Biyani KN (2004) Correctness of component-based adaptation. In: Component-based software engineering, LNCS, vol 3054/2004. Springer, pp 48–58

[Koe04] Koenig B (2004) Analysis and verification of systems with dynamically evolving structure. Ph.D. thesis, Universitat Stuttgart

[RW87] Ramadge PJ, Murray Wonham W (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230

[TFGG07] Tivoli M, Fradet P, Girault A, Gößler G (2007) Adaptor synthesis for real-time components. In: Tools and algorithms for the construction and analysis of systems, LNCS, vol 4424. Springer, pp 185–200

[WGT14] Winetzhammer S, Greenyer J, Tichy M (2014) Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems. In: Amyot D, Fonseca i Casas P, Mussbacher G (eds) System analysis and modeling: models and reusability, Lecture notes in computer science, vol 8769. Springer International Publishing, Berlin, pp 126–141

[ZC06] Zhang J, Cheng BHC (2006) Model-based development of dynamically adaptive software. In: Proceedings of the 28th international conference on Software engineering, ICSE '06. ACM, New York, pp 371–380

Received 6 March 2015
Accepted in revised form 14 October 2015 by Heike Wehrheim
Published online 4 January 2016