Technical Report 2016-630

Title: A Controller Synthesis Framework for Automated Service Composition

Authors: Francis Atampore, Juergen Dingel and Karen Rudie

School of Computing
Queen's University
Kingston, Ontario, K7L 3N6, Canada

September 12, 2016

A Controller Synthesis Framework for Automated Service Composition ☆

Francis Atampore[a,*], Juergen Dingel[a], Karen Rudie[b]

[a] *School of Computing, Queen's University, Kingston, Ontario, K7L 3N6, Canada*
[b] *Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, K7L 3N6, Canada*

Abstract

Nowadays, Web services allow interoperability among distributed software applications deployed on different platforms and architectures which in effect plays a major role in electronic businesses. Web services allow organizations to carry out certain business activities automatically and in a distributed fashion. However, in some circumstances, a single service is not able to perform a certain task and it becomes imperative to compose two or more services in order to complete a task. Thus, a key research challenge in this field is the problem of *automatic service composition*. Several approaches exist that tackle the problem of automatic service composition, however, the task of generating provably correct Web service compositions still remains a challenging and complex task.

In this paper, we develop a formal framework for modeling Web service compositions based on Supervisory Control Theory (SCT) of discrete-event systems. We model services that exchange messages and exhibit nondeterministic behaviours. The objective is to synthesize a supervisor which interacts with a given set of Web services through messages to guarantee that a given specification is satisfied. A key novelty of this work is the application of control theory to service-oriented computing and the incorporation of run-time input into the supervisor generation process. First, we describe a novel supervisory control framework for automated composition of Web services. The framework employs Labelled Transition Systems (LTSs) equipped with guards and data variables to model Web services and provides a technique to synthesize a controller. We model the interactions of services asynchronously and we use the guards and data variables to allow us to express certain preconditions which are then propagated from the system requirements through the overall composite service. Second, we then develop a set of algorithms to generate a controller satisfying a given functional requirement also specified in LTSs. Besides the standard disabling and enabling of events, the generated controller in our framework has the ability to enforce certain events based on run-time information to drive the system towards its goal. In addition, the controller is able to impose restrictions on the kind of data that can be sent or received by services. This includes the automatic generation of stronger guards or preconditions which impose restrictions on which path to take during execution. Lastly, we state a theorem capturing the existence of a controller and provide a proof to demonstrate the correctness of the proposed approach.

Keywords: Automatic Service Composition, Supervisory Control Theory, Controller Synthesis, Web Services, Discrete Event Systems, Labelled Transition Systems

☆ The authors gratefully acknowledge support from the Natural Sciences and Engineering Research Council of Canada
* Corresponding author
Email addresses: akfransua@yahoo.com (Francis Atampore), (dingel@cs.queensu.ca) (Juergen Dingel), (karen.rudie@queensu.ca) (Karen Rudie)

1. Introduction

Today, we are undergoing a major paradigm shift in software development as a result of the emergence of the World Wide Web. This has brought about the Service-Oriented Computing paradigm which aims to encapsulate software components and expose them as services through network-accessible, platform and language independent interfaces. The most widely adopted software design patterns in SOC for realizing the SOC model into an architecture is known as Service-Oriented Architecture (SOA) [17], which in essence is a logical way of organizing and developing distributed software systems by providing services to end-user applications or to others, and whose interface descriptions can be published and discovered. The most widely used implementation of SOA are Web services [35].

The vision underpinning Web services can be fully achieved if we envision a collaboration between a community of numerous service providers and service consumers who interact in order to achieve certain business goals. Thus, one of the key functionalities of Web services is *service composition*, which seeks to create, select and integrate pre-existing services to develop new value-added services and applications. Hence, it promotes the rapid development of software systems by reducing the cost and effort for developing new services from scratch. Furthermore, the output of a service composition (i.e., *composite service*) could serve as the input (*atomic service*) for further service compositions (re-usability). A typical motivating example of a Web service composition is given as follows. Consider a group of business clients who are going on a business trip and want to make a reservation for a flight ticket, a hotel room and a car for a particular destination and a period of time. There exist only an airline reservation Web service, a hotel reservation Web service and a car rental Web service separately. Clearly, we want to combine these Web services rather than implementing a new one.

Several approaches have been proposed that attempt to address the problem of automatic service composition and most of these techniques are motivated by the works in AI planning and cross-enterprise workflow techniques [60, 27, 11, 29, 23, 40, 44]. However, the task of generating provably correct Web service compositions still remains a challenging and complex task.

In this paper, we develop a formal framework **Supervisor Aware Service Composition Architecture (SASCA)** for modeling Web service composition based on Supervisory Control Theory (SCT) of Discrete-Event Systems (DES). Discrete-Event Systems control theory is a branch of control theory that models behaviours as sequences of discrete events instead of continuous functions of time. The classical Ramadge-Wonham approach to the supervisory control problem (SCP) [14], [55] is defined as follows: given a plant $\mathcal{G}$ modeled in the form of a state-transition system which captures the behaviours of the process to be controlled according to some possible events, given a set of specifications $\mathcal{L}$ which describes the legal sequences of events of the plant, synthesize a supervisor $\mathcal{S}$ so that $\mathcal{S}$ restricts $\mathcal{G}$ in such a way that all its executions satisfy $\mathcal{L}$ and such that $\mathcal{S}$ is maximally permissive. The DES control methods fit well into the problem of automatic service composition if the problem is reduced to observing events of a system and restricting its behaviour to specific sequences. Hence, we can apply existing DES techniques and algorithms to address the problem of automatic service composition. With respect to other service composition approaches, Supervisory Control Synthesis has several benefits which are as follows. It results in a correct-by-construction control synthesis, and the generated controller is maximally permissive by preventing a system behaviour only if it violates the system requirements. It also relies on automata theory to provide a well-defined syntax and semantics for modeling systems which could be very useful for specifying services. In addition, DES provides a standard way that can be used to model various business logics and requirements in a dynamic environment. Supervisory control theory has been applied to generate concurrency control code in multi-threaded programs, and some these works have been very successful. Notable ones are the work by Dragert et al. [16, 4] and the work by Wang et al.[52, 51]. It is based on this background that we propose to apply

SCT to service-oriented computing. To the best of our knowledge our work is the first of its kind to apply SCT to deal with the problem of Web service composition.

The approach we propose employs Labelled Transition Systems (LTSs) equipped with guards, and data variables to model a given set of Web services specified in WS-BPEL [3]. To this end, we provide an SCT modeling formalism based on LTSs and then we describe a novel technique to synthesize a composition satisfying a given functional requirement (data and control flow) also specified in LTS. That is, the problem of synthesizing a composition can be reduced to the problem of supervisor synthesis when the available services and a goal specification represent the plant and the legal language (desired behaviour), respectively, in DES. In this way, the problem of orchestrating data and control flow requirements can be achieved using the notion of controllability in DES where the supervisor enacts control by disabling and enabling certain actions in order to enforce the given goal. The inputs to the system are the set of Web services specified in WS-BPEL and the requirements also specified in LTSs. Internally, we represent these Web services using LTSs, Next, the proposed supervisory control framework based on LTSs is applied to synthesize a controller that ensures that the given composition requirements are satisfied. A key novelty of this work is the application of control theory to service-oriented computing and the incorporation of run-time input into the supervisor generation process. The contributions that we make in this paper are as follows:

(I) We provide a novel supervisory control framework for automated composition of Web services, which uses Labelled Transition Systems (LTSs) with guards and data variables to model a given set of Web service specifications in industrial standard languages like WS-BPEL. The framework models the interactions of services asynchronously and uses guards and data variables to allow us to express certain preconditions which are then propagated from the system requirements through the overall composite service. We provide insight into how to express and define functional requirements (data and control flow) for a composition and then we provide a formalism for the problem of automated composition based on controller synthesis.

(II) We then develop a set of algorithms based on the formalism provided to generate a controller satisfying a given functional requirement also specified in LTSs. Beyond the standard disabling and enabling of events, the generated controller in our framework has the ability to enforce certain events based on run-time information to drive the system towards its goal. In addition, the controller is able to impose restrictions on the kind of data or variables that can be sent or received by the services. This includes the automatic generation of stronger guards or conditions which impose restrictions on which path to take during execution.

(III) We demonstrate the correctness of our approach by stating a theorem on the existence of a controller and providing a proof for this theorem.

The rest of the paper is organized as follows. First, in Section 2, we will provide a brief introduction to the standard supervisory control theory of DES and then, we will describe informally the problem of automated service composition by means of an illustrative example in the travel domain. In Section 3, we present the formal language and various definitions that will serve as the basis for our formalism. In Section 4, we develop the service composition framework based on the formalism provided in Section 3. We present the set of algorithms for composition synthesis in Section 5 and in Section 6, we provide proofs for our formalism. In Section 7 we discuss the relevant related work and in Section 8 we provide some concluding remarks and an overview of future work.

2. Background

2.1. Supervisory Control Theory

In this section, we give a brief overview of the supervisory control approach of Ramadge and Wonham, often called Ramadge Wonham Theory (RWT) [42]. They considered a DES modeled at an untimed level of abstraction which relies on feedback control to restrict the system so as to achieve a given set of specifications. In this framework, the behaviour of a system is modeled with an FSM known as the plant $\mathcal{G}$. Events are represented by the transitions in $\mathcal{G}$, and the language generated $\mathcal{L}(\mathcal{G})$ represents the behaviour of $\mathcal{G}$. That is, the language $\mathcal{L}(\mathcal{G})$ contains strings that may be unacceptable because they violate some rule or nonblocking conditions that we wish to impose on the system. The undesirable behaviour could be states where $\mathcal{G}$ blocks, through a deadlock or a livelock, or inadmissible states. To this end, the behaviour of $\mathcal{G}$ is not satisfactory and must be controlled by restricting the behaviour to a subset of $\mathcal{L}(\mathcal{G})$. A supervisor $\mathcal{S}$ is introduced in order to alter the behaviour of $\mathcal{G}$.

In the basic model, the event set of $\mathcal{G}$ is partitioned into two disjoint sets, namely, the *set of controllable events* Σ_c, meaning an event in Σ that can be disabled and the *set of uncontrollable events* Σ_{uc}, meaning an event in Σ that cannot or should not be prevented from occurring. According to Cassandras et al. [14], an event might be modeled as uncontrollable because it is inherently unpreventable or it models a change of sensor readings not due to a command; it cannot be prevented due to hardware or actuation limitations; or it is modeled as uncontrollable by choice, as an example when the event has high priority and thus should not be disabled, or when the event represents the tick of a clock. Ideally, a supervisor is given by a function $S : \mathcal{L}(\mathcal{G}) \rightarrow 2^{\Sigma}$ that maps the sequence of generated events to a subset of controllable events to be disabled. The synchronous product of $\mathcal{S}$ and $\mathcal{G}$ is the marked language denoted by $\mathcal{L}_M(\mathcal{S}/\mathcal{G})$ that represents the behaviour of the plant $\mathcal{G}$ under supervision of $\mathcal{S}$. The work by Ramadge et al. [55] states the necessary and sufficient conditions for the existence of a supervisor. In this framework, a specification given as an FSM provides the *desired* behaviour of the plant, and is called the *legal language* $\mathcal{E}$. A plant $\mathcal{G}$ is called *controllable* with respect to a specification $\mathcal{E}$ if for any string s from the prefix closure of $\mathcal{E}$, there are no uncontrollable events e that could be generated by $\mathcal{G}$ at the state reached by s, such that se would not be in the prefix closure of $\mathcal{E}$. That is, if something cannot be prevented, it must be legal. The Supervisory Control Architecture addressed by this work assumes that the plant spontaneously generates all events and the role of the supervisor is to enable/disable controllable events (since $\mathcal{S}$ is not allowed to ever disable a feasible uncontrollable event). The supervisor $\mathcal{S}$ guarantees not only deadlock-freedom (nonblocking) and adherence to the specification $\mathcal{E}$, but also minimal restrictiveness. These strong theoretical guaranteeness set DES apart from other approaches.

2.2. Highlights of the Proposed Method

In the following, we will highlight some of the relevant concepts that are pertinent to the proposed technique. Our approach lies between the event-based supervisory control [55] and state-avoidance control problem [22]. However, our approach extends the basic control capabilities in these approaches as follows. Apart from the supervisor being able to prevent certain events from occurring by properly disabling and enabling controllable events, the supervisor is also able to prevent the system from reaching certain sets of states designated as *forbidden* states by using run-time information of variables in the system. This is due to the fact that some of the events in our model are black-box (atomic actions) in nature and may exhibit non-deterministic properties. We deal with this non-determinism through model refinement and the adaptation of event enforcement supervisory control theory [15]. Therefore, the generated supervisor not only restricts the behaviour of the plant, but also has the opportunity to actively enforce certain events. In addition, the

generated supervisor is able to restrict the system by assigning stronger guards to data variables. This allows us to control the data a service can send or receive.

The use of run-time information in our model is inspired by the fact that the services we model are nondeterministic and partially controllable. That is, the outputs of a service cannot be predicted a priori and some of the internal computations of a service are hidden from other external services. For example, the information whether there are still seats available on a flight cannot be known until run-time or whether there is enough cash in a customer bank account will only be available at run-time. Due to non-determinism and the black box nature of some of the events in our model, the use of the classic supervisory control theory in many cases will result in an empty controller or an overly restrictive controller.

Intuitively, in order to model services using supervisory control theory, we need to provide support for (i) message exchanges (ii) data and variables (iii) conditions and (iv) information or data that may not be known until runtime. Standard supervisory control theory may not be sufficient to model services or may lead to an overly restrictive controller.

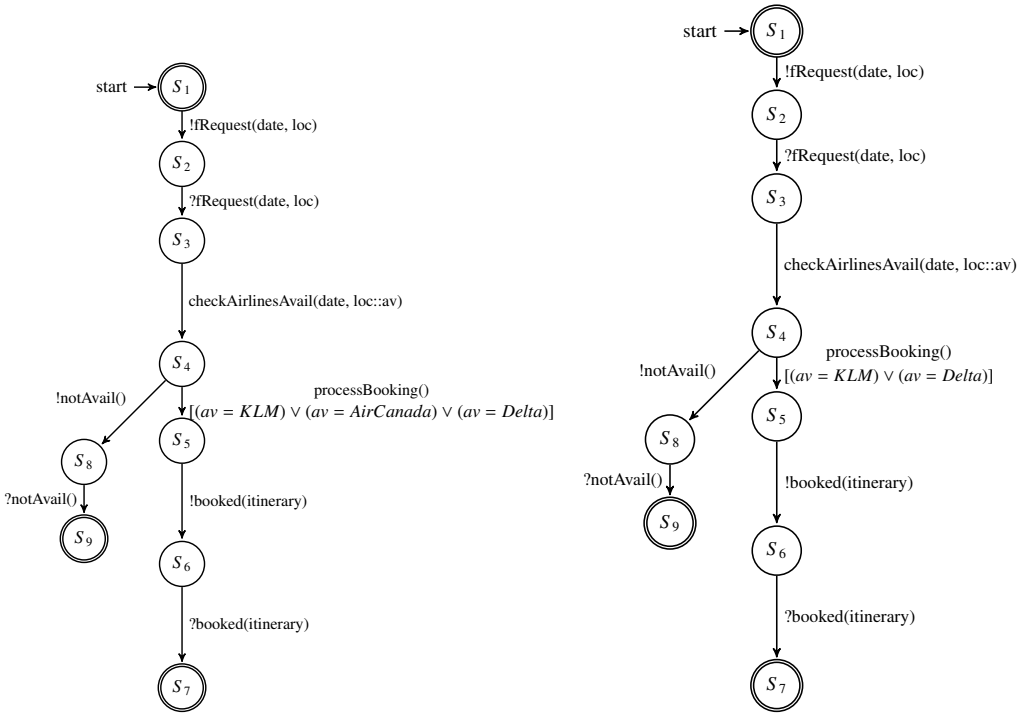
Let us illustrate the above discussions with the following example. Consider Figure 1 which models an airline reservation system and its specification. For now we are not interested in the formal details of this example; we will deal with this in subsequent sections. Figure 1(a) represents an airline system (plant) which upon accepting a request, checks the availability of an airline and returns an offer if it is available. Assume that Figure 1(b) is the system requirements to be met. The specification permits the plant to do everything except for the transition from state S_4 to S_5 where there is a restriction on what branch to take based on the value of the variable. Assume the transitions labeled *checkAirlinesAvail*(*date*, *loc* :: *av*) and *processBooking*() are not controllable. Now, at state S_4 the specification allows the plant to transition to state S_5 only if the value of the variable *av* is either KLM or Delta; anything else is not allowed. The use of the classic DES will never allow the plant to reach state S_4 which implies that the system will receive a request but will never return a response, which does not make much sense in the service domain. In other words, it will be overly restrictive if the classic DES is applied directly. The use of supervisory control based on extended finite state machines would also not work, since the value of the variable *av* is unknown and can not be tracked until state S_4. The event *checkAirlinesAvail*(*date*, *loc* :: *av*) is assumed to be black-box, i.e., the effect that it has on variables are not known.

2.3. Service Composition Problem

In this section, we present the composition problem by means of an example. The example described here is a modified version of a well-known example for illustrating Web service composition in the business domain. As mentioned above, the service composition problem comes into play when a request to a service (component service) is not satisfied by a single service and then multiple services are identified and constructed into a new service (*composite service*) to satisfy the request or the desired functionality through orchestrating the services involved.

Let us consider a *Flight Reservation and Purchase System (FRPS)* that offers customers travel packages by allowing customers to make a reservation for a specified airline and to make payment in order to reserve the flight. All interactions are managed by Web services. The objective of the *Flight Reservation and Purchase System* can be attained by composing an *Airline Service*, a *Bank Service*, a *Hotel Service* and an *On-line Customer Interface Service*. We assume that these services are represented in WS-BPEL. The main challenge is how to compose these services so that the user can directly ask the combined service to reserve and purchase a ticket satisfying some given system requirements. In the following, we will provide an informal description of these services.

- Airline Service: The Flight service is designed to receive requests for booking a specified flight for a given date and location. It checks an internal database for flight availability, and sends an offer with



(a) S_1 (b) S_2

Figure 1

a cost and a flight schedule in response to the client's request. The client can either accept or refuse the offer; if the client decides to accept the offer, the FRPS will book the flight and provide additional information such as an electronic ticket.

- Bank Service: The Bank service is designed to receive a request to check that a credit card, debit card or money order can be used to make a payment and provides an option for its clients to check their current balance or withdraw from the account to make a payment for a purchase. The transaction may fail if the card provided is not valid or if there are not sufficient funds in the client's accounts.

- Hotel Service: The Hotel service accepts requests for providing information on available hotels for a given date and a given location. It checks for the availability of hotels and selects a specific hotel based on the client's request and returns an offer with a cost and other hotel information. The external service that invoked the Hotel service can choose to refuse or accept the offer. In case of acceptance, the hotel proceeds with the booking and sends a confirmation message to the client.

- On-line User Interface Service: This service serves as a customer interface through which the client can interact with the Flight Reservation and Purchase System. It receives input messages from the user and in return sends output messages to the user and also facilitates interactions among the available services.

The following is a typical sequence of events that can take place when making a reservation using the above

services. The customer makes a travel reservation by sending a request through the On-line User Interface Service to the Flight Reservation and Purchase System. The request is received by the Flight Reservation and Purchase System which may specify the type of airline (example, KLM, Delta or Air Canada), the location and the time of travel as well as the details of the hotel the customer wants. The Flight Reservation and Purchase System checks for the availability of a flight based on the information provided by the customer and a given system requirement, and returns an offer to the customer if available, otherwise a failure message is generated. In case an offer is sent to the customer, he or she may decide to accept or cancel the offer. In the event that the customer accepts to purchase the ticket, then the Flight Reservation and Purchase System proceeds to check the customer's credit card or authenticates his or her debit card or any other means of payment and finally transfers an appropriate amount of funds from the client's bank account to the airline's bank account. Composing these services must take into account certain business constraints such as the following: (1) The Hotel service should not be booked if the flight is not available. 2) The client can make payment using either a debit card or a credit card but not money order payment. 3) The composed service should process only flight reservations involving KLM or Delta Airlines but not Air Canada. 4) The customer must accept the offer before his or her bank account is charged.

3. Service and Supervisory Control Theory Representation

In this section, we discuss a well known formal model known as Labelled Transition Systems which can be used to express services represented in the standard WS-BPEL/WSDL. That is, in order to represent services we use Labelled Transition Systems augmented with guards and variables which have semantics similar to most existing service description languages (e.g., WSDL, WS-BPEL) and can easily be modeled and manipulated to provide a solution to the composition problem. As noted before, basic supervisory control theory utilizes finite state machines (FSM) for modeling systems which falls short when it comes to service representations with variables and data. Labelled transition systems extended with data variables and guards which we will refer to as a Service Labelled Transition System (SLTS) have the ability to model the manipulation of data conveniently, and to support compact representation of models relative to FSM. In addition, finite state machines are not sufficient to represent reactive systems and systems that store and exchange data information. A Service Labelled Transition System will have infinitely many states whenever the variables take their values from an infinite domain (often called Symbolic Transition Systems). Hence, an SLTS can have an infinite number of states. However, in this paper, we consider a deterministic finite state SLTS which takes its variables from a finite domain. Before going into the details of our formalization, we will first formally define the notion of an SLTS and its properties.

3.1. The Web Service Labelled Transition System (SLTS)

In this paper, we assume that each WS-BPEL/WSDL of a Web service is formally represented as a Service Labelled Transition System. In the literature this formal language is often referred to as a guarded automaton [21]. It is essentially an Extended Finite State Machine (EFSM) [49] without an update function or an action language.

The formal model we present here consists of a set of *states* which model the dynamism of a system. The evolution of the system from one state to another is determined by its current *state* and the evaluation of a guard of a transition. We distinguish among four kinds of events, namely, input actions, which represent the reception of messages, output actions, which represent messages sent to external services, atomic actions which may modify the value of a variable arbitrarily, and a special action τ, which represents an internal unobservable event.

Definition 3.1. *A Service Labelled Transition System is modeled by a tuple* $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ *where*

- $\mathcal{S}$ *is a finite set of states*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states*
- $\Sigma = \mathcal{I} \cup \mathcal{O} \cup \mathcal{A} \cup \{\tau\}$ *is the set of events (i.e, set of actions), where* $\mathcal{I}$, $\mathcal{O}$, $\mathcal{A}$ *denote the set of input messages (denoted* $?m(\overrightarrow{x})$*), outputs messages (denoted* $!m(\overrightarrow{x})$*) and atomic operations, respectively, and* $\mathcal{I} \cap \mathcal{O} = \emptyset$*, and* τ *represents an internal unobservable event*
- $V = \{v_1, ..., v_n\}$ *is a finite set of data variables over a given domain*
- $\Gamma \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \mathcal{A}) \times G \times \mathcal{S}$ *is the transition relation, where G is the set of guards over a subset of the variables in V*
- $\mathcal{S}^F \subseteq \mathcal{S}$ *is the set of final states*

We employ infix notation and we write $s \xrightarrow{e[g]} s'$ as shorthand for $(s, g, e, s') \in \Gamma$. A tuple $\lambda \in \Gamma$, $\lambda{=}s \xrightarrow{e[g]} s'$ is a transition in $\mathcal{G}_W$, where $e \in \Sigma$, and g is a guard in G, a condition or a predicate defined over variables and formulas. The absence of an explicit guard on a transition means that the condition is always true. The dynamics of an SLTS depends on the current stage of the system and on the valuation of the transition guards with respect to the current value of a variable. In the sequel, let Σ^* denote the set of all finite strings of the form $\alpha_1\alpha_2...\alpha_n$ of events from Σ, including the empty string ϵ.

The possible behaviour of an SLTS is modeled by the set of *executions*. The *execution* of an SLTS is represented by the set of all possible *runs*. A *run* of an $SLTS$ is a sequence of transitions $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} ... \xrightarrow{\alpha_{n-1}[g_{n-1}]} s_n$ such that $\forall i < n$, $s_i \xrightarrow{\alpha_i[g_i]} s_{i+1} \in \Gamma$ and the trace of the run is given by $\alpha_1\alpha_2...\alpha_{n-1}$. The *language* generated by an SLTS, denoted by $\mathcal{L}(\mathcal{G}_w)$, is the set of words $\mathcal{L}(\mathcal{G}_w) = \{w \in \Sigma^* | s_0 \overset{w[g]}{\Longrightarrow} y, s_0 \in \mathcal{S}^0, y \in \mathcal{S}\}$ where $s_0 \overset{w[g]}{\Longrightarrow} y$ denotes a multi-step transition relation which is defined inductively as a finite sequence of applications of a transition relation which produces a state y that a sequence of events leads to from the initial state s_0 and g denotes a sequence of guards.

The formal language that we define here differs from extended finite state automata [49] in that we do not require an update function. Hence, in some cases it becomes impossible to track the values of variables in our model. That is, there is no action language, but we assume the updating of variables is done internally, which makes it difficult to track the values of variables.

We distinguish between two kinds of transitions, the first one is a *static transition* $s \xrightarrow{e} s'$ which does not depend on a variable; the guard of this transition is always true and is triggered when the event on the transition takes place. The second type of transition is a *dynamic transition* $s \xrightarrow{e[g]} s'$ which has non-trivial guards that depend on a variable which is fired only if the guards on the transition evaluates to true and the event has occurred. For example, in Figure 1(a), the transition $S_1 \xrightarrow{!fRequest(date,loc)} S_2$ is a static transition whereas the transition $S_4 \xrightarrow{processBooking()\,[(av=KLM)\vee(av=AirCanada)\vee(av=Delta)]} S_5$ is a dynamic transition.

Definition 3.2. *Subguards*
Let g_1 *and* g_2 *be two guards. We call* g_2 *a subguard of* g_1 *denoted by* $g_2 \leq g_1$*, if* g_2 *is stronger than* g_1*, i.e.,* $g_1 \wedge g_2 = g_2$.

In order to model behaviours common to two or more SLTSs, we define parallel product in such a way that an event can be executed only if it is contained in all the SLTSs involved. This will allow us to model multiple requirements of a system. For example, given LTS_1 and LTS_2 specifying certain given system requirements, an event is allowed to occur only if it is allowed in both SLTSs.

Definition 3.3. *Parallel Product*
Given two SLTSs $\mathcal{G}_{W_1} = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ *and* $\mathcal{G}_{W_2} = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$ *their parallel product is given by* $\mathcal{G}_{W_1} \times \mathcal{G}_{W_2} = (\mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \Gamma_1 \times \Gamma_2, \mathcal{S}_1^F \times \mathcal{S}_2^F)$ *such that the transition relation* $\Gamma_1 \times \Gamma_2$ *is defined as follows.*

- $(s_1, s_2) \xrightarrow{\alpha[g]} (s_1', s_2') \in \Gamma_1 \times \Gamma_2$, $\alpha \in \Sigma_1 \cap \Sigma_2$ *if* $s_1 \xrightarrow{\alpha[g]} s_1' \in \Gamma_1$ *and* $s_2 \xrightarrow{\alpha[g]} s_2' \in \Gamma_2$,
- *Undefined otherwise*

In the parallel product, the transitions of two SLTSs must always synchronize on shared events $\Sigma_1 \cap \Sigma_2$, where $\Sigma_1 = (\mathcal{I}_1 \cup \mathcal{O}_1 \cup \mathcal{A}_1)$ and $\Sigma_2 = (\mathcal{I}_2 \cup \mathcal{O}_2 \cup \mathcal{A}_2)$.

Analogous to the standard SCT where the legal language is a *sublanguage* of the plant, in our framework we use the notion of a simulation relation to describe the relationship between a system (plant) and a given specification, both modeled as SLTSs.

Definition 3.4. *Simulation Relation with Guards*
Given two SLTSs $\mathcal{G}_{W_1} = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ *and* $\mathcal{G}_{W_2} = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$, $\mathcal{G}_{W_2}$ *simulates* $\mathcal{G}_{W_1}$ *if there exists a relation* $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *such that* $\forall (s_1, s_2) \in R$, *if* $s_1 \xrightarrow{\alpha[g_1]} s_1' \in \Gamma_1$ *then* $\exists\, s_2', \exists\, g'$ *such that* $s_2 \xrightarrow{\alpha[g']} s_2' \in \Gamma_1$ *and* $g_1 \leq g'$ *and* $(s_1', s_2') \in R$.

That is, every transition taken by $\mathcal{G}_{W_1}$ can be matched by $\mathcal{G}_{W_2}$. In essence, when the two SLTSs are represented as their respective *execution trees* (i.e, a possible infinite unfolding of a given SLTS) often $\mathcal{G}_{W_1} \preceq \mathcal{G}_{W_2}$ means that $\mathcal{G}_{W_1}$ is a subtree of $\mathcal{G}_{W_2}$.

3.2. Service Representation

In our framework, we assume that Web services are described in WS-BPEL; based on this we automatically extract the SLTS models. Formally, we model a service as an SLTS as defined above. We also assume that the available Web services reside in a repository in which we select the required services that meet a given functionality. As defined above an event $\alpha \in \Sigma$ of an SLTS could be an input message, an output message, an atomic operation or an internal action.

Input and Output Messages: We denote a reception of a message as $?m(\overrightarrow{t})$ and an emission of message as $!m(\overrightarrow{t})$, where m is the name of the message also known as the message header, and t is the set of data parameters or variables. The symbols ? and ! are used to denote the direction of messages. Variables are local to a service and only one service can modify a variable.

Atomic Operations: Operations such as function invocations are denoted by *nameOperation(I::O)* with input parameters I and output parameters O. Atomic operations are indivisible functions that can modify the variables in a service. The atomic operations/functions that we consider here are similar to atomic processes defined in OWL-S and BPEL, which can access and modify the variables of a Web service. OWL-S defines an atomic process as a non-decomposable Web-accessible program. It is executed by a single (e.g., http) call, and returns a response. It does not require an extended conversation between the calling program or agent. We assume that these atomic functions are local to a given service. The atomic operation, e.g., $function(i_1, i_1 :: o_1, o_2)$ takes as inputs a set of variables $\{i_1, i_1\}$ and returns a set of variables $\{o_1, o_1\}$ as

output. The effects that the atomic operation has on its output variables are not visible to the entire system. It can be observed that there are many cases where it will not make sense to assume static information about Web services. In a dynamic environment, Web services information may change while the Web service procedure is operating at runtime. Typical examples are the following: whether a product is in stock, how much it will cost or how much has been bid for it, what the weather is like, what time a train or airplane will arrive, what seats are available for an airplane or a concert, what shipping facilities are available for a shipping request, all of which are unknown before runtime. The output of an atomic function in our model depends on the time and the circumstances of invocation. That is, the output of the atomic operation depend not only on the available input, but also on the current state of the whole system. In addition, we assume that the service providers keep details about the atomic operations secret. For example, if a service solves sophisticated routing problems, the service provider does not want the description of the service to reveal how the results are computed. Due to the nondeterministic nature of atomic operations, we treat them as black-box events.

We model input messages and atomic functions as uncontrollable since we cannot control inputs from the user. We assume that no service can deny an input action from other services, while it is completely up to the service to control its outputs. On the other hand, output messages from the system are modeled as controllable. In the context of Web services a guard (g) represents the preconditions on variables. Internal actions represent internal unobservable computations of a service but we will not model such behaviours in this paper. Figure 2 below shows the SLTS representations of the four component services of the **Flight Reservation and Purchase System** example introduced in Section 2.3.

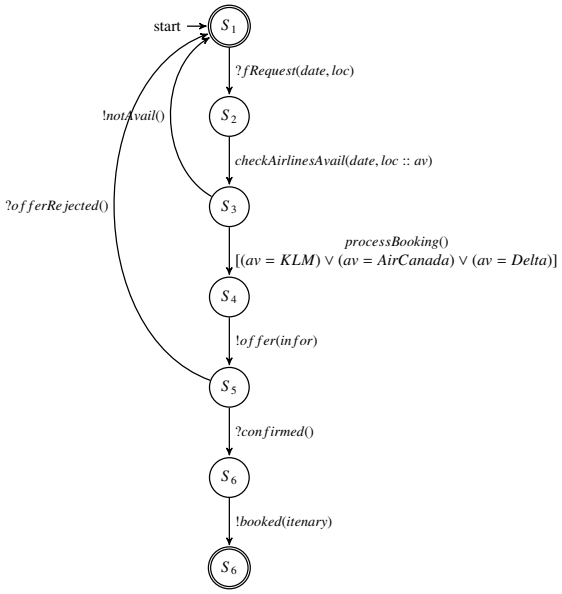
4. Supervisor Aware Service Composition Architecture (SASCA)

4.1. Controller Synthesis for Service Composition

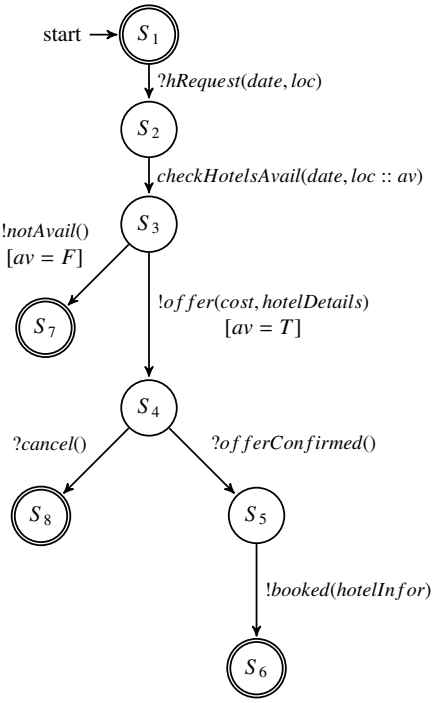
In this section, based on the representation of services using SLTSs, we formalize the problem of composing Web services, and we describe its solution by means of supervisory control theory of discrete-event systems. Our model of synthesized Web services relies deeply on message passing, interaction with data and actions. The composition problem that we consider here is as follows: given a set of available services $\mathcal{G}_{W_1}$, $\mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$ and a set of specifications $\mathcal{T}^W$ representing the goal (or desired) service over the same environment (same set of atomic actions), we would like to construct a controller C satisfying some controllability and nonblocking constraints which interacts with the available services to satisfy the specification $\mathcal{T}^W$. Thus, C serves as a controller that restricts the system in such a way that all its executions satisfy $\mathcal{T}^W$ and so that C is maximally permissive. In addition to requiring that the generated controller satisfies the controllability and nonblocking criteria, the controlled system is also free of errors that may result from communication among component services. We assume that both the available services and the goal service are expressed in SLTSs as defined above. Figure 3 shows the basic architecture diagram for the *SASCA* framework. The inputs to the system are the set of component Web services specified in WS-BPEL and the requirements are specified as SLTSs. We then provide a translator to generate the SLTSs representations from the WS-BPEL descriptions of the available services. The diagram shows the important internal representations from when the input enters the system to when a controller is generated. The framework also depicts an intermediate preprocessing step of the plant to achieve a more refined model suitable for composition synthesis. The final output of the synthesis is a WS-BPEL executable file. In the rest of this section, we discuss the core details of our approach including relevant definitions and theorems.

4.2. Asynchronous Communication and System to be Controlled

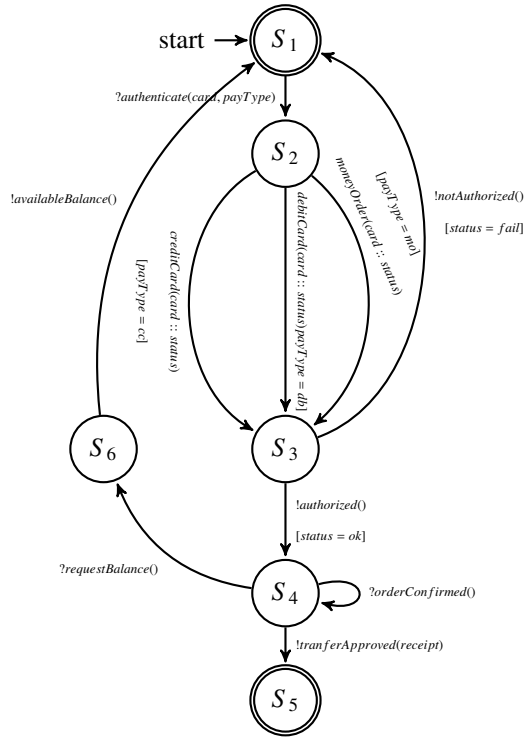
In our formalism, we use asynchronous communication to model the interaction among the available services. Synchronous semantics requires that during a message exchange, the sender and the receiver have



(a) The SLTS $\mathcal{G}_{W_1}$ of an Airline Service

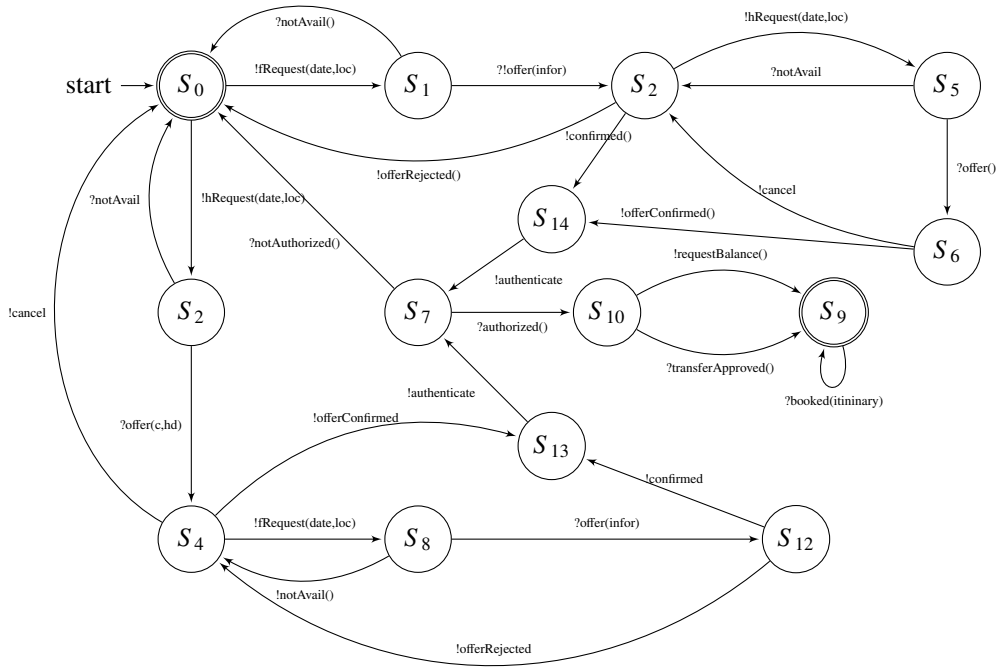


(b) The SLTS $\mathcal{G}_{W_2}$ of a Hotel Service



(c) The SLTS $\mathcal{G}_{W_3}$ of a Bank Service

Figure 2: Available Component Services for **Flight Reservation and Purchase System**



(d) The SLTS $\mathcal{G}_{W_4}$ of an On-line Interface Service

Figure 2: Available Component Services for **Flight Reservation and Purchase System**

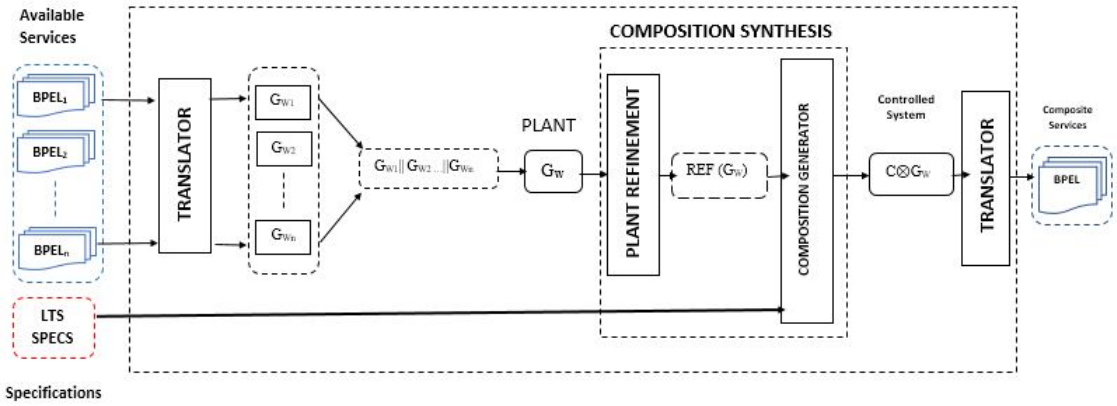


Figure 3: Supervisor Aware Service Composition Architecture (SASCA)

to synchronize the send and receive actions, and the sender blocks until a reply is received. However, in the domain of Web services where component services are dynamically discovered and plugged in to obtain a composite service (loosely coupled), using synchronous semantics may go a long way to limit the applicability of our model. Hence, in this paper we assume that Web services interact in an asynchronous fashion. Asynchrony can be achieved by employing unbounded memory to store the variables and parameters exchanged among component services. However, in this work the way we model service interactions does not take into consideration how the messages are stored and retrieved. Asynchrony eliminates the situation where the sender halts its process and wait for a reply from the receiver. The asynchronous semantics that we adopt here make implementation easier compared to synchronous semantics, however, it is very hard to reason about communication systems modeled using asynchronous semantics. In general, modeling the composition of communicating systems could result in various undesirable behaviours such as unspecified receptions and non-executable interactions of the system [58, 12]. We will refer to these undesirable properties (unspecified receptions and non-executable interactions) as *communication design errors*.

The framework we propose has two inputs as shown in Figure 3, the composition requirements $\mathcal{T}^W$ and the set of component Web services with SLTSs as $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$. The set of available services $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$ evolves independently, but together they form a combined system $\mathcal{G}_W$ whose behaviours we need to control. The individual component services cannot communicate among themselves; in order to exchange messages a controller is generated to mediate the interactions among component services. In the supervisory control domain, $\mathcal{G}_W$ models the plant which represents the set of possible behaviors. As a first step in the composition process we obtain $\mathcal{G}_W$ by combining the set of available services whose SLTSs is given by $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$ by means of an Asynchronous Parallel Composition which captures the notion of asynchronous communication [25].

Definition 4.1. *Asynchronous Parallel Composition*
Given two SLTSs $\mathcal{G}_{W_1} = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ *and* $\mathcal{G}_{W_2} = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$ *their asynchronous parallel composition is given by* $G_{W_1} \parallel G_{W_2} = (\mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \Gamma_1 \parallel \Gamma_2, \mathcal{S}_1^F \times \mathcal{S}_2^F)$ *such that the transition relation* $\Gamma_1 \parallel \Gamma_2$ *is defined as follows.*

- $(s_1, s_2) \xrightarrow{\alpha[g_1]} (s_1', s_2) \in \Gamma_1 \| \Gamma_2$, *if* $s_1 \xrightarrow{\alpha[g_1]} s_1' \in \Gamma_1$

- $(s_1, s_2) \xrightarrow{\alpha[g_2]} (s_1, s_2') \in \Gamma_1 \| \Gamma_2$, *if* $s_2 \xrightarrow{\alpha[g_2]} s_2' \in \Gamma_2$
- *Undefined otherwise*

Definition 4.1 can be extended to n services by observing that it is associative, i.e., $(\mathcal{G}_{W_1} \parallel (\mathcal{G}_{W_2}) \parallel \mathcal{G}_{W_3} = \mathcal{G}_{W_1} \parallel (\mathcal{G}_{W_2} \parallel \mathcal{G}_{W_3})$. Therefore, without ambiguity we can write $\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$ to represent the composition of multiple Web services. The definition of asynchronous parallel composition given above is defined so that the output from one service is consumed by the input of another service to produce an internal action such that the guards on both the output and input transitions are satisfied. In addition, we allow individual services to make independent moves. Definition 4.1 describes all possible behaviours of a given set of available services. We assume that the available services do not interact among themselves; any form of communication is through the supervisor. Hence, we require that the input (output) messages of a service are disjoint from the inputs (output) of another service. Atomic operations are local to a service. Generally, variables of a service have local scope and hence, each service refers to different internal variables. In cases where the name of variables conflicts among the components services, we assume appropriate relabeling of variables will be made to resolve the conflicts. Figure 4 illustrates Definition 4.1 with an example without guards on transitions. The SLTS in Figure 4(c) represents the asynchronous composition of SLTSs in Figure 4(a) and Figure 4(b).

Given a set of available services, forming the asynchronous parallel composition $\mathcal{G}_W$=$\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$ could result in communication errors. This leads us to the second step of our composition process in the next section.

4.3. Preprocessing Design Errors

Applying Definition 4.1 to combine the available services may result in two main communication design errors: cases where messages are sent to a service but it is unable to receive them and cases where a service expects a message which another service is unable to provide. That is, given the system to be controlled which represents the asynchronous parallel composition of available services $\mathcal{G}_W = \mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$, $\mathcal{G}_W$ may contain the following errors as defined below. We want the combined set of services to be free from *communication errors*. Consider for instance an airline service that provides several functionalities and is able to receive requests to book different kinds of flights from customers including *book_KML*, *book_Air_Canada*, *book_Ethiopian_airline* and so on. In this situation, the airline service may be providing more functionalities than what a particular client service actually needs. Therefore, it will be necessary to make the airline service cooperate with the client service by restricting its set of messages to a subset of the client's request.

An *unspecified reception* is a situation where one service can send a message at a reachable state, but other services are not able to receive it. That is, the SLTS description of a service contains an emission that cannot be consumed by the related component services involved in the composition. Consider Figure 5(a) and Figure 5(b), S_1 and S_2 can communicate based on the message headers *requestFlight* and *flightOffer*, however when S_2 is in state s_1, it is capable of sending an additional message *!searchFlight* which cannot be consumed by S_1. In the following, we formally define unspecified receptions.

Definition 4.2. *Unspecified-receptions*
Given an SLTS $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ *of a web service, a transition* $s_j \xrightarrow{!m(x)[g_j]} s_{j+1} \in \Gamma$ *is an unspecified reception, if for a given execution or run* $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \dots s_j \xrightarrow{!m(x)[g_j]} s_{j+1} \dots s_{n-1} \xrightarrow{\alpha_{n-1}[g_{n-1}]} s_n$ *of* $\mathcal{G}_W$ *there is no* $i > j$ *such that* $s_i \xrightarrow{?m(x)[g_i]} s_{i+1} \in \Gamma$.

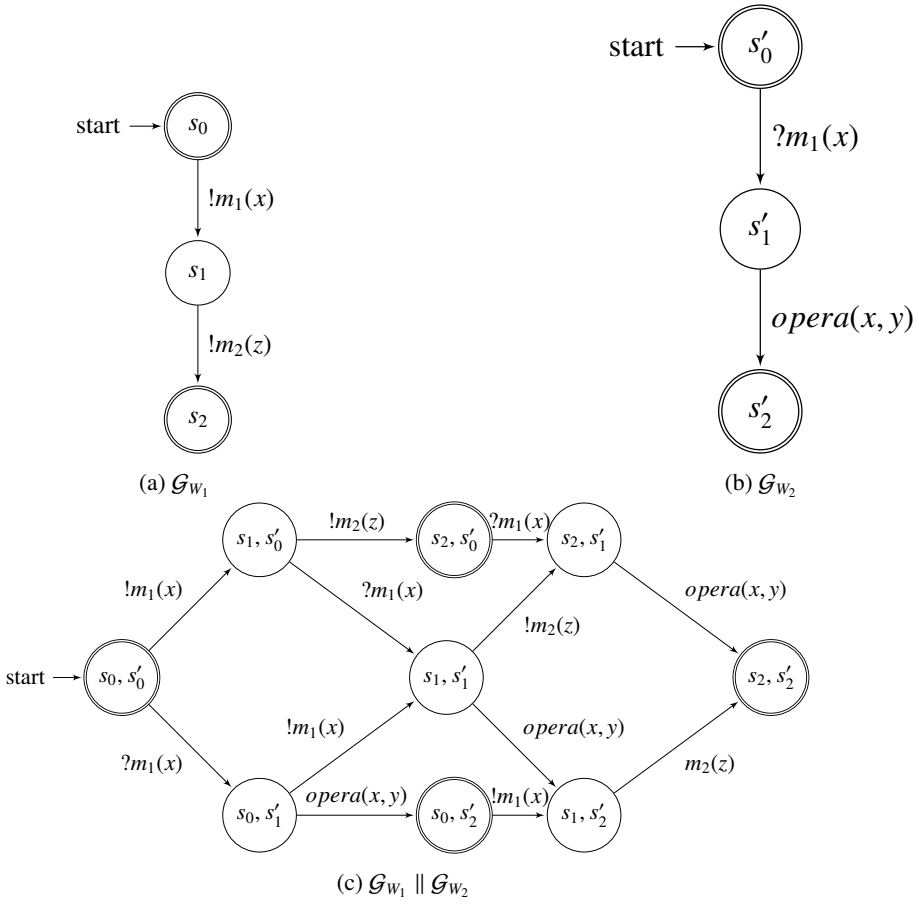


Figure 4: Asynchronous Parallel Product

Similarly, *non-executable interactions* refer to a situation in which one service is able to receive a message that has not already been sent by some other service. This results in additional unmatched receptions. For instance, in Figure 5, when S_1 and S_3 are combined by asynchronous parallel composition, the combined system will be stuck at a state in which S_1 will be waiting on the reception of *?flightOffer* at state s_1 while S_3 will be waiting on either the reception of *?searchFlight* or *?requestHotel* which is not being sent by any of these services.

Definition 4.3. *Non-executable interactions*
Given an SLTS $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ *of a Web service, a transition* $s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \in \Gamma$ *is non-executable interaction if for a given execution or run* $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \ldots s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \ldots s_{n-1} \xrightarrow{\alpha_{n-1}[g_{n-1}]} s_n$ *of* $\mathcal{G}_W$ *if there is no* $i < j$ *such that* $s_i \xrightarrow{!m(x)[g_i]} s_{i+1} \in \Gamma$.

That is, if there is a transition in the trace that expects an input message ($?m(\overrightarrow{x})$) but there is no corresponding output message ($!m(\overrightarrow{x})$) on a transition that precedes the input message.

Definition 4.4. *Communication-Error Free SLTSs*

An SLTS $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ is said to be communication-error free if it is free from unspecified receptions and non-executable interactions.

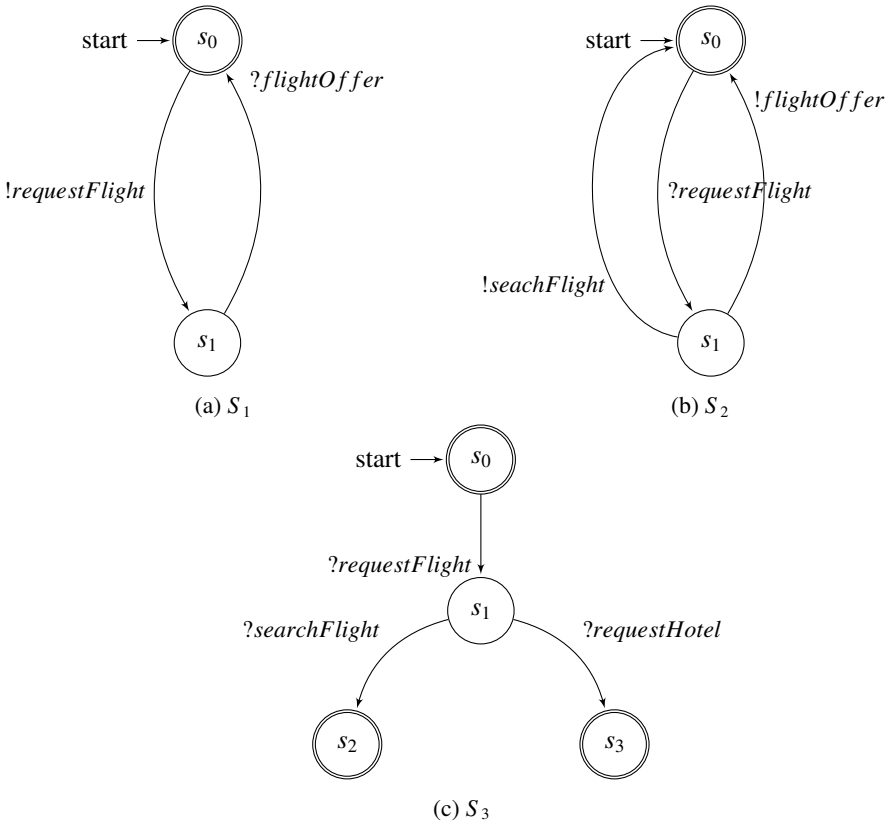


Figure 5: Unspecified-receptions and Non-executable interactions

In order to ensure freedom from *non-executable interactions* and freedom from *unspecified receptions* in the composition system, we perform a prefiltering step as part of our composition generation process to refine the system to be controlled. Given the SLTS $\mathcal{G}_W$ representing asynchronous parallel composition of the available services, we perform a refinement or preprocessing on $\mathcal{G}_W$ to get a communication-error free SLTS. We denote a plant which is a valid SLTS or the refined plant as Ref($\mathcal{G}_W$). This preprocessing step removes all paths that contain any unspecified and non-executable interaction *errors* from the original plant. This does not affect the functionality of the system, since we want to consider a communication-error free set of communicating services. That is, once the services are composed, input (output) messages that are not consumed by other services will become useless and may obstruct system progress.

4.4. Composition Requirements

The composition requirements are also given as SLTSs which specify the possible accepted interactions that must hold in the composition. We require that the system requirements to be satisfied be clearly specified in terms of its input/output messages and atomic operations that would be made available to other services.

We also assume that the SLTSs of the specification are also communication-error free by Definition 4.4. There could be multiple specifications. In that case, we use parallel product to put them together. Intuitively, a supervisor in this case will be one that guarantees that all specifications are achieved. We assume that the designer of the specification is aware of the set of services available and must specify the specification in such a way that it is simulated by the combined services. In the case that the specification cannot be simulated by the plant, then further refinement must be done. This refinement can be done manually or automatically. However, we do not discuss that in this paper.

In this framework, specification can take one of the following forms: (i) A composition requirement can specify a set of constraints on the ordering of events and actions. A typical example of these constraints in our flight booking system is that the credit card of the user must be verified by the Bank service before a booking confirmation is delivered to the customer. Another ordering requirement could be that the Flight service must confirm flight availability before the hotel is booked. In other words, the hotel should not be booked if the flight cannot be booked. A simple SLTS specification expressing this composition requirement is depicted in Figure 6(a). We assume that self loops would be used at certain states to indicate that other transitions are allowed to occur at those states. ii) Another form of composition requirement is to specify stronger guards that limit the values that can be taken by a variable or a data parameter from a given domain. This can be used to restrict the values of a variable that can be sent or received by services. In Figure 6(b), the SLTS specifies that the airline service from our running example can accept reservations for only KLM and Delta ($(av = KLM) \vee (av = Delta)$) but not Air Canada. This specification restricts the values of the variable av. Hence, a correct composition must not allow Air Canada reservations to be made. Figure 6(c) also specifies the kind of payment that can be made by a client to the Flight Reservation and Purchase System. The SLTS specifies that the system can only accept payment made by credit card or debit card. iii) One can also explicitly specify a set of forbidden states that the system should not reach during execution. For example, a specification that specifies that the cost of a product c should not exceed a limit m, i.e., $c < m$ implies that $c \geq m$ leads to an unsafe state.

4.5. Controller Synthesis

In this section, we study how to synthesize a controller that will ensure that the system's behaviour satisfies the given requirements. We assume that the system to be controlled is given by the asynchronous parallel composition of the available services $\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$ and the system requirements (target service) are given by $\mathcal{T}^W$. Now, we require that the asynchronous parallel composition of the available services simulates the goal services. That is, $\mathcal{T}^W \preceq \mathcal{G}_W$. In the case that it does not simulate the goal service we perform refinement on the target services. The following definition specifies the parallel product with refinement to safe (good) and forbidden (bad) states.

Definition 4.5. *Composition Refinement*
Given two SLTSs $\mathcal{G}_{W_1} = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ *and* $\mathcal{T}^W = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$ *representing the plant and the specification respectively, we can compute their parallel product as well as refine the plant with respect to the specification in such a way that the behaviour not allowed by the specification end in bad or forbidden states in the plant. A bad or forbidden state is a state reachable in* $\mathcal{G}_{W_1}$ *but not in* $\mathcal{T}^W$. *The composition refinement of the plant and the specification denoted by* $\mathcal{G}_{W_1} \times_{ref} \mathcal{T}^W$ *is given by* $\mathcal{C}^0 = (\mathcal{S}_1 \times (\mathcal{S}_2 \cup \{s^{Bad}\}), \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \Gamma_1 \times \Gamma_2, \mathcal{S}_1^F \times \mathcal{S}_2^F)$, *where* s^{Bad} *denotes a bad or forbidden state and the transition relation* $\Gamma_1 \times \Gamma_2$ *is defined as follows.*

- $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge g_2]} (s_1', s_2') \in \Gamma_1 \times \Gamma_2$ *and* $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge \neg g_2]} (s_1', s^{Bad}) \in \Gamma_1 \times \Gamma_2$, *if* $s_1 \xrightarrow{\alpha[g_1]} s_1' \in \Gamma_1$ *and* $s_2 \xrightarrow{\alpha[g_2]} s_2' \in \Gamma_2$

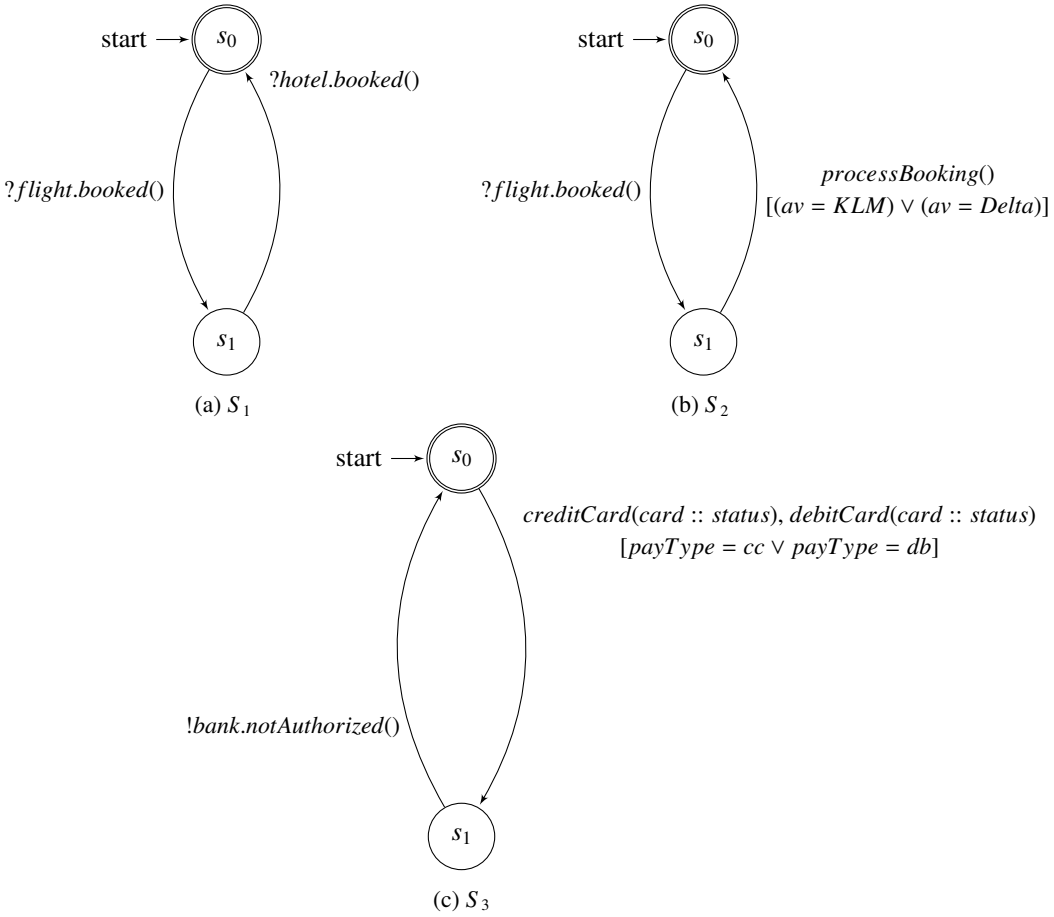


Figure 6: Composition Requirements for Flight Booking Example

- $(s_1, s_2) \xrightarrow{\alpha} (s_1', s^{Bad}) \in \Gamma_1 \times \Gamma_2$, *if* $s_1 \xrightarrow{\alpha} s_1' \in \Gamma_1$ *and* $s_2 \xrightarrow{\alpha} s_2' \notin \Gamma_2$
- *Undefined otherwise*

The first item of the definition creates two new transitions in C^0 (the refined SLTS) with the same events but different guards. Intuitively, The first transition given by $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge g_2]} (s_1', s_2')$ replaces the guards of C^0 with that of the specification and the resultant state is a state allowed by both the plant and the specification. The second transition given by $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge \neg g_2]} (s_1', s^{\text{Bad}})$ is essentially the same as the former, however the guard of the latter transition is $g_1 \wedge \neg g_2$ which results in a new state allowed by the plant but unsafe in the specification. The second item of the definition creates a new transition in C^0 if an event is allowed by the plant but not legal in the specification.

Now the set of states of C^0 is given by $Y{=}\mathcal{S}_1 \times (\mathcal{S}_2 \cup \{s^{\text{Bad}}\})$. A state $(s_1, s_2) \in Y$ is said to be forbidden if $s_2 = s^{\text{Bad}}$. That is, it is a bad state. We will denote $\mathcal{S}^{\text{Bad}}$ as the set of bad states of C^0. That is the set of states reachable in $\mathcal{G}_{W_1}$ but not in $\mathcal{T}^W$. The state $(s_1, s_2) \in Y$ is safe if $s_2 \neq s^{\text{Bad}}$. States that are not in $\mathcal{S}^{\text{Bad}}$ are called *safe or good states* denoted by $\mathcal{S}^{Good}_{C^0}$. Now, by strengthening the guards of C^0 with respect to the

plant so that forbidden states in C^0 are not reachable we obtain a new SLTS which we will call a *safe SLTS* of C^0. We will show how to strengthen the guards of C^0 later on in Algorithm 5 of Section 5.

We assume that the set of events Σ is partitioned into three disjoint subsets namely, *controllable* events $\Sigma_c \subseteq \Sigma$, *uncontrollable* $\Sigma_{uc} \subseteq \Sigma$ and *enforceable* events $\Sigma_f \subseteq \Sigma$. Controllable events can be disabled by the controller while uncontrollable events cannot be prevented from occurring. In addition, the enforceable events are special events that can be enforced by the controller. They are able to preempt both controllable and uncontrollable events at run-time but not static transitions. We do not assume any relationship between the set of controllable and enforceable events at this moment. The notion or the intent of control in this framework involves the following techniques. That is, the controller exert control as follows. Firstly, the generated controller prevents the system from firing or taking a particular path that violate the control requirement and secondly, it also prevents the system from reaching states designated as forbidden. In order to achieve the above control goals the supervisor enacts control based on the following three control criteria:

1. Disabling of controllable events on a transition (static transition)
2. Assignment of stronger guards to controllable transitions (transitions whose events are controllable)
3. Enforcement of enforceable events

To develop our control synthesis algorithms and strategies, we assume that the system evolves from one state to another based on the kind of transitions (static or dynamic transitions) at a given state. Thus, it is imperative to study the kind of transitions at a given state. We will explore the notion of control based on whether the transition is static or dynamic, or whether the values of the variable used on the transition can be tracked or not. Once, we have generated C^0 from the Definition 4.5, we will iteratively pare down C^0 until it satisfies the requirements.

Static Transition Case: Given a static transition (i.e., a transition with the trivial guard "true"), if this transition is associated with a controllable event which is allowed by the plant $\mathcal{G}_W$ but that violates system requirements, then we assume that this transition will be disabled by the supervisor. However, if the event associated with this transition is an uncontrollable event, then we must ensure that this static transition does not occur in the plant. For a static transition if the specification does not allow it, we will not allow the system to reach a state where it can occur.

Dynamic Transition Case (Dynamic Type 1 Transition): Let $\mathcal{G}_{W_1} = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ and $\mathcal{G}_{W_2} = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$ be two Web services, and suppose that the transition $t_1 = s_1 \xrightarrow{!m(v_1)[g_1]} s_1' \in \Gamma_1$ is an emission of a variable v_1 from G_{W_1} to G_{W_2} and $t_2 = s_2 \xrightarrow{?m(v_1)[g_2]} s_2' \in \Gamma_2$ is reception of v_1 by G_{W_2}. Now, if the variable v_1 (the content of the message $!m(v_1)$) has not been modified from its original value at the state where it was defined or last assigned until the state at which it is actually used, then the value of v_1 has not changed. This implies that we can easily track the value of the variable v_1 in the message from the service that sent it to the receiving service. Now, at the state that this variable is being used, if there is a condition on a transition (t_2) from this state that imposes a restriction on the set of values the variable can take, then we need to make sure that the guard on t_1 is never true for those values to prevent the system from reaching an illegal state. Hence, the supervisor can enact control by restricting the value of the variable at the sending service side before it would be received by the receiving service. The control strategy employed to deal with this kind of transition is to assign stronger guards to a controllable transition. Hence, we generate the guard $g_1 \wedge \neg g_2$ and attach it to the transition t_1. In our example above, the Bank service Figure 2(c) can accept a debit card, credit card or money order as a means of payment, but the specification in Figure 6(c) prevents a payment with money order. To satisfy this constraint, we put a condition on the transition of the service that will send a request for payment to the Bank service not to send a request for money order payment.

Dynamic Nondeterministic Transition Case (Dynamic Type 2 Transition): This case deals with atomic operations whose output value is unknown until runtime. This introduces an issue of nondeterminism into our model. Since the values of the variable are unknown until runtime we cannot treat this case in the same way as the previous case. During design time we will classify certain events (e.g., failure message events) as enforceable events. If such a transition does not exist we will introduce a new transition into the plant and the specification. To be able to prevent transitions that may cause a specification violation and that have guards containing outputs of atomic operations, we will rely on enforceable events to preempt uncontrollable events from happening when the output of the variable from atomic operations violates a specification. That is, if a failure could occur in the system due to values associated to atomic operations, then enforceable transitions are used to preempt the failure. This can be done by modification of the plant [15, 54]. Consider Figure 2(a), the transition from state S_2 to S_3 labeled with *checkAirlinesAvail*(*date*, *loc* :: *av*) has an output variable *av* which can take KLM, AirCanada or Delta as its values. The operation *checkAirlinesAvail*(*date*, *loc* :: *av*) is assumed to be black-box, so we do not know which value it will assign to *av*. Now, the specification in Figure 6(b) limits the values that *av* can take to only KLM and Delta. To ensure that the transition from state S_3 to S_4 in Figure 2(a) is never taken when the value of *av* is AirCanada, we mark the transition $S_3 \xrightarrow{!notAvail()} S_1$ in Figure 2(a) as a enforceable transition. The value of *av* is monitored so that this enforceable transition can be used to preempt other transitions at runtime when the value of *av* violates a specification.

Definition 4.6. *Controlled System*
Let $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ *be the an SLTS of a given plant, and let* $C = (\mathcal{S}_C, \mathcal{S}_C^0, \mathcal{I}_C, \mathcal{O}_C, \mathcal{A}_C, \Gamma_C, \mathcal{S}_C^F)$ *represent the controller of* $\mathcal{G}_W$. *The controlled system* $C \otimes \mathcal{G}_W$ *representing the behaviour of* $\mathcal{G}_W$ *when constrained (controlled) by* C *is given by* $C \otimes \mathcal{G}_W = (\mathcal{S}_C \times \mathcal{S}, \mathcal{S}_C^0 \times \mathcal{S}^0, \mathcal{I}_C \cup \mathcal{I}, \mathcal{O}_C \cup \mathcal{O}, \mathcal{A}_C \cup \mathcal{A}, \Gamma_C \times \Gamma, \mathcal{S}_C^F \times \mathcal{S}^F)$
where:

- $(s_1, s_2) \xrightarrow{m[g_1 \wedge g_2]} (s_1', s_2') \in \Gamma_C \times \Gamma$

$$\text{if } \begin{cases} s_1 \xrightarrow{!m[g_1]} s_1' \in \Gamma_C \text{ and } s_2 \xrightarrow{?m[g_2]} s_2' \in \Gamma, \\ \text{or} \\ s_1 \xrightarrow{?m[g_1]} s_1' \in \Gamma_C \text{ and } s_2 \xrightarrow{!m[g_2]} s_2' \in \Gamma, \end{cases}$$

- $(s_1, s_2) \xrightarrow{\alpha[g]} (s_1', s_2') \in \Gamma_C \times \Gamma$, $\alpha \in \mathcal{A}_c \cup \mathcal{A}$ *if* $s_1 \xrightarrow{\alpha[g]} s_1' \in \Gamma_C$ *and* $s_2 \xrightarrow{\alpha[g]} s_2' \in \Gamma$,
- *Undefined otherwise*

In other words, a transition is possible in the plant if it is also possible in the controller transition system, which implies that the guards are true and the transition can be fired.

Definition 4.7. *(Controlled System and Specification) Let m be the message header of the output message !m and the input message ?m. Let* $\mathcal{H}$ *be the set of message headers of a given SLTS. Given a controlled system* $C \otimes \mathcal{G}_W = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Omega_1, \mathcal{S}_1^F)$ *and a specification* $\mathcal{T}^W = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Omega_2, \mathcal{S}_2^F)$ *a simulation relation between* $C \otimes \mathcal{G}_W$ *and* $\mathcal{T}^W$ *is a relation* $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *such that* $\forall (s_1, s_2) \in R$,

- *if* $s_1 \xrightarrow{m[g]} s_1'$ *in* Ω_1 *and* $m \in \mathcal{H}$ *then* $\exists s_2', s_2 \xrightarrow{!m[g]} s_2'$ *in* Ω_2 *and* $(s_1', s_2') \in R$ *or* $\exists s_2', s_2 \xrightarrow{?m[g]} s_2'$ *in* Ω_2 *and* $(s_1', s_2') \in R$
- *if* $s_1 \xrightarrow{\alpha[g]} s_1'$ *in* Ω_1 *and* $\alpha \in \mathcal{A}_1$ *then* $\exists s_2', s_2 \xrightarrow{\alpha[g]} s_2'$ *in* Ω_2 *and* $(s_1', s_2') \in R$

4.6. Controllability

The original setting of supervisory control theory considers *language-based* controllability [14, 55] which assumes the underlying automata to be deterministic. Language-based controllability was subsequently extended to provide a stronger notion of controllability called *state-controllability* [31, 18, 59] for non-deterministic discrete-events systems. In this work, we rely on a notion similar to state-based controllability to capture the concept of controllability of SLTS in our model.

For a given SLTS $\mathcal{T}^W$ specification which is simulated by a given plant $\mathcal{G}_W$, a reachable state $(p^{\mathcal{G}_W}, q^{\mathcal{T}^W})$ in $\mathcal{G}_W \times \mathcal{T}^W$ is uncontrollable if the following holds. 1) If an uncontrollable transition labeled with α (static transition) can be fired from the state $p^{\mathcal{G}_W}$ in the plant but not from the state $(p^{\mathcal{G}_W}, q^{\mathcal{T}^W})$ in $\mathcal{G}_W \times \mathcal{T}^W$. 2) If an uncontrollable transition labeled with α and a guard g_1 can be fired in $\mathcal{G}_W$ at state $p^{\mathcal{G}_W}$ and this same uncontrollable transition labeled with α but a different guard g_2 is possible at $(p^{\mathcal{G}_W}, q^{\mathcal{T}^W})$ in the $\mathcal{G}_W \times \mathcal{T}^W$, then whenever g_1 evaluates to *true* for a given set of values of a variable v, g_2 does not always evaluate to *true* for the same values of v. That is, g_1 does not imply g_2. This implies that the uncontrollable transition at $(p^{\mathcal{G}_W}, q^{\mathcal{T}^W})$ leads to a *forbidden* state.

Let $s \xrightarrow{\delta}$ denote that there exists at least one state s' such that $s \xrightarrow{\delta} s'$ and denote $E_P^{\mathcal{S}}(s) = \{a \in \Sigma \mid s \xrightarrow{a}\}$ as the set of enabled *static transitions* of the state $s \in \mathcal{S}$ of the SLTS P. Similarly, let $s \xrightarrow{\delta[g]}$ denote that there exists a guard g and at least one state s' such that $s \xrightarrow{\delta[g]} s'$ and let $E_P^{\mathcal{D}}(s) = \{(a, g) \mid s \xrightarrow{a[g]}, a \in \Sigma, g \in G\}$ represent the set of *dynamic transitions* enabled at state s of the SLTS P.

Definition 4.8. *Controllability*
Given two SLTSs $\mathcal{G}_W = (\mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{A}_1, \Gamma_1, \mathcal{S}_1^F)$ *and* $\mathcal{T}^W = (\mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{A}_2, \Gamma_2, \mathcal{S}_2^F)$*, representing the plant and the specification, respectively, such that* $\mathcal{T}^W \preceq \mathcal{G}_W$*. A state* $(p, q) \in \mathcal{S}_1 \times \mathcal{S}_2$ *is controllable if the following holds:*

1. *Static Controllability:*

$$\forall \delta \in \Sigma_{uc} : \delta \in E^{\mathcal{S}}_{\mathcal{G}_W}(p) \implies \delta \in E^{\mathcal{S}}_{(\mathcal{G}_W \times \mathcal{T}^W)}((p, q))$$

2. *Dynamic Controllability:*

$$\forall \delta \in \Sigma_{uc} : (\delta, g_1) \in E^{\mathcal{D}}_{\mathcal{G}_W}(p) \implies [\exists g_2 : (\delta, g_2) \in E^{\mathcal{D}}_{(\mathcal{G}_W \times \mathcal{T}^W)}((p, q)) \wedge (g_1 \implies g_2)]$$
$$\textit{or } [\exists \delta', \exists g_3 : (\delta', g_3) \in E^{\mathcal{D}}_{(\mathcal{G}_W \times \mathcal{T}^W)}((p, q)) \textit{ and } \delta' \in \Sigma_f]$$

A plant $\mathcal{G}_W$ is said to be *state controllable with respect to* $\mathcal{T}^W$if all reachable states of $\mathcal{G}_W \times \mathcal{T}^W$ are state controllable. According to Definition 4.8, uncontrollable transitions that are enabled in the reachable states of the plant state q by following the same trace, must also be enabled at the corresponding reachable state (p, q) of $\mathcal{G}_W \times \mathcal{T}^W$. Thus, we say $\mathcal{G}_W$ is controllable if: 1) δ is uncontrollable and δ is the current static transition event enabled in $\mathcal{G}_W$ implies that δ is also enabled at the corresponding state of $\mathcal{G}_W \times \mathcal{T}^W$. 2) δ is a dynamic uncontrollable event and a guard g_1 is possible at the current state of $\mathcal{G}_W$ then, it implies that there exists a corresponding uncontrollable dynamic transition and a guard g_2 in $\mathcal{G}_W \times \mathcal{T}^W$ such that g_2 is true only if g_1 is true, or there exists an enforceable transition δ' that can preempt any uncontrollable transition in the current enabled state.

Now, let $g_n^A(v)$ denote a guard g_n with a variable v whose values depend on an output of an atomic operation A of a given transition system (e.g., in Figure 7, $A = func_1(x, y), v = y, g_n = (y < 10)$). We state the following corollary as a consequence of controllability of a given plant and a specification. Here we will assume that a guard depends on only one variable.

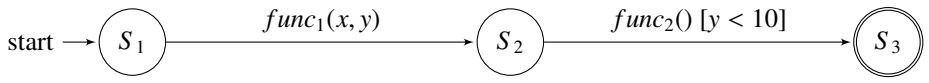


Figure 7

Corollary 4.1 below states that given a plant $\mathcal{G}_W$ and a specification $\mathcal{T}^W$, if a dynamic type 2 transition, say T, is enabled at a state p of the plant $\mathcal{G}_W$ but not at a corresponding state (p, q) of the specification $\mathcal{T}^W$ then, for the state (p, q) to be state controllable it implies that there must exist an enforceable transition also enabled at (p, q) to preempt T.

Corollary 4.1. *Let* $\mathcal{G}_W = (\mathcal{S}_{\mathcal{G}_W}, \mathcal{S}^0_{\mathcal{G}_W}, \mathcal{I}_{\mathcal{G}_W}, O_{\mathcal{G}_W}, \mathcal{A}_{\mathcal{G}_W}, \Gamma_{\mathcal{G}_W}, \mathcal{S}^F_{\mathcal{G}_W})$ *and* $\mathcal{T}^W = (\mathcal{S}_{\mathcal{T}^W}, \mathcal{S}^0_{\mathcal{T}^W}, \mathcal{I}_{\mathcal{T}^W}, O_{\mathcal{T}^W}, \mathcal{A}_{\mathcal{T}^W}, \Gamma_{\mathcal{T}^W}, \mathcal{S}^F_{\mathcal{T}^W})$ *denote a plant and its specification, respectively, and let* $(p, q) \in \mathcal{S}_{\mathcal{G}_W} \times \mathcal{S}_{\mathcal{T}^W}$. *If* $\exists\, \delta \in \Sigma_{uc}, \exists\, g^A(v) \in G : (\delta, g^A(v)) \in E^D_{\mathcal{G}_W}(p)$ *but* $\nexists (g^A(v))' : (\delta, (g^A(v))') \in E^D_{\mathcal{T}^W}(p, q)$, *and* $g^A(v) \wedge (g^A(v))'$ *satisfiable, then if* (p, q) *is state controllable then there must exists* $\delta' \in \Sigma_f$ *and a guard* g'' *such* $(\delta', g'') \in E^D_{\mathcal{T}^W}(p, q)$.

Proof. The proof follows from the second part of definition 4.8 where the transition is a dynamic type 2 transition with guard $g^A_n(v)$. ■

Example 4.1. *Illustration of Corollary 4.1*

Consider the plant service $\mathcal{G}_W$ *in Figure 8 and the target service* $\mathcal{T}^W$ *of Figure 9.* $T = S_2 \xrightarrow{func_2()[y<10]} S_3 \in \Gamma_{\mathcal{G}_W}$ *is a dynamic type 2 transition whose guard* $(y < 10)$ *has a variable whose values depends on the output of the atomic operation* $funct_1(x, y)$.

Since $\mathcal{T}^W$ *does not allow* T *at state* S_2, *if* $\mathcal{G}_W$ *is state controllable with respect to* $\mathcal{T}^W$ *then there must exist an enforceable transition* T' *also enabled at state* S_2 *which is given by* $S_2 \xrightarrow{!failMesg()} S_4 \in \Gamma_{\mathcal{T}^W}$ *to preempt* T *at runtime.*

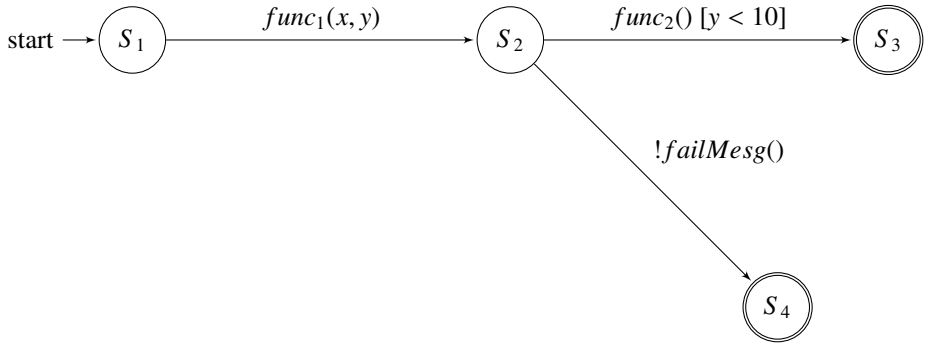


Figure 8: Plant Service $\mathcal{G}_W$

The control solution that we seek in our approach requires that the system does not reach a state from which the only exiting transitions lead to unsafe states. We formalize this in the following definition.

Definition 4.9. *Non-blocking*

An SLTS $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, O, \mathcal{A}, \Gamma, \mathcal{S}^F)$ *is non-blocking if* $s \xrightarrow{s[g_i]} s' \in \Gamma \Rightarrow \exists\, \delta \in \Sigma^*$ *and a guard* g_j *such that* $s' \overset{\delta[g_j]}{\Longrightarrow} s''$ *and* $s'' \in \mathcal{S}^F \backslash \mathcal{S}^{Bad}$, *where* $\mathcal{S}^{Bad}$ *is the set of unsafe (bad) states.*

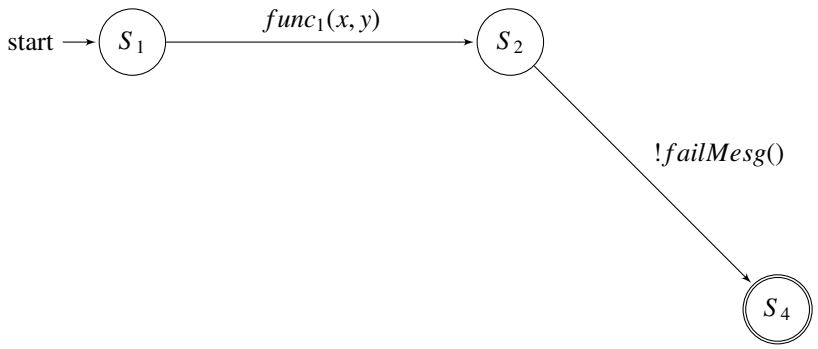


Figure 9: Target Service $\mathcal{T}^W$

A controller is minimally restrictive in the sense that it only disallows transitions that must be disallowed. It is natural to require that a controller restricts the plant as little as possible. We formalize this qualitative property in the following definition using the pre-order notion implied by simulation relation.

Definition 4.10. *Minimally Restrictive Controller*
Given a plant $\mathcal{G}_W$ and a specification $\mathcal{T}^W$, a controller C for $\mathcal{G}_W$ is minimally restrictive if there does not exist a controller C' for $\mathcal{G}_W$ such that $C \otimes \mathcal{G}_W \preceq C' \otimes \mathcal{G}_W$.

The composition problem that we consider is as follows. Given a set of available services $\mathcal{G}_{W_1}$,$\mathcal{G}_{W_2}$,...,$\mathcal{G}_{W_n}$ and a set of specifications $\mathcal{T}^W$ representing the goal service over the same environment (same set of messages and atomic actions), we would like to construct a controller C such that the $C \otimes (\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n})$ is simulated by $\mathcal{T}^W$ satisfying some controllability and nonblocking constraints. Thus, C serves as a controller that interacts with the uncontrolled system in such a way that all its executions satisfy $\mathcal{T}^W$ and that C is maximally permissive. That is, we seek to generate an SLTS which interacts with the system $\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$ to satisfy the specification $\mathcal{T}^W$. In addition to requiring that the generated controller satisfies controllability and nonblocking criteria, the controlled system is also free of errors that may result from communication among component services. We formalize this in the following problem statement.

Definition 4.11. *Composition Problem*
Let $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$ be a set of SLTSs and let $\mathcal{T}^W$ be the composition requirements. The composition problem is to find a non-blocking, communication-error free and minimally restrictive controller C such that $C \otimes (\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}) \preceq \mathcal{T}^W$.

The definition implies that the controller constrains the plant such that every transition that can be taken by the controlled system $C \otimes (\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n})$ can also be taken in the specification. The intuition is that controllability will be necessary and sufficient to solve the composition problem as formulated in Section 6.

5. Composition Synthesis Algorithm

In this section, we provide the core details of our approach by presenting a set of algorithms that can be used to generate a composition. The composition generation technique proposed in our framework is an incremental process. Algorithm 1 presents a step-by-step process that can be used to build a controllable, non-blocking and communication error free controller. The algorithm takes the set of available component

services and a goal service specifying the functional requirements as inputs. The algorithm first refines the plant $\mathcal{G}_W$ by removing communication design errors (*non-executable interactions* and *unspecified receptions*) which is given by Line 2. Once the plant has been transformed into its communication-error free SLTS form, we check whether the target service is simulated by the plant $\mathcal{T}^W \preceq \mathcal{G}_W$ (Line 3). The function `simulationCheck`$(\mathcal{G}_W, \mathcal{T}^W)$ is an implementation of the simulation relation given in Definition 3.4. The function `simulationCheck`$(\mathcal{G}_W, \mathcal{T}^W)$ takes $\mathcal{G}_W$ and $\mathcal{T}^W$ as inputs, and checks if $\mathcal{G}_W$ simulates $\mathcal{T}^W$. It returns "true" if $\mathcal{G}_W$ simulates $\mathcal{T}^W$ and false if $\mathcal{G}_W$ does not simulate $\mathcal{T}^W$. In the case that a simulation relation exists between the plant and the target service, we then make adjustments to the plant to include a special event that will enable the controller to enforce enforceable events at runtime. This is given at Line 4 of Algorithm 1. The algorithm then computes the composition refinement (given by Definition 4.5) of the plant and the target service to get a new SLTS C^0 upon which further minimization steps will be performed (Line 5).

The next step of the algorithm (`repeat until` loop Lines 7-12) then performs various reductions on C^0. Line 9 performs static controllability minimization which is given by Algorithm 2. Line 10 eliminates blocking states and states from which only bad states are reachable. In Line 11 of the algorithm we perform dynamic controllability minimization and generate stronger guards to ensure that all executions of the C^0 lead to safe states. Lines 7-12 performs a fixed point computation on C^0 and terminates when a fixed point is reached (i.e., if $C^k == C^{k-1}$). Finally, we refine C^K by removing communication errors in each trace. To ensure that the generated controller is able to communicate with the available services, we reverse the direction of the messages of C^k in Line 14. This implies that an input message say *?m(x)* in C^k will become an output message *!m(x)* and vice versa. The message header *m* does not change, it is only the directions of the message that change. In the event that the algorithm does not find a simulation relation between the plant and the target service we iteratively refine the target service until a simulation relation is found.

Algorithm 1 Composition Synthesizer (Controller)

Input: The SLTS representing synchronous parallel product of the available services ($\mathcal{G}_{W_1} \parallel \mathcal{G}_{W_2} ... \parallel \mathcal{G}_{W_n}$) given by $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ and a target service $\mathcal{T}^W = (\mathcal{S}_{\mathcal{T}^W}, \mathcal{S}^0_{\mathcal{T}^W}, \mathcal{I}_{\mathcal{T}^W}, \mathcal{O}_{\mathcal{T}^W}, \mathcal{A}_{\mathcal{T}^W}, \Gamma_{\mathcal{T}^W}, \mathcal{S}^F_{\mathcal{T}^W})$
Output: communication-error free, nonblocking and controllable Controller

1: **procedure** COMPOSER($\mathcal{G}_W, \mathcal{T}^W$)
2: $\mathcal{G}_W \leftarrow$ removeCommunicationErrors($\mathcal{G}_W$)
3: **if** simulationCheck($\mathcal{G}_W, \mathcal{T}^W$) **then**
4: $\mathcal{G}_W \leftarrow$ plantAndSpecAdjustment($\mathcal{G}_W$), $\mathcal{T}^W \leftarrow$ plantAndSpecAdjustment($\mathcal{T}^W$)
5: $C^0 \leftarrow \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$
6: $k \leftarrow 0$
7: **repeat**
8: $k \leftarrow k + 1$
9: $C^k \leftarrow$ staticControllability($\mathcal{G}_W, C^{k-1}$)
10: $C^k \leftarrow$ unsafeStateMinimization(C^k)
11: $C^k \leftarrow$ dynamicControllabilityAndGuardGeneration($\mathcal{G}_W, C^k$)
12: **until** $C^k == C^{k-1}$
13: $C^k \leftarrow$ removeUnsafeState(C^k)
14: $C^k \leftarrow$ removeCommunicationErrors(C^k)
15: $C \leftarrow$ reverseMessageDirection(C^k)
16: **else if** (refineTargetToSimulatePlant($\mathcal{G}_W, \mathcal{T}^W$)$\neq \emptyset$) **then**
17: $\mathcal{T}^W \leftarrow$ refineTargetToSimulatePlant($\mathcal{G}_W, \mathcal{T}^W$)
18: Go to step 4.
19: **else**
20: **return** *null*
21: **end if**
22: **return** C
23: **end procedure**

Algorithm 2 converts a given SLTS into its communication-error free form as given in Definition 4.2 and Definition 4.3. The input to this algorithm is the asynchronous parallel composition of the available services. Algorithm 2 traverses the SLTS to eliminate *unspecified receptions* and *non-executable interactions*. Line 3-9 checks every run of the given SLTS for *unspecified receptions* and removes it. Similarly, Line 12-19 deals with *non-executable interactions*.

Algorithm 3 constructs a static controllable SLTS and iteratively creates new transitions that lead to bad states from a given SLTS. The input to this algorithm is the plant, and the composition refinement of the plant and the target service C^k. In the first iteration of the `repeat until` loop C^k is given by C^0.

Algorithm 2 removeCommunicationErrors

Input: $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$
Output: communication-error free SLTS of $\mathcal{G}_S$

1: **procedure** REMOVECOMMUNICATIONERRORS($\mathcal{G}_W$)
2: | * Removal of unspecified receptions */
3: | **for each** transition, $t = s_j \xrightarrow{!m(x)[g_j]} s_{j+1} \in \Gamma$ **do**
4: | | **for each** run $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} ...s_j \xrightarrow{!m(x)[g_j]} s_{j+1} \ldots$ in $\mathcal{G}_W$ **do**
5: | | | **if** $\nexists\, t' : t' = s_i \xrightarrow{?m(x)[g_i]} s_{i+1} \in \Gamma$ with $j < i$ **then**
6: | | | | Eliminate(r) * removes an the execution r from a set of executions */
7: | | | **end if**
8: | | **end for**
9: | **end for**
10: | * Removal of non-executable interactions */
11: | **for each** transition, $t = s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \in \Gamma$ **do**
12: | | **for each** run $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \ldots s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \ldots$ in $\mathcal{G}_W$ **do**
13: | | | **if** $\nexists\, t' : t' = s_i \xrightarrow{!m(x)[g_i]} s_{i+1} \in \Gamma$ with $i < j$ **then**
14: | | | | Eliminate(r) * removes an the execution r from a set of executions */
15: | | | **end if**
16: | | **end for**
17: | **end for**
18: | **return** $\mathcal{G}_W$
19: **end procedure**

The set of states of C^k is partitioned into safe states $\mathcal{S}^{Good}_{C^k}$ and bad states $\mathcal{S}^{Bad}_{C^k}$. For a given state p in $\mathcal{G}_W$ and a corresponding state in $(p, q) \in C^k$, if a static uncontrollable transition is enabled at state p but not in (p, q) (this is given by the first and second `for` loops), first the algorithm creates a new bad state $s^{\text{Bad}} \in \mathcal{S}_{C^k}$ and all dynamic transitions leading to (p, q) are diverted to s^{Bad} (Lines 6-10). This keeps the structure of dynamic transitions. Second, the uncontrollable state is eliminated including all outgoing transitions (Lines 11-13). Finally, unreachable states and associated transitions are also eliminated at Line 14. On the other hand, in Lines 19-23 of the algorithm, if there is a controllable transition enabled at state p of $\mathcal{G}_W$ but not in (p, q) of C^k then Line 21 of the algorithms marks this transition as disabled.

Algorithm 3 Static Controllability Minimization

Input: $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ and $C^k = (\mathcal{S}_{C^k}, \mathcal{S}^0_{C^k}, \mathcal{I}_{C^k}, \mathcal{O}_{C^k}, \mathcal{A}_{C^k}, \Gamma_{C^k}, \mathcal{S}^F_{C^k})$

Output: Static controllable SLTS of C^k, I.e., this algorithm produces an SLTS C^k such that all state of $\mathcal{G}_W$ are state controllable with respect to C^k

1: **procedure** staticControllability($\mathcal{G}_W, C^k$)
2: Let $\mathcal{S}_{C^k} = \mathcal{S}^{Good}_{C^k} \cup \mathcal{S}^{Bad}_{C^k}$, where $\mathcal{S}^{Good}_{C^k}$ is the set of safe states and $\mathcal{S}^{Bad}_{C^k}$ is the set of unsafe states
3: **for each** state $p \in \mathcal{S}$ and a corresponding state $(p, q) \in \mathcal{S}_{C^k}$ **do**,
4: **for each** static transition in Γ with an event $t \in \Sigma_{uc}$ such that $t \in E^{\mathcal{S}}_{\mathcal{G}_W}(p)$ **do**
5: **if** $t \notin E^{\mathcal{S}}_{C^k}((p, q))$ **then**
6: **for all** state $b \in \mathcal{S}_{C^k}$ with a dynamic transition $t' = b \xrightarrow{e[g]} (p, q)$ **do**
7: Create a new state s^{Bad} in C^k such that $z = b \xrightarrow{e[g]} s^{\text{Bad}}$, where s^{Bad} is an unsafe state
8: $\mathcal{S}^{Bad}_{C^k} \leftarrow \mathcal{S}^{Bad}_{C^k} \cup \{s^{\text{Bad}}\}$
9: $\Gamma_{C^k} \leftarrow \Gamma_{C^k} \cup \{z\}$
10: **end for**
11: Eliminate all transitions associated to the state (p, q)
12: and update the set of transitions Γ_{C^k} accordingly
13: $\mathcal{S}_{C^k} \leftarrow \mathcal{S}_{C^k} \backslash \{(p, q)\}$
14: $C^k \leftarrow$ *unreachableStateTransitionMinimization*(C^k) `\* remove unreachable states and associated transitions */`
15: **end if**
16: **end for**
17: **for each** static transition in Γ with an event $t \in \Sigma_c$ such that $t \in E^{\mathcal{S}}_{\mathcal{G}_W}(p)$ **do**
18: **if** $t \notin E^{\mathcal{S}}_{C^k}((p, q))$ **then**
19: $E^{\mathcal{S}}_{\mathcal{G}_W}(p) \backslash \{t\}$ `\* disable t */`
20: **end if**
21: **end for**
22: **end for**
23: **return** C^k
24: **end procedure**

Algorithm 4 presents a minimization technique to deal with unsafe states and blocking states. This algorithm takes an SLTS C^k as its input. C^k is assumed to be the SLTS obtained after some iterations of the `repeat Until` loop in Algorithm 1 (Lines 7-11). Specifically C^k is the output of Algorithm 3. The iteration of the first `for loop` statement collects and stores all states from which no marked state is reachable or from which only bad states can be reached (Lines 6-9). The algorithm stores these states in the buffer *BlockandUnsafe*. For each state in *BlockandUnsafe*, Lines 12-16 of the algorithm create a new bad state s^{Bad} and assign any dynamic transition that leads to a state in *BlockandUnsafe* to s^{Bad}. This is done to preserve the structure of dynamic transitions as done in Algorithm 3. Finally, Lines 18-24 eliminate all states and transitions collected at Lines 6-10. That is, all states in *BlockandUnsafe* and all associated transitions that lead to a state in *BlockandUnsafe* are removed.

Algorithm 4 Unsafe State Minimization

Input: $C^k = (\mathcal{S}_{C^k}, \mathcal{S}^0_{C^k}, \mathcal{I}_{C^k}, \mathcal{O}_{C^k}, \mathcal{A}_{C^k}, \Gamma_{C^k}, \mathcal{S}^F_{C^k})$
Output: non-blocking SLTS

procedure UnsafeStateMinimization(C^k)
  Let $\mathcal{S}_{C^k} \leftarrow \mathcal{S}^{Good}_{C^k} \cup \mathcal{S}^{Bad}_{C^k}$, where $\mathcal{S}^{Good}_{C^k}$ is the set of safe states and $\mathcal{S}^{Bad}_{C^k}$ is the set of unsafe states
  Let $\Sigma \leftarrow \mathcal{I}_{C^k} \cup \mathcal{O}_{C^k} \cup \mathcal{A}_{C^k}$
  Let BlockandUsafe = $\emptyset$
  `\* The following for loop collects and store all the states that lead to blocking */`
  for each state $s \in \mathcal{S}_{C^k}$ **do**,
    if $\nexists\, \delta \in \Sigma$, $\nexists\, g \in G$ and $\nexists\, s' \in \mathcal{S}^F \backslash \mathcal{S}^{Bad}_{C^k}$ such that $s \overset{\delta[g]}{\Longrightarrow} s'$ **then**
      BlockandUsafe $\leftarrow$ BlockandUsafe $\cup\{(s)\}$
    end if
  end for
  `\* The following for loop creates new bad states */`
  for all state $b \in \mathcal{S}_{C^k} \backslash \mathcal{S}^{Bad}_{C^k}$ with a dynamic transition such that $b \xrightarrow{e[g']} q$ and $q \in$ BlockandUsafe **do**
    Create a new state $s^{\text{Bad}} \in \mathcal{S}^{Bad}_{C^k}$ in C^k such that $z = b \xrightarrow{e[g']} s^{\text{Bad}}$
    $\mathcal{S}^{Bad}_{C^k} \leftarrow \mathcal{S}_{C^k} \cup \{s^{\text{Bad}}\}$
    $\Gamma_{C^k} \leftarrow \Gamma_{C^k} \cup \{z\}$
  end for
  `\* We eliminate all states and associated transitions in BlockandUsafe */`
  for All $q \in$ BlockandUsafe **do**
    for All $t \in \Gamma_{C^k}$, such that $t = q \xrightarrow{\alpha[g_1]}$ or $\exists\, b$ such that $t = b \xrightarrow{\alpha[g_2]} q$ **do**
      `\* eliminate all transitions associated with` q `*/`
      $\Gamma_{C^k} \leftarrow \Gamma_{C^k} \backslash \{t\}$
      $\mathcal{S}_{C^k} \leftarrow \mathcal{S}_{C^k} \backslash \{q\}$
    end for
  end for
  return C^k
end procedure

Algorithm 5 presents an approach that can be used to compute a safe and dynamic controllable SLTS of a given system. This algorithm implements the second part of the definition of controllability given in Definition 4.8. It involves the generation and attachment of stronger guards to transitions and the collections of variables to be monitored at runtime as well as the removal of dynamic uncontrollable states and transitions. In addition, controllable dynamic transitions that lead to bad states are disabled.

First, we assume that the set of variables is partitioned into trackable and non-trackable variables. Trackable variables (which we will call deterministic variables because their occurrence is deterministic) are those variables whose values do not change from where they were declared to where they are being used, whereas non-trackable (which we will call nondeterministic variables because their occurrence is nondeterministic) variables are those whose values we cannot predict from when they were declared to when they are used. Specifically, trackable variables are associated with dynamic type 1 transitions while non-trackable variables are associated with dynamic type 2 transitions. Non-trackable variables are the output of atomic operations.

Now, the algorithm starts by collecting all transitions that lead to bad states from a given state (Lines 13-27). In these steps we keep track of transitions that lead to a bad state based on the evaluation of nondeterministic variables (Lines 14-19), this is given by the first `if` statement. The `else` statement after the `if` statement keeps track of transitions that lead to an unsafe states due to the evaluation of deterministic variables (Lines 20-27). The next step of the algorithm strengthens the guards of each transition (Lines 30-38). Now, given that the value of deterministic variables can be tracked implies that we can trace back to where it was originally defined from where it is being used in order to strengthen the guard. Given a transition which leads to a bad state due to deterministic variable ($z = s_x \xrightarrow{\alpha[g_i(d)]} s_{x+1}$), we check every run of C^k to locate where it was declared first ($s_j \xrightarrow{!m(d)[g]} s_{j+1}$) and then we strengthen the guards which is given by taking the conjunction of the current guard on the transition and the negation of the guard of where it is being used ($t = s_j \xrightarrow{!m(d)[g_j \wedge \neg g_i(d)]} s_{j+1}$). Lines 40-53 of the algorithm checks every state that has a transition that leads to a bad state due to nondeterministic variables for enforceable transitions. In the case that an enforceable transition is enabled at this state, we save the variable for runtime monitoring (Lines 42-44). The runtime monitoring involves equipping the generated controller with additional capability to be able to track a given variable for certain values and then trigger certain actions based on the values of this values .On the other hand, if there is no enforceable transition enabled at this state, the algorithm (Line 46) first creates a new state in $\mathcal{S}_C^{Bad}$ and diverts all dynamic transitions to it as done in Algorithm 2. Next we completely eliminate the entire state and all transitions associated with this state from C^k. Finally, Line 52 disables all dynamic controllable transitions (both dynamic type 1 and dynamic type 2 transitions) that are enabled at the plant state but not at the corresponding state of C^k.

Algorithm 5 Dynamic Controllability and Guard Generation

Input: $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$ and $C^k = (\mathcal{S}_{C^k}, \mathcal{S}^0_{C^k}, \mathcal{I}_{C^k}, \mathcal{O}_{C^k}, \mathcal{A}_{C^k}, \Gamma_{C^k}, \mathcal{S}^F_{C^k})$

Output: A safe and dynamic controllable SLST of C^k with respect to $\mathcal{G}_W$

procedure DYNAMICCONTROLLABILITYANDGUARDGENERATION($\mathcal{G}_W$, C^k)
- Let $BP \leftarrow \emptyset$ be the set of bad state predicate transitions with trackable variables
- Let $BS \leftarrow \emptyset$ be the set of states from which a bad state is reachable based on trackable variables
- Let $BPN \leftarrow \emptyset$ be the set of bad state predicate transitions with non-trackable variables
- Let $BSN \leftarrow \emptyset$ be the set of states from which a bad state is reachable based on non-trackable variables
- Let $SP \leftarrow \emptyset$ be the set of safe state predicate
- Let $g_i(d)$ be a guard which depends on a variable $d \in V$
- Let $Det \subseteq V$ be the set of deterministic variables
- Let $nonDet \subseteq V$ be the set of non deterministic variables (i.e., variables whose values depends on the output of atomic operations)
- Let $i \leftarrow 0$, $j \leftarrow 0$
- Let *runtimeVariables* be the set of variables to be monitored at runtime
- **for All** $s_x \in \mathcal{S}_{C^k}$ **do**
  - **for each** $\delta \in \Sigma_{uc}$ and $g_i(d) \in G$ enabled at state s_x (i.e., $(\delta, g_i(d)) \in E^{\mathcal{D}}_C(s_x)$) such that $t = s_x \xrightarrow{\delta[g_i(d)]} s_{x+1}$ in C^k **do**,
    - **if** $s_{x+1} \in \mathcal{S}^{Bad}_{C^k}$ **then**
      - **if** $d \in nonDet$ is the output variable of the atomic operation **then**
        - $BPN \leftarrow BPN \cup \{t\}$
        - $BSN \leftarrow BSN \cup \{s_x\}$
      - **end if**
    - **else**
      - **if** $d \in Det$ **then**
        - $BP \leftarrow BP \cup \{t\}$
        - $BS \leftarrow BS \cup \{s_x\}$
      - **end if**
    - **end if**
    - $i \leftarrow i + 1$
  - **end for**
- **end for**
- `\* Guard propagation and Attachment */`
- **for each** $s_x \in BS$ **do**
  - **for each** $z \in BP$, such that $z = s_x \xrightarrow{\alpha[g_i(d)]} s_{x+1} \in BP$ **do**
    - **for each** $t \in \Gamma_{C^k}$, such that $t = s_j \xrightarrow{!m(d)[g]} s_{j+1}$ **do**
      - **for all** run $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \dots s_j \xrightarrow{!m(d)[g_j]} s_{j+1} \dots s_x \xrightarrow{\alpha[g_i(d)]} s_{x+1} \dots$ in C^k, $j < x$ **do**
        - $t \leftarrow s_j \xrightarrow{!m(d)[g_j \wedge \neg g_i(d)]} s_{j+1}$ `\* guard strengthening */`
      - **end for**
    - **end for**
  - **end for**
- **end for**
- `\* Event Enforcement and collection of runtime monitoring variables */`
- **for each** $s_x \in BSN$ **do**
  - **for each** $z \in BPN$, such that $z = s_x \xrightarrow{\alpha[g_i(d)]} s_{x+1} \in BPN$ **do**
    - **if** *exists* $\alpha' \in E^{\mathcal{D}}_{C^k}(s_x) \wedge \alpha' \in \Sigma_f$ **then**
      - save variable for runtime monitoring and enforcement
      - runtimeVariables $\leftarrow$ runtimeVariables $\cup\{d\}$ `\* variable saved to be monitored at runtime */`
    - **else**
      - Create new bad state s^{Bad} and divert all dynamic transitions that lead to s_x to s^{Bad}
      - Eliminate all transitions $(t, g) \in E^{\mathcal{D}}_{C^k}(s_x)$
      - $\mathcal{S}_{C^k} \leftarrow \mathcal{S}_{C^k} \backslash \{s_x\}$
    - **end if**
  - **end for**
- **end for**
- $C \leftarrow$ disableControllableDynamicTransition($\mathcal{G}_W, C$) `\* disable dynamic controllable transitions */`
- **return** C

end procedure

Algorithm 6 Disable Dynamic Controllable Transitions

Input: $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F)$, $C = (\mathcal{S}_C, \mathcal{S}_C^0, \mathcal{I}_C, \mathcal{O}_C, \mathcal{A}_C, \Gamma_C, \mathcal{S}_C^F)$
Output: C where dynamic controllable transitions that do not satisfy system requirements are disabled.

1: **procedure** DISABLECONTROLLABLEDYNAMICTRANSITION($\mathcal{G}_W$, C)
2: | **for each** state $p \in \mathcal{S}$ *such that* $\exists\, \delta \in \Sigma_c$ and $\exists g \in G$ *such that* $(\delta, g) \in E^D_{\mathcal{G}_W}(p)$ **do**
3: | | **for each** state $(p, q) \in \mathcal{S}_C$ *such that* $\nexists (\delta, g') \in E^D_{C \setminus \mathcal{S}_C^{Bad}}(p)$ and $g \wedge g'$ is satisfiable **do**
4: | | | $((\delta, (g \wedge \neg g')) \notin E^D_{C \setminus \mathcal{S}_C^{Bad}}(p, q))$ `\* disable` $(\delta, (g \wedge \neg g'))$`, since` $\delta \in \Sigma_c$ `*/`
5: | | **end for**
6: | **end for**
7: **end procedure**

Algorithm 6 is called by Algorithm 5 at Line 52 to disable dynamic controllable transitions that lead to a bad state.

Algorithm 7 Remove All Bad States and Associated Transition

Input: An SLTS C with states partitioned into good and bad states
Output: A SLTS C without bad states and transitions that lead to bad states

1: **procedure** REMOVEUNSAFESTATE(C)
2: | **while** $(\mathcal{S}_C^{Bad} \neq \emptyset)$ **do**
3: | | **if** $(\exists\, t = s \xrightarrow{\delta[g]} s^{\text{Bad}}$ such that $s^{\text{Bad}} \in \mathcal{S}_C^{Bad})$ **then**
4: | | | `\* eliminate all transitions associated with` s' `*/`
5: | | | $\Gamma_C \leftarrow \Gamma_C \setminus \{t\}$
6: | | | $\mathcal{S}_C \leftarrow \mathcal{S}_C \setminus \{s^{\text{Bad}}\}$
7: | | **end if**
8: | **end while**
9: | **return** C
10: **end procedure**

Once the iteration of the `repeat until` loop of Algorithm 1 has terminated, all states in $\mathcal{S}_C^{Bad}$ would been made unreachable and there is no need to keep them. Hence, Algorithm 7 is called to remove all bad states and transitions in the set of bad states $\mathcal{S}_C^{Bad}$. Algorithm 7 iterates over the set of states in $\mathcal{S}_C^{Bad}$ and eliminates all states in $\mathcal{S}_C^{Bad}$ including associated transitions.

Algorithm 8 Reverse The Direction Of Messages Of C

Input: An SLTS C

Output: An SLTS C with input messages changed to output messages and vice versa.

1: **procedure** REVERSEMESSAGEDIRECTION(C)
2: | **for each** transition $t = s \xrightarrow{!m[g]} s' \in \Gamma_C$ such that $!m \in \mathcal{I}_C$ **do**
3: | | $t \leftarrow s \xrightarrow{?m[g]} s' \in \Gamma_C$ * set output messages to input messages */
4: | | $\mathcal{I}_C \leftarrow \mathcal{I}_C \cup \{?m\}$
5: | | $\mathcal{O}_C \leftarrow \mathcal{O}_C \backslash \{!m\}$
6: | **end for**
7: | **for each** transition $t = s \xrightarrow{?m[g]} s' \in \Gamma_C$ such that $?m \in \mathcal{O}_C$ **do**
8: | | $t \leftarrow s \xrightarrow{!m[g]} s' \in \Gamma_C$ * set output messages to input messages */
9: | | $\mathcal{O}_C \leftarrow \mathcal{O}_C \cup \{!m\}$
10: | | $\mathcal{I}_C \leftarrow \mathcal{I}_C \backslash \{?m\}$
11: | **end for**
12: | **return** C
13: **end procedure**

Once all the issues relating to controllability and non-blocking have been dealt with, the next stage of the algorithm is to reverse the directions of the messages of the resulting controller. This is done to allow for communication between the plant and the controller. This ensures that an output message in the plant's transition system can be consumed by an input message in the controller's transition system and vice versa. Given an SLTS C as the input to Algorithm 8, it reverses the direction of the messages of C. That is, given an input (output) message $?m(x)$, Algorithm 8 will change it to and output (input) message *!m(x)* and vice versa.

A composition generation process may fail if the given plant cannot simulate its specification. Algorithm 9 iteratively pares down a given specification so that it can be simulated by a given plant. The algorithm takes $\mathcal{G}_W$ and $\mathcal{T}^W$ as inputs such that there is no simulation relation between $\mathcal{G}_W$ and $\mathcal{T}^W$ and returns a new specification $\mathcal{T}^{W'}$ that can be simulated by $\mathcal{G}_W$. Line 2 of Algorithm 9 defines a maximal relation R given by the cross product of the states of $\mathcal{G}_W$ and $\mathcal{T}^W$. Now Line 6 of the algorithm iterates over each pair of reachable states $(t_1, s_i) \in R$ such that there exists a transition $t_i \xrightarrow{\delta[g_1]} t_i' \in \Gamma_{\mathcal{T}^W}$ and then checks for the following three cases where a plant may fail to simulate a given specification.

- There is no matching transition at state s_i of the plant. In this case the transition $t_i \xrightarrow{\delta[g_1]} t_i' \in \Gamma_{\mathcal{T}^W}$ will be removed (Lines 8-9).
- There is exists a transition $s_i \xrightarrow{\delta[g_2]} t_i' \in \Gamma_{\mathcal{G}_W}$ and $(t_1, s_i) \in R$ but the guard g_1 is not a subguard of g_2, i.e., $g_1 \leq g_2$. In this case the guard on $\mathcal{T}^W$ is strengthened to that of $\mathcal{G}_W$ (Lines 11-12).
- There is exists a transition $s_i \xrightarrow{\delta[g_2]} t_i' \in \Gamma_{\mathcal{G}_W}$ and $g_1 \leq g_2$ but $(t_1, s_i) \notin R$. In this case the transition $t_i \xrightarrow{\delta[g_1]} t_i' \in \Gamma_{\mathcal{T}^W}$ is eliminated (Lines 13-15).

The algorithm terminates when all transitions that prevent the specification from being simulated by the plant have been dealt with, i.e., $\mathcal{T}^W == \mathcal{T}^{W'}$.

Algorithm 9 Refine Target To Simulate Plant

Input: $\mathcal{G}_W = (\mathcal{S}_{\mathcal{G}_W}, \mathcal{S}^0_{\mathcal{G}_W}, \mathcal{I}_{\mathcal{G}_W}, \mathcal{O}_{\mathcal{G}_W}, \mathcal{A}_{\mathcal{G}_W}, \Gamma_{\mathcal{G}_W}, \mathcal{S}^F_{\mathcal{G}_W})$ and $\mathcal{T}^W = (\mathcal{S}_{\mathcal{T}^W}, \mathcal{S}^0_{\mathcal{T}^W}, \mathcal{I}_{\mathcal{T}^W}, \mathcal{O}_{\mathcal{T}^W}, \mathcal{A}_{\mathcal{T}^W}, \Gamma_{\mathcal{T}^W}, \mathcal{S}^F_{\mathcal{T}^W})$
Output: $\mathcal{T}^{W'}$ where $\mathcal{T}^{W'}$ is derived from $\mathcal{T}^W$ and $\mathcal{T}^{W'} \preceq \mathcal{G}_W$

procedure REFINETARGETTOSIMULATEPLANT($\mathcal{T}^W, \mathcal{G}_W$)
  Let $R \leftarrow \mathcal{S}_{\mathcal{T}^W} \times \mathcal{S}_{\mathcal{G}_W}$
  Let $\mathcal{T}^{W'} \leftarrow \emptyset$
  repeat
    $\mathcal{T}^{W'} \leftarrow \mathcal{T}^W$
    for each $(t_i, s_i) \in R$ such that $\exists\, \delta \in \Sigma$, a guard g_1 such that there is a transition $t_i \xrightarrow{\delta[g_1]} t'_i \in \Gamma_{\mathcal{T}^W}$ **do**
      where t_i and s_i are reachable states
      if (there is no transition $s_i \xrightarrow{\delta[g_2]} s'_i \in \Gamma_{\mathcal{G}_W}$ such that $(g_1 \leq g_2)$ and $(t'_i, s'_i) \in R$) **then**
        $\Gamma_{\mathcal{T}^W} \backslash \{t_i \xrightarrow{\delta[g_1]} t'_i\}$ * `this removes the execution from the initial state to`
        t'_i `in` $\mathcal{T}^W$ */
      else if (there is a transition $s_i \xrightarrow{\delta[g_2]} s'_i \in \Gamma_{\mathcal{G}_W}$ and $(t'_i, s'_i) \in R$ but and $(g_1 \not\leq g_2)$) **then**
        $g_1 \leftarrow g_2$ * `change the guards to that of` $\mathcal{G}_W$ */
      else if (there is a transition $s_i \xrightarrow{\delta[g_2]} s'_i \in \Gamma_{\mathcal{G}_W}$ and $(g_1 \leq g_2)$ but $(t'_i, s'_i) \notin R$) **then**
        $\Gamma_{\mathcal{T}^W} \backslash \{t_i \xrightarrow{\delta[g_1]} t'_i\}$ * `this removes the execution from the initial state to`
        t'_i `in` $\mathcal{T}^W$ */
      end if
    end for
  until $(\mathcal{T}^{W'} == \mathcal{T}^W)$
  return $\mathcal{T}^{W'}$
end procedure

Algorithm 10 present a procedure to modify a given SLTS to include a special transition (ϵ transition) which we will call a "timeout" transition which will be used at runtime to provide the generated controller the ability to be able to preempt certain dynamic type 2 transitions (which we will call preemptable transitions) using enforceable transitions. Self-loops are treated in a similar way. The technique presented here is identical to that in the work by Wonham [54].

Algorithm 10 Plant and Specification Adjustment

Input: A SLTS $\mathcal{M} = (\mathcal{S}_\mathcal{M}, \mathcal{S}^0_\mathcal{M}, \mathcal{I}_\mathcal{M}, \mathcal{O}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \Gamma_\mathcal{M}, \mathcal{S}^F_\mathcal{M})$
Output: A new SLTS constructed by modifying $\mathcal{M}$ to include a "timeout" transition (ϵ transition) to enable event enforcement at runtime.

1: **procedure** PLANTANDSPECADJUSTMENT($\mathcal{M}$)
2: Initialize the set dynamic_2_Transition to be all dynamic type 2 transitions such that
3: dynamic_2_Transition $\subseteq \Gamma_\mathcal{M}$
4: `\* Initialize the set of dynamic 2 transitions which forms the set of`
5: `preemptable transitions*/`
6: **for** each state s $\in \mathcal{S}_\mathcal{M}$ **do**
7: **if** ($\exists\ \delta \in \Sigma_f$ and exists a guard g such that $s \xrightarrow{\delta[g]} \in \Gamma_\mathcal{M}$) $\wedge$ ($\exists\ \lambda$ and exists a guard g'
8: such that $s \xrightarrow{\lambda[g']} \in$ *dynamic_2_Transition*) **then**
9: split s into 2 states s'*and*s'' and create a new transition such that $s' \xrightarrow{\epsilon} s''$ where ϵ serves as a
10: delay to help preempt preemptable events.
11: **for each** transition t enabled at s **do**
12: **if** $t \in$ dynamic_2_Transition **then**
13: define t with its source state at s''
14: **else** define t with its source state at s'
15: **end if**
16: **end for**
17: $\mathcal{S}_\mathcal{M} \backslash \{s\} \cup \{s'\} \cup \{s''\}$
18: $\Gamma_\mathcal{M} \cup \{s' \xrightarrow{\epsilon} s''\}$
19: **end if**
20: **end for**
21: **return** $\mathcal{M}$
22: **end procedure**

Example 5.1 below is a small example to illustrate how Algorithm 1 works.

Example 5.1.

Consider the plant $\mathcal{G}_W$ (assumed to be the asynchronous parallel composition of some given services) in Figure 10(a) and the specification $\mathcal{T}^W$ in Figure 10(b) as the input to Algorithm 1. As noted above input and function invocation transitions are considered as uncontrollable and output transitions are controllable (i.e., $!msg_1(x), !msg_3(var), !msg_2(z) \in \Sigma_c$, $?msg_2(z)$, $?msg_1(x)$, $?msg_2(z)$, $?msg_3(var)$, $atom_3(x)$, $atom_1(x, y)$, $atom_2(y, v)$, $atom_3(x)$, $?fail() \in \Sigma_{uc}$, $!fail() \in \Sigma_f$). Clearly, $\mathcal{G}_W$ simulates $\mathcal{T}^W$ and both SLTSs are communication-error free. Line 5 of Algorithm 1 computes the composition refinement C^0 which is represented in Figure 10(c). Applying the rest of Algorithm 1 (Lines 7-22) to C^0 gives the following (where we will only consider steps of the algorithm that are relevant to this example):

1. **(Algorithm 1 Lines 7-12)** within the `repeat until` loop we have the following steps:
   (a) **k=1** means we will pass C^0 to the three functions within the `repeat until` loop (Algorithm 1 Lines 7-12)
      i. **Static Controllability** (Line 9 Algorithm 1)
      Starting from the initial state of C^0 every state of C^0 satisfies the static controllability condition except for state (s_5, t_5) where there is an uncontrollable static transition $(s_5) \xrightarrow{?msg_2(z)} (s_6)$

enabled in the plant but not at the corresponding state (s_5, t_5) of C^0. This implies that state (s_5, t_5) and the associated transitions $((s_5, t_5) \xrightarrow{atom_3(x)} (s_2, t_2)$ and $(s_3, t_3) \xrightarrow{?msg_1(x)} (s_5, t_5))$ will be eliminated from C^0 (Lines 11 -14 of Algorithm 3). This produces the SLTS in Figure 10(d)

ii. **Blocking states** (Line 10 Algorithm 1)
At this stage no state of the C^0 in Figure 10(d) is blocking, we proceed to the next step.

iii. **Dynamic Controllability** (Line 11 Algorithm 1)
At this stage of the algorithm, there are three transitions that lead to bad states $((s_2, t_2) \xrightarrow{atom_1(x,y)[3\leq x<7]} (BAD_1), (s_4, t_4) \xrightarrow{atom_2(y,v)[y=a]} (BAD_2), (s_1, t_2) \xrightarrow{msg_3(var)[var \in \{dr\}]} (BAD_3))$ in the C^0 of Figure 10(d). By applying the next step of the algorithm we have the following.

- **Strengthening and Attachment of Guards** (Lines 30-38 of Algorithm 5)
The two uncontrollable transitions $(s_2, t_2) \xrightarrow{atom_1(x,y)[3\leq x<7]} (BAD_1)$ and $(s_1, t_1) \xrightarrow{msg_3(var)[var \in \{dr\}]} (BAD_3)$ are dynamic type 1 transitions since the values of the variables x and var can be tracked from where they were declared. Hence, at this stage the algorithm strengthens the guards of these transitions so that the state BAD_1 and BAD_3 are made unreachable by attaching the guard $\neg(3 \leq x < 7)$ and $var \notin \{dr\}$ to the transition $(s_0, t_0) \xrightarrow{!msg_1(x)} (s_1, t_1)$ and the transition $(s_0, t_0) \xrightarrow{!msg_3(var)} (s_1, t_1)$, respectively. The resultant SLTS is given in Figure 10(e).
- **Event Enforcement and Collection of Runtime Variable** (Lines 40-50 of Algorithm 5)
The transition $(s_4, t_4) \xrightarrow{atom_2(y,v)[y=a]} (BAD_2)$ is a dynamic type 2 transition since the variable y depends on the output of the atomic operation $atom_1(x, y)$ which cannot be tracked. This step of the algorithm will check for the enforceable transition $(s_4, t_4) \xrightarrow{!fail()} (s_0, t_0)$ and save the variable y to be monitored at runtime and then enforce $!fail()$ when y=a, to prevent BAD_2 from being reached.

(b) **k=2** means we will pass C^1 (Figure 10(e)) to the three functions in the `repeat until` loop (Algorithm 1 Lines 7-12)

i. **Static Controllability**(Line 9 Algorithm 1)
Again starting from the initial state of C^1 (Figure 10(e)) every state of C^1 satisfies the static controllability condition except for state (s_3, t_3) where there is an uncontrollable static transitions $(s_3) \xrightarrow{?msg_1(x)} (s_5)$ enabled in the corresponding state of the plant but not at the state (s_3, t_3) of C^1. This implies that state (s_3, t_3) and the associated transitions $((s_3, t_3) \xrightarrow{?msg_2(z)} (s_1, t_1)$ and $(s_7, t_7) \xrightarrow{!msg_2(z)} (s_3, t_3))$ will be eliminated from C^1 (Lines 11 -14 of Algorithm 3).

ii. **Blocking states** (Line 10 Algorithm 1).
Now, because in the previous step the transition from state (s_7, t_7) to state (s_3, t_3) was eliminated, state (s_7, t_7) becomes a blocking state since it is not a final state and does not lead to a final state. Given this we have the following steps:
- create a new state BAD_4 and assign any dynamic transition leading to state (s_7, t_7) to BAD_4 (Algorithm 4 Lines 13-15)
- then eliminate (s_7, t_7) (Algorithm 4 Lines 18-24)

The results of (i) and (ii) are shown in the diagram in Figure 10(f).

iii. **Dynamic Controllability** (Line 11 Algorithm 1)
A new bad state BAD_4 was added to C^1 and needs to be made unreachable. (Note that all other BAD states are still unreachable.) We have the following step:
- **Strengthening and Attachment of Guards** (Lines 30-38 of Algorithm 5)
$(s_1, t_1) \xrightarrow{msg_3(var)[var \in \{cr\}]} (BAD_4)$ is an uncontrollable dynamic type 1 transitions since the

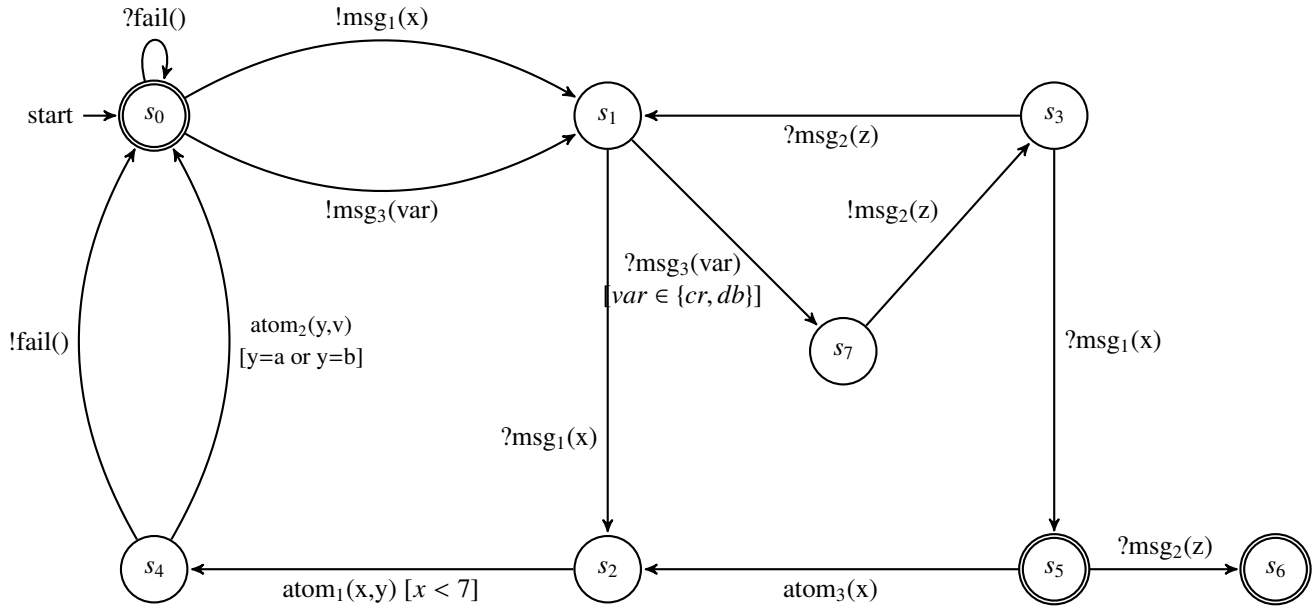
values of the variable *var* can be tracked from when it was declared. Hence, at this stage the algorithm strengthens the guard of this transition so that the state BAD_4 is made unreachable by attaching the guard $var \notin \{cr\}$ to the transition $(s_0, t_0) \xrightarrow{!msg_3(var)[var \notin \{dr\}]} (s_1, t_1)$ (shown Figure 10(f)).

(c) **k=3** means we will pass C^2 (Figure 10(f)) to the three functions in the `repeat until` loop. Iterating over the `repeat until` loop again will return the same SLTS shown in Figure 10(f), i.e., $C^3 = C^2$, hence the loop terminates.

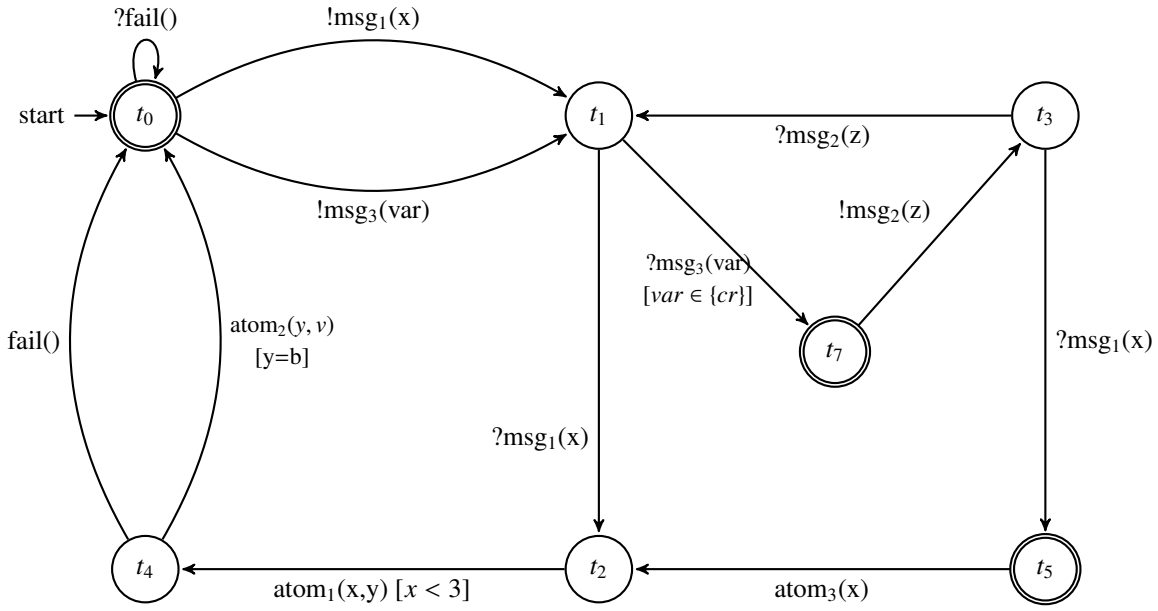
2. **Line 13 of Algorithm 1**
Now once the loop terminates, the next stage is to remove all the bad states ($BAD_1, BAD_2, BAD_3, BAD_4$) from C^3 by calling Algorithm 7. In addition, all transitions in and out of these bad states are removed.
3. **Line 14 of Algorithm 1**
C^3 is not communication-error free since the transition $(s_0, t_0) \xrightarrow{!msg_3(var)[var \notin \{dr,cr\}]} (s_1, t_1)$ can emit the message $!msg_3(var)$ but there is no matching input transition in C^3 to consume it, hence this step eliminates $(s_0, t_0) \xrightarrow{!msg_3(var)[var \notin \{dr,cr\}]} (s_1, t_1)$ from C^3 and the plant is updated accordingly.
4. **Line 15 of Algorithm 1,**
This step reverses the messages of C^2 producing the final output C as shown in Figure 10(g).

5.1. Discussion

The task of labeling an existing transition as enforceable transition or creating a new enforceable transition in the controlled system to be used to preempt dynamic type 2 transitions is mostly dependent on the domain and the designer perspective. In addition, in choosing the target state of an enforceable transition is dependent of the current state of the system and the domain being modeled. That is, in case there is a failure as a result unsuspected output of a dynamic type 2 transition, what state should the system go to; should the system transition to the initial state, should the system try to do the previous transition again and so on. In Example 5.1 the choosing of the enforceable transition was trivial since the plant already had a transition $(s_4, t_4) \xrightarrow{!fail()} (s_0, t_0)$ that lead to the initial state and because this example had no domain restriction.

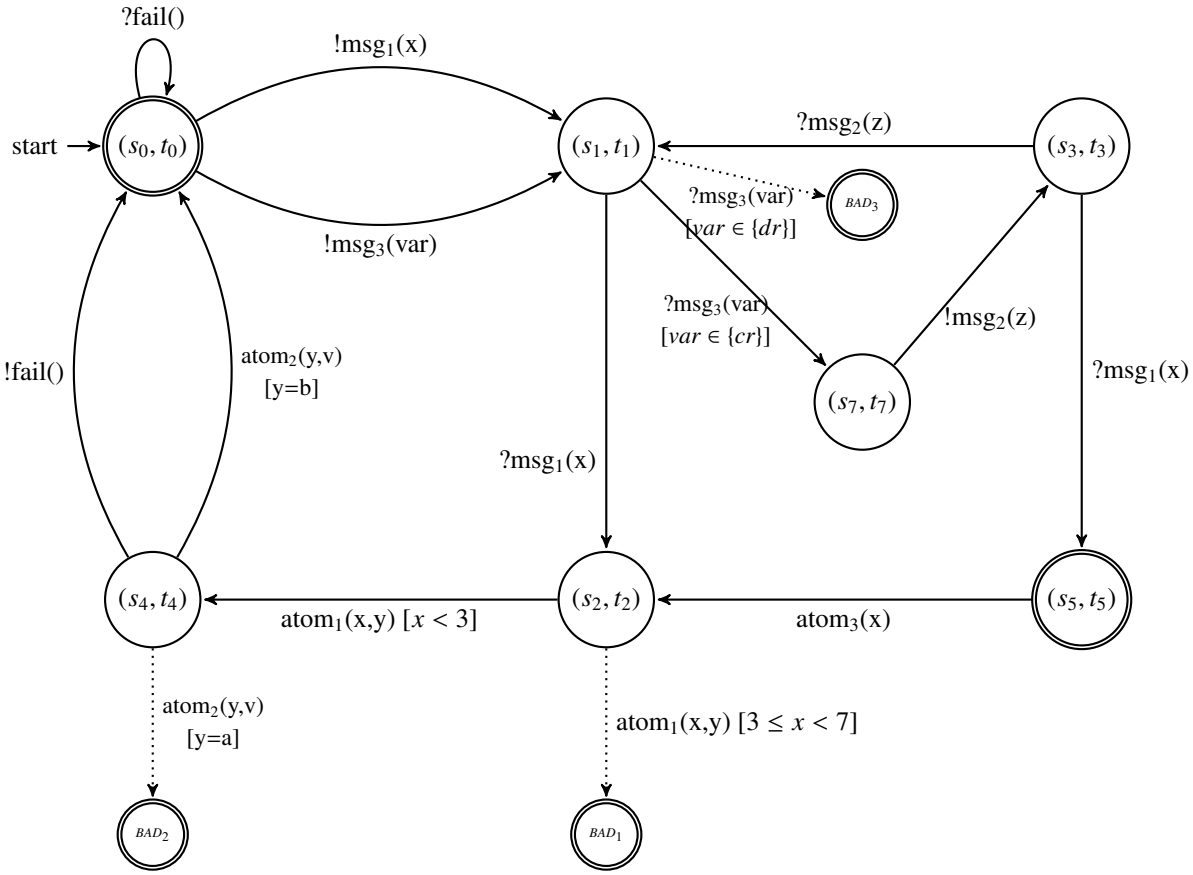


(a) Plant $\mathcal{G}_W$

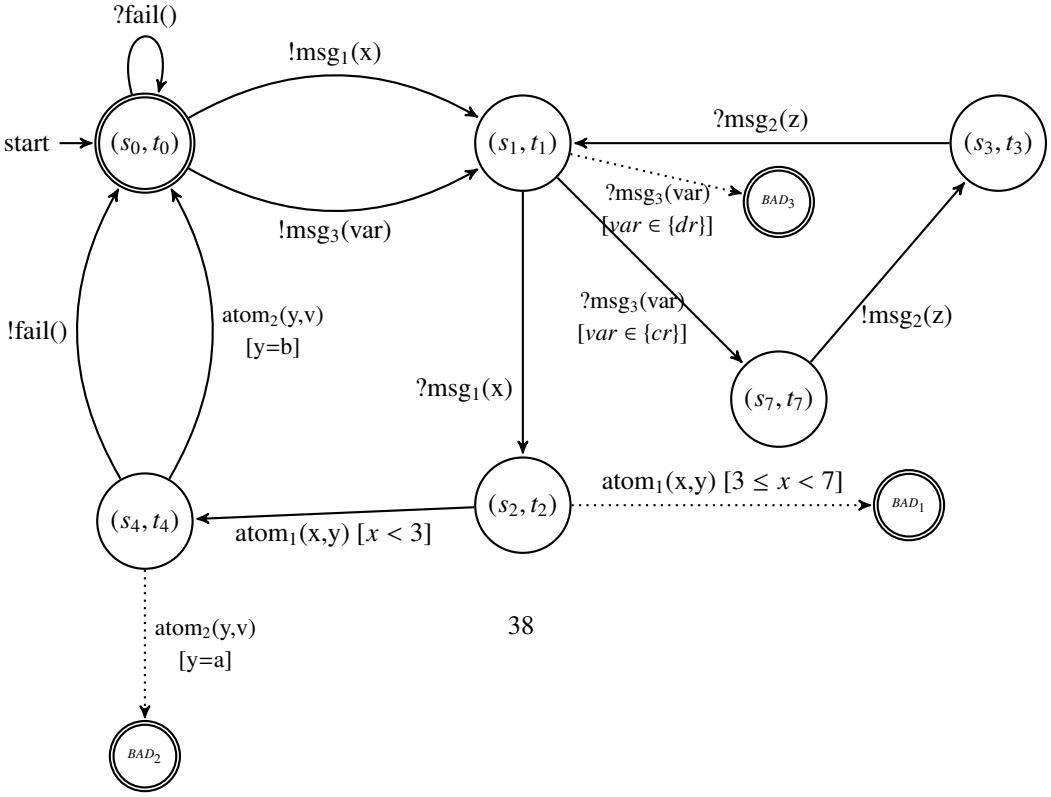


(b) Target services $\mathcal{T}^W$

Figure 10: Illustrative Example using the Algorithm

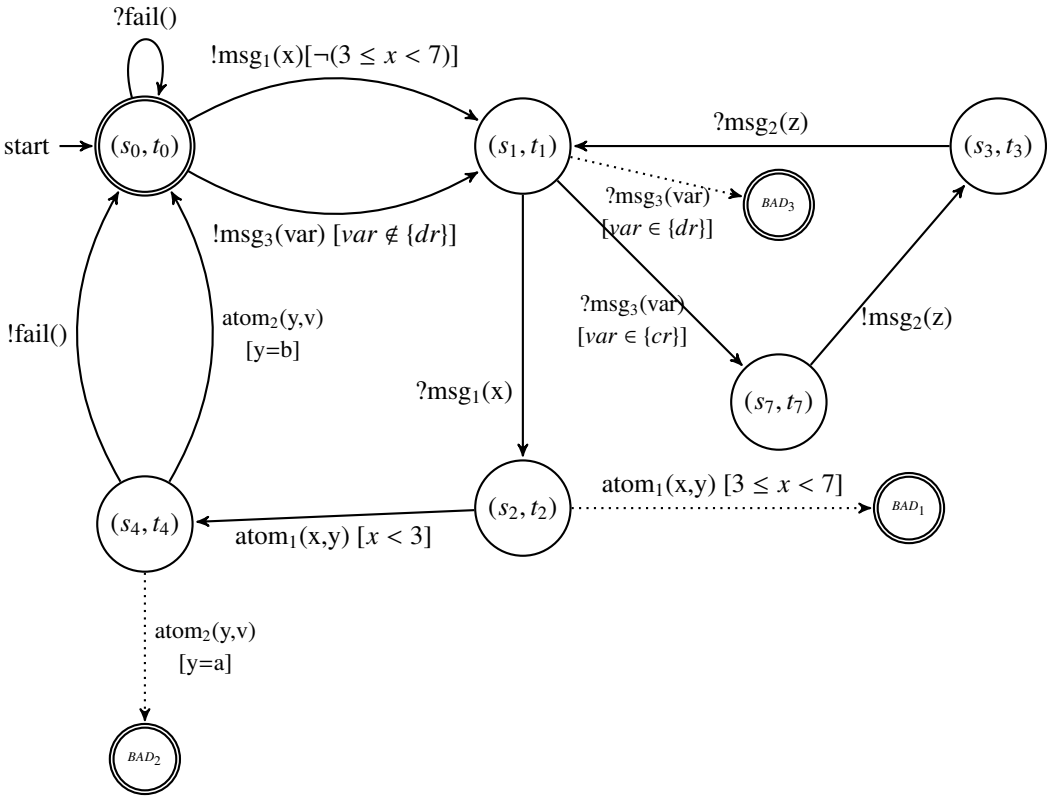


(c) Composition Refinement $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$

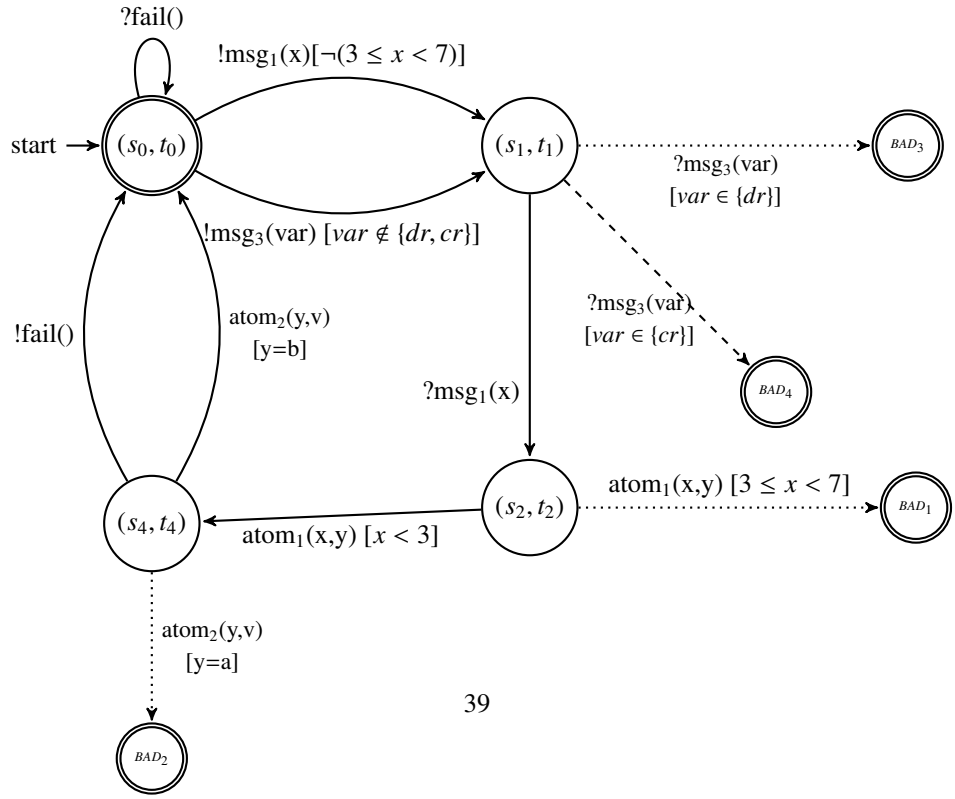


(d) C^0 after applying static controllability

Figure 10: Illustrative Example using the Algorithm

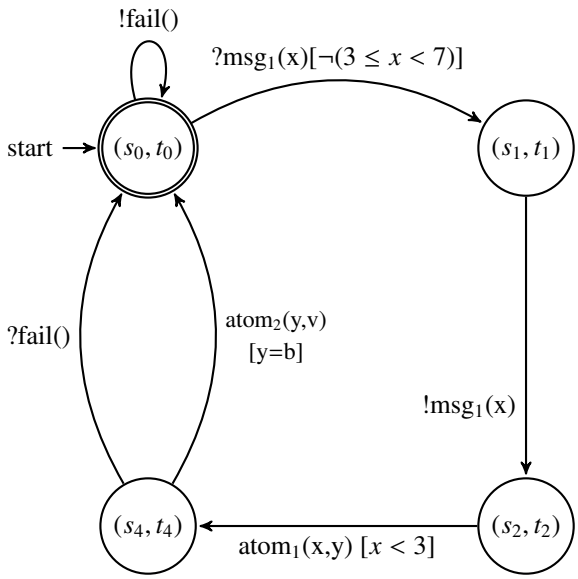


(e) C^1 (the variable $y = a$ is monitored at runtime)



(f) C^2 (the variable $y = a$ is monitored at runtime)

Figure 10: Illustrative Example using the Algorithm



(g) Final output of Algorithm 1 C (the variable $y = a$ is monitored at runtime)

Figure 10: Illustrative Example using the Algorithm

We have applied Algorithm 1 to various small examples. Also, Algorithm 1 has been manually applied to the flight booking example introduced earlier. The asynchronous parallel product forming the plant of this example has 150 states and 3410 transitions while computing the composition refinement of the plant and the specification yielded a transition system C^0 with 175 states and 3945 transitions. Applying Lines 6-16 of the algorithm to C^0 will further reduce the number of states and transitions. We hope to automate the process in the near future.

6. Proof of Correctness (Soundness and Completeness)

In this section, we present a theorem and a proof that proves the correctness of our approach. Theorem 6.4 shows that there exists a controller which solves the composition problem stated in Definition 4.11. We will start with various definitions and then we will state the main theorem of the section and finally provide a constructive proof for the theorem.

6.1. Proof of Controller Existences

Before we prove Theorem 3, let us consider the following lemmata resulting from observations made from the construction of the controller C by Algorithm 1. Let $\mathcal{S}_C$ and $\mathcal{S}_{C^0}$ denote the set of states of C and C^0, respectively. Also, let $\mathcal{S}_{C^0} \backslash \mathcal{S}_{C^0}^{Bad} = \{s \mid s \in \mathcal{S}_{C^0} \wedge s \notin \mathcal{S}_{C^0}^{Bad}\}$. In the proofs that follow, we will refer to bad states $\mathcal{S}_C^{Bad}$ as states that are not reachable in the specification or that violate controllability or nonblocking conditions. Also, during the construction of C by Algorithm 1, new states are created. These states are also marked as bad states, since they do not satisfy either some controllability or nonblocking conditions.

The first lemma says that in constructing C, our algorithm only removes states from C^0 that are bad. That is, a state in the resulting C is a state in the original C^0 and is not a bad state.

Lemma 6.1. *Given a controller C generated by Algorithm 1 such that $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$, then $\mathcal{S}_C \subseteq \mathcal{S}_{C^0} \backslash \mathcal{S}_{C^0}^{Bad}$.*

Proof. The proof is done constructively,
In Algorithm 1, Line 5, C is initially given by $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$ and $\mathcal{S}_{C^0} = \mathcal{S}_{C^0}^{Good} \cup \mathcal{S}_{C^0}^{Bad}$. This implies that at this stage in the construction of C by the algorithm, the set of states of C is equal to the set of states of C^0. That is, $\mathcal{S}_C = \mathcal{S}_{C^0}$ which means $\mathcal{S}_C = \mathcal{S}_{C^0}^{Good} \cup \mathcal{S}_{C^0}^{Bad}$. Let C^k denote the resultant SLTS obtained after some k iterations of the `repeat until` loop of Algorithm 1 on C^0. Now to show that $\mathcal{S}_C \subseteq \mathcal{S}_{C^0} \backslash \mathcal{S}_{C^0}^{Bad}$, we prove the following:

(i) Upon termination of Algorithm 1, all the states in $\mathcal{S}_{C^0}^{Bad}$ have been made unreachable and eliminated and hence would not be in the final output C of Algorithm 1,

(ii) In the iterations of Algorithm 1, some of the states in $\mathcal{S}_{C^0}^{Good}$ become bad states and are made unreachable and eliminated,

(iii) All new bad states created by the algorithm are made unreachable and eliminated before the algorithm terminates .

To show (i), let $q \in \mathcal{S}_C^{Bad}$ and suppose that $\exists\, p \in \mathcal{S}_{C^0}, \exists\, \delta \in \Sigma$ and $\exists\, g \in G$ such that the transition $t = p \xrightarrow{\delta[g]} q$ in C^0, then in the first iteration of the `repeat until` of Algorithm 1 loop one of the following holds:

(a) if $\delta \in \Sigma_{uc}$ then

- in the case that t is static Lines 11-13 of Algorithm 3 will eliminate t which implies that q is also eliminated.
- in the case that t is a dynamic type 1 transition, then the guard g will be strengthened by Algorithm 5 in Lines 30-38. Hence, q becomes unreachable from any good state and finally deleted at Line 13 of Algorithm 1.
- in the case that t is a dynamic type 2 transition then by Corollary 4.1, Algorithm 5 Lines 40-51 will ensure that there is an enforceable transition also enabled at state p to preempt t at runtime. Hence, state q becomes unreachable and is later deleted at Line 13 of Algorithm 1.

(b) if $\delta \in \Sigma_c$ then
in all cases t (static or dynamic transition) would be disabled by Algorithm 1, making q unreachable and later deleted at Line 13 of Algorithm 1.

To show (ii) we note that after some k iteration of the `repeat until` loop of Algorithm 1 on C^0 some good state in $\mathcal{S}_{C^0}^{Good}$ becomes bad due to controllability (Definition 4.8) or blocking (Definition 4.9). These new bad states are treated in the same way as done in (i), which implies that they are never reachable in the final output of Algorithm 1.

The proof of (iii) is as follows: During the construction of C, Algorithm 1 creates completely new bad states in the process of constructing C (Lines 7-9 of Algorithm 3, Lines 12-15 of Algorithm 4 and Line 46 of Algorithm 5), however, these new bad states are also treated and deleted in the same way as in (i) before the termination of Algorithm 1.

From (i), (ii) and (iii) it is clear that by the time Algorithm 1 terminates the set of bad states of its final output C will be empty, i.e., $\mathcal{S}_C^{Bad} = \emptyset$, and some of the states in $\mathcal{S}_{C^0}^{Good}$ would have been converted into bad and removed too. This means that the set of states of C is only a subset of $\mathcal{S}_{C^0}^{Good}$. Thus, $\mathcal{S}_C \subseteq \mathcal{S}_{C^0} \backslash \mathcal{S}_{C^0}^{Bad}$. ■

In the following lemma we show that a state $s \in \mathcal{S}_{C^0} \backslash \mathcal{S}_C$ is either a bad state in C^0 or was made bad at the k^{th} iteration of the `repeat until` loop of Algorithm 1 over C^0.

Lemma 6.2. *Given that $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$ and $s \in \mathcal{S}_{C^0} \backslash \mathcal{S}_C$, then one of the following holds*

(i) $s \in \mathcal{S}^{Bad}_{C^0}$,

(ii) $s \in \mathcal{S}^{Bad}_{C^k}$ *where C^k is the SLTS obtained after some k iterations of the* `repeat until` *loop of Algorithm 1*

Proof. Proof of (i): given that $s \in \mathcal{S}_{C^0} \backslash \mathcal{S}_C$ implies that $s \notin \mathcal{S}_C$ and from Lemma 6.1 it implies that $s \in \mathcal{S}^{Bad}_{C^0}$ which proves (i).

Proof of (ii): This follows from Lemma 6.1 item (ii) by noting that, during the iterations of the `repeat until` loop of Algorithm 1, a good state s in C^{k-1} is changed to a bad state $s \in \mathcal{S}^{Bad}_{C^k}$ because either s is not state controllable or leads to a violation of controllability (Definition 4.8) or results in blocking (Definition 4.9). ∎

In the following lemma we show that the set of good states of C is a subset of the set of good states of C^0.

Lemma 6.3. *Given a controller C generated by Algorithm 1 such that $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$, then $\mathcal{S}^{Good}_{C} \subseteq \mathcal{S}^{Good}_{C^0}$.*

Proof. The proof of this lemma is similar to that of Lemma 6.1 and so we omit it here. ∎

Theorem 6.4 (Controller Existence). *Given a system modeled by an SLTS $\mathcal{G}_W$ and a specification $\mathcal{T}^W$ with $\mathcal{T}^W \preceq \mathcal{G}_W$, a controller C exists such that $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$ if and only if $\mathcal{G}_W$ is state controllable with respect to $\mathcal{T}^W$.*

Proof. The proof is done constructively,

Let $\mathcal{G}_W = (\mathcal{S}_{\mathcal{G}_W}, \mathcal{S}^0_{\mathcal{G}_W}, \mathcal{I}_{\mathcal{G}_W}, \mathcal{O}_{\mathcal{G}_W}, \mathcal{A}_{\mathcal{G}_W}, \Gamma_{\mathcal{G}_W}, \mathcal{S}^F_{\mathcal{G}_W})$ and $\mathcal{T}^W = (\mathcal{S}_{\mathcal{T}^W}, \mathcal{S}^0_{\mathcal{T}^W}, \mathcal{I}_{\mathcal{T}^W}, \mathcal{O}_{\mathcal{T}^W}, \mathcal{A}_{\mathcal{T}^W}, \Gamma_{\mathcal{T}^W}, \mathcal{S}^F_{\mathcal{T}^W})$
Let $Composer(\mathcal{G}_W, \mathcal{T}^W)$ denote the transition system obtained from $\mathcal{G}_W$ and $\mathcal{T}^W$ by applying Algorithm 1 (i.e., the final output of Algorithm 1). Let $C = Composer(\mathcal{G}_W, \mathcal{T}^W)$.

We will denote the set of states of C^0 as $\mathcal{S}_{C^0}$ and the set of transitions of C^0 as Γ_{C^0}. Let $\mathcal{S}^{Good}_{C^0}$ and $\mathcal{S}^{Bad}_{C^0}$ denote the set of good states of C^0 and the set of bad states of C^0, respectively. Let $C^0 \backslash \mathcal{S}^{Bad}_{C^0}$ denote the SLTS obtained after removing the set of bad states $\mathcal{S}^{Bad}_{C^0}$ from C^0, i.e., C^0 excluding the set of bad states. We will assume similar notation for C. We make the following observations. In Algorithm 1, C is initially computed from $\mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$ (Line 5) and then certain reduction steps are further performed on it. By definition, $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$ in Line 5 of Algorithm 1 (definition of composition refinement). The states of C^0 are partitioned into the set of *good* states and the set of *bad* states, respectively. That is, $\mathcal{S}_{C^0} = \mathcal{S}^{Good}_{C^0} \cup \mathcal{S}^{Bad}_{C^0}$. The set of good states and transitions leading to good states of C^0 lie in $\mathcal{T}^W$. Specifically, C is obtained from C^0 after removing certain transitions and bad states.

Let $C \otimes \mathcal{G}_W$=$(\mathcal{S}_{C\otimes\mathcal{G}_W}, \mathcal{S}^0_{C\otimes\mathcal{G}_W}, \mathcal{I}_{C\otimes\mathcal{G}_W}, \mathcal{O}_{C\otimes\mathcal{G}_W}, \mathcal{A}_{C\otimes\mathcal{G}_W}, \Gamma_{C\otimes\mathcal{G}_W}, \mathcal{S}^F_{C\otimes\mathcal{G}_W})$ denote the controlled system when $\mathcal{G}_W$ is under control of C.

Part 1:
if: given that $C = Composer(\mathcal{G}_W, \mathcal{T}^W)$ and $\mathcal{T}^W$ is controllable.
To prove:

1. $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$

That is, show that when the plant is coupled with C, the resultant transition system is simulated by $\mathcal{T}^W$.

Example 6.1. *First let us illustrate this part of the proof of Theorem 6.4 by an example (i.e., given that $C = Composer(\mathcal{G}_W, \mathcal{T}^W)$ and $\mathcal{T}^W$ is controllable, show that $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$). Let the SLTS in Figure 10(a) and Figure 10(b) represent the plant and the specification, respectively. It can be verified that $\mathcal{T}^W \preceq \mathcal{G}_W$. Figure 10(c) is the result of computing the composition refinement ($C^0 = \mathcal{G}_W \times_{ref} \mathcal{T}^W$) of the SLTSs in Figure 10(a) and Figure 10(b), respectively. Now it can be seen that C^0 is made up good states and bad states. Our proof to the theorem will be in two parts. The first part of the proof would be to show that ($C^0 \otimes \mathcal{G}_W$) simulates $\mathcal{T}^W$ when the transitions ($(s_2, t_2) \xrightarrow{atom_1(x,y)[3\le x<7]} (BAD_1)$, $(s_4, t_4) \xrightarrow{atom_2(x,y)[y=a]} (BAD_2)$, $(s_1, t_2) \xrightarrow{mesg_3(var)[var \in \{dr\}]} (BAD_3)$) have been eliminated from C^0 and the bad states made unreachable. It can be seen that the controller C in Figure 10(g) has no transitions that lead to bad states and it can also be easily verified that $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$ which will constitute the proof of the second part. That is, the second part of the proof would be to show that C generated from C^0 by Algorithm 1 has no bad states and has no transitions that lead to bad states and, hence, satisfies $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$.*

The proof of Part 1 will proceed as follows. Firstly, in (a) we would define a relation R and show that R is a simulation relation between ($C^0 \otimes \mathcal{G}_W$) and $\mathcal{T}^W$ when C^0 is restricted to having only good states. i.e., $(C^0 \backslash \mathcal{S}^{Bad}_{C^0} \otimes \mathcal{G}_W) \preceq \mathcal{T}^W$. Secondly, in (b) we shall further define another relation $R \upharpoonright \mathcal{S}_C$, a subset of R and establish that $R \upharpoonright \mathcal{S}_C$ is a simulation relation between ($C \otimes \mathcal{G}_W$) and $\mathcal{T}^W$ if C is generated without bad states and transitions that lead to bad states. Finally, in (c) we shall show that actually C has no bad states and has no transitions that lead to bad states.

We define the relation

$$R = \{ (((s_n, t_n), s_n), t_n) \mid \exists\, \delta_0, \delta_1, \ldots, \delta_n \in \Sigma, \exists\, s_0, s_1, \ldots, s_n \in \mathcal{S}_{\mathcal{G}_W}, \exists\, t_0, t_1, \ldots, t_n \in \mathcal{S}_{\mathcal{T}^W}, \exists\, g_0, g_1, \ldots, g_n \in G : (s_0 \in \mathcal{S}^0_{\mathcal{G}_W}, t_0 \in \mathcal{S}^0_{\mathcal{T}^W}, [\forall\, (0 < i \le n)\, [s_{i-1} \xrightarrow{\delta_{i-1}[g_{i-1}]} s_i \in \Gamma_{\mathcal{G}_W}] \wedge [t_{i-1} \xrightarrow{\delta_{i-1}[g_{i-1}]} t_i \in \Gamma_{\mathcal{T}^W}]]) \} \quad (1)$$

(a) We will start by showing that R is a simulation relation between $C^0 \otimes \mathcal{G}_W$ and $\mathcal{T}^W$ when the states of $C^0 \otimes \mathcal{G}_W$ are constrained to only good states, i.e., $\mathcal{T}^W$ simulates $((C^0 \backslash \mathcal{S}^{Bad}_{C^0}) \otimes \mathcal{G}_W)$.
Consider $(((s_n, t_n), s_n), t_n) \in R$ and $(s_n, t_n) \in \mathcal{S}^{Good}_{C^0}$,
and suppose $((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C^0 \otimes \mathcal{G}_W}$ for some $\delta_n \in \Sigma$ and $(s_{n+1}, t_{n+1}) \in \mathcal{S}^{Good}_{C^0}$, then by definition of $\times_{\text{ref}}$ used in the construction of C^0 it implies that $\exists\, t_{n+1} \in S_{\mathcal{T}^W}$ *such that* $t_n \xrightarrow{\delta_n[g_n]} t_{n+1} \in \Gamma_{\mathcal{T}^W}$ and therefore $(((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R$.
This is because from the definition of $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$, every transition that leads to a good state in C^0 from (s_n, t_n) can be matched by $\mathcal{T}^W$. This implies that R is a simulation relation between $((C^0 \backslash \mathcal{S}^{Bad}_{C^0}) \otimes \mathcal{G}_W)$ and $\mathcal{T}^W$.
As in Example 6.1 if we eliminate all bad states and their associated transitions $((s_2, t_2) \xrightarrow{atom_1(x,y)[3\le x<7]}$

(BAD_1), $(s_4, t_4) \xrightarrow{atom_2(x,y)[y=a]} (BAD_2)$, $(s_1, t_2) \xrightarrow{mesg_3(var)[var \in \{dr\}]} (BAD_3)$) *from* C^0 *of Figure 10(c) and combine it with the plant in Figure 10(a) we get a system that is simulated by the specification in Figure 10(b).*
Now in (b) and (c) we prove the following claim:

$$C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$$

(b) (Transitions of C that lead to good states from good states)
Based on the results obtained in (a), we want to show that $\mathcal{T}^W$ also simulates $C \otimes \mathcal{G}_W$ based on the fact that C is constructed from C^0 and that C has no bad states and no transitions that lead to bad states.

From Algorithm 1, C is built from C^0 by making bad states unreachable and removing all transitions that lead to bad states.

(i) Based on Lemma 6.1 and Lemma 6.2, we define another relation $R \upharpoonright \mathcal{S}_C$ a projection of the states of $\mathcal{S}_C$ into R given by:

$$R \upharpoonright \mathcal{S}_C = R \backslash \{(((s_n, t_n), s_n), t_n) \mid ((s_n, t_n), s_n) \notin \mathcal{S}_{C \otimes \mathcal{G}_W}\}$$

i.e., the set of pairs in R excluding those not in $\mathcal{S}_{C \otimes \mathcal{G}_W}$. We also note that $(R \upharpoonright \mathcal{S}_C) \subseteq R$.

(ii) Now we are ready to show that $R \upharpoonright \mathcal{S}_C$ is a simulation relation between $C \otimes \mathcal{G}_W$ and $\mathcal{T}^W$ if no transition in C leads to a bad state.
Consider $(((s_n, t_n), s_n), t_n) \in R \upharpoonright \mathcal{S}_C$ for some states $((s_n, t_n), s_n) \in S_{C \otimes \mathcal{G}_W}$ and $t_n \in S_{\mathcal{T}^W}$, and $(s_n, t_n) \in \mathcal{S}_C^{Good}$, then from (i) we have that $(((s_n, t_n), s_n), t_n) \in R$
Now suppose

$$\begin{aligned}
&((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes \mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Good} \\
&\implies ((s_n, t_n), s_n) \xrightarrow{\delta_n[g'_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C^0 \otimes \mathcal{G}_W} \textit{ with } (g_n \preceq g'_n), \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_{C^0}^{Good} \\
&\qquad\qquad \text{(by the construction of } C \text{ from } C^0) \\
&\implies \exists\, t_{n+1} \in S_{\mathcal{T}^W} \textit{ such that } t_n \xrightarrow{\delta_n[g'_n]} t_{n+1} \in \Gamma_{\mathcal{T}^W} \textit{ with } (g_n \preceq g'_n), \\
&\textit{and } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \\
&\qquad\qquad \text{(by the fact that R is a simulation relation between } C^0 \otimes \mathcal{G}_W \text{ and } \mathcal{T}^W) \\
&\implies \exists\, t_{n+1} \in S_{\mathcal{T}^W} \textit{ such that } t_n \xrightarrow{\delta_n[g'_n]} t_{n+1} \in \Gamma_{\mathcal{T}^W} \textit{ with } (g_n \preceq g'_n), \\
&\textit{and } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \upharpoonright \mathcal{S}_C \qquad \text{(by definition of } R \upharpoonright \mathcal{S}_C) \\
&\implies \exists\, t_{n+1} \in S_{\mathcal{T}^W} \textit{ such that } t_n \xrightarrow{\delta_n[g_n]} t_{n+1} \in \Gamma_{\mathcal{T}^W} \textit{and } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \upharpoonright \mathcal{S}_C \\
&\qquad\qquad \text{(by definition of simulation relation on } g_n \text{ and } g'_n)
\end{aligned}$$

That is, $R \upharpoonright \mathcal{S}_C$ is a simulation relation between $(C \backslash \mathcal{S}_C^{Bad}) \otimes \mathcal{G}_W$ and $\mathcal{T}^W$. So now we have that every transition that leads to a good state from a good state in $C \otimes \mathcal{G}_W$ can be simulated by $\mathcal{T}^W$. Hence, we only have left to show that C has no bad states and has no transitions that lead to bad states from a good state.

(c) (Transitions of C that lead to bad states from a good state)
Here, we show that all bad states and transitions that lead to bad states are eliminated from C during the construction of C from C^0. That is, C has no bad states and has no transitions that lead to bad

states. For example, the synthesis of the controller C in Figure 10(g) from C^0 in Figure 10(c) resulted in the elimination of all transitions that lead to bad states and transitions that violate controllability (Definition 4.8).
Now consider

$$T = ((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes \mathcal{G}_W} \tag{2}$$

and $(s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}$ such that T is a transition in the sequence (execution)

$$s_0 \xrightarrow{\delta_0[g_0]} s_1 \xrightarrow{\delta_1[g_1]} s_2 \ldots s_{n-1} \xrightarrow{\delta_{n-1}[g_{n-1}]} s_n$$

In Cases 1, 2, 3 and 4 below, we show that the final output of Algorithm 1 given by C has no bad states and has no transitions that result in bad states. That is, after applying Algorithm 1 to C^0 results in the elimination of all bad states and transitions that lead to bad states. We show that (2) is never the case. There are three possible cases based on control decision and the kind of transition during the synthesis of C from Algorithm 1.That is, given a transition T which leads to a bad state in C, then after applying Algorithm 1 T would be eliminated.

(I) Case 1 (Static controllability)
We assume that T is a static transition which leads to a bad state in C and we consider the case where T is uncontrollable and the case where T is controllable. In the case that T is an uncontrollable transition, it implies that T is not enabled in $\mathcal{T}^W$ due to the fact that $\mathcal{T}^W$ is controllable. Now since T is not enabled in $\mathcal{T}^W$, it follows that Algorithm 1 would have eliminated T during the construction of C. Hence it would be a contradiction if it holds that T is in C. A similar situation prevails in the case that T is a controllable transition and leads to a bad state. It follows that T is not enabled in $\mathcal{T}^W$, hence Algorithm 1 would have disabled T to prevent it from occurring. Hence, it contradicts the hypothesis that T is in C. We formally prove the above two cases in the following two steps, respectively.

- Suppose that T is a static transition where $g_n = true$ and let $\delta_n \in \Sigma_{uc}$ Then, from (2) we have

$$\begin{aligned}
((s_n, t_n), s_n) \xrightarrow{\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes \mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\
\implies (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_C \text{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\
\text{(from the definition of } \otimes) \\
\implies t_n \xrightarrow{\delta_n} t_{n+1} \notin \Gamma_{\mathcal{T}^W} \\
\text{(Since } \mathcal{T}^W \text{ is controllable, any bad state and transition leading to a bad state which is not in } C \text{ is also not in } \mathcal{T}^W) \\
\implies \delta_n \notin E^{\mathcal{S}}_{\mathcal{T}^W}(t_n) \\
(\delta_n \text{ will not be enabled at } t_n \text{ of } \mathcal{T}^W \text{ (which violates the definition of static controllability))} \\
\implies T \textit{ would have been eliminated from } C \textit{ by algorithm 1 (Lines 3-15)} \\
\text{(hence it will be a contradiction to say that } (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_{C \otimes \mathcal{G}_W})
\end{aligned}$$

This can be seen from the construction of C by algorithm 3 (Lines 3-15), where any uncontrollable transition T enabled at s_n in $\mathcal{G}_W$ but not enabled at a corresponding state (s_n, t_n) of C will lead to the elimination of (s_n, t_n) from the states of C (Line 12). The state (s_n, t_n) will not even exist in the set of states of C.

- Similarly, suppose that T is a static transition where $g_n = true$ and let $\delta_n \in \Sigma_c$, then from (2) we have

$$\begin{aligned}((s_n, t_n), s_n) \xrightarrow{\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\ \implies (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_C \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\ \text{(from the definition of } \otimes) \\ \implies t_n \xrightarrow{\delta_n} t_{n+1} \notin \Gamma_{\mathcal{T}^W} \\ \text{(any static controllable transition that leads to a bad state in } C \text{ is disabled in } \mathcal{T}^W) \\ \implies \delta_n \notin E^S_{\mathcal{T}^W}(t_n) \qquad (\delta_n \text{ will not be enabled at } t_n \text{ of } \mathcal{T}^W) \\ \implies T \textit{ will be disabled by Algorithm 3 (Line 19)} \qquad (\text{Since } \delta_n \in \Sigma_c) \\ \implies \textit{It will be a contradiction to say that } T \in \Gamma_{C\otimes\mathcal{G}_W}\end{aligned}$$

  That is, T will be disabled if it leads to a bad state. Hence, no controllable transition will end up in a bad state.

(II) Case 2 (Dynamic controllability 1, stronger guards generation)
Here we assume that T is a dynamic type 1 transition which leads to a bad state in C and we consider the case where T is uncontrollable and the case where T is controllable. In the case that T is an uncontrollable transition, it implies that T is either not allowed in $\mathcal{T}^W$ at all or it allowed but the guards on both transitions are not satisfied as a result of the controllability of $\mathcal{T}^W$. It follows that Algorithm 1 would have strengthened the guard on T during the construction of C and making the guard on T stronger, which implies that the state (s_{n+1}, t_{n+1}) is not reachable. It follows that T is not a valid transition. Hence, it would be a contradiction if it holds that T is in C. A similar scenario holds in the case that T is a controllable transition and leads to a bad state. It follows that T is not allowed in $\mathcal{T}^W$ at all or the guards on both transitions do not agree. Now since T is controllable, Algorithm 1 would have disabled T when the guards are not satisfiable preventing it from occurring. Hence, it contradicts the hypothesis that T is in C. We formally prove the above two cases in the following two steps, respectively.

- Suppose that T is a dynamic type 1 transition and $\delta_n \in \Sigma_{uc}$ *(e.g., in Example 6.1 take T to be the transition $((s_2, t_2) \xrightarrow{atom_1(x,y)[3\le x<7]} (BAD_1))$ of Figure 10(c)).*
  Then, from (2) we have,

$$\begin{aligned}((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\ \implies (s_n, t_n) \xrightarrow{\delta_n[g_n]} (s_{n+1}, t_{n+1}) \in \Gamma_C \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad} \\ \text{(from the definition of } \otimes) \\ \implies \nexists (t_n \xrightarrow{\delta_n[g'_n]} t_{n+1}) \in \Gamma_{\mathcal{T}^W} \textit{ such that } (g'_n \wedge g_n) \textit{ is satisfiable} \\ \text{(since } \mathcal{T}^W \text{ is controllable and the from the contruction of } C) \\ \implies (\delta_n, g_n) \notin E^{\mathcal{D}}_{\mathcal{T}^W}(t_n) \qquad (T \text{ would not be enabled at the state } t_n \text{ of the } \mathcal{T}^W) \\ \implies \textit{the guard } g_n \textit{ of } C \textit{ would have been strengthened and transition } T \textit{ eliminated}\end{aligned}$$

By Algorithm 5 (Lines 28-37), the guards of C are strengthened to that of $\mathcal{T}^W$, in particular the guard g_n on $T = ((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W}$ will never be true and the state (s_{n+1}, t_{n+1}) will be eliminated. Hence, it will be a contradiction to say that $T \in \Gamma_{C\otimes\mathcal{G}_W}$

- Similarly, suppose that T is a dynamic transition of type 1 and let $\delta_n \in \Sigma_c$, then from (2) we have

$$\begin{aligned}((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}\\ \implies (s_n, t_n) \xrightarrow{\delta_n[g_n]} (s_{n+1}, t_{n+1}) \in \Gamma_C \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}\\ \text{(from the definition of } \otimes)\\ \implies \nexists (t_n \xrightarrow{\delta_n[g'_n]} t_{n+1}) \in \Gamma_{\mathcal{T}^W} \textit{ such that } (g'_n \wedge g_n) \textit{ is satisfiable})\end{aligned}$$

(i.e., a controllable transition that leads to a bad state in C means either the transition is in $\mathcal{T}^W$ but the guards are not satisfied for all values or the transition is not in $\mathcal{T}^W$ at all)

$$\begin{aligned}&\implies T \textit{ will be disabled by Algorithm 6 (Line 4)} && \text{(since } \delta_n \in \Sigma_c)\\ &\implies \textit{It will be a contradiction to say that } T \in \Gamma_{C\otimes\mathcal{G}_W}\end{aligned}$$

(III) Case 3 (Dynamic controllability type 2, event enforcement)
Suppose that T is a dynamic type 2 transition which leads to a bad state in C . In this particular case (Case 3) the guard on T has a variable whose value depends on the output of an atomic operation (e.g., in Figure 7). We consider separately the case where T is uncontrollable and the case where T is controllable.

In the case that T is an uncontrollable transition, it implies that T is either not allowed in $\mathcal{T}^W$ at all or it allowed but the guards on both transitions are not satisfied as a result of controllability of $\mathcal{T}^W$. Now by controllability of event enforcement, there must exist another enforceable transition, say T', also enabled at the same state as T to be used to preempt T during runtime. This means that (s_{n+1}, t_{n+1}) is unreachable. It follows that T is not a valid transition in C. This contradicts the hypothesis that T is in C.

Similarly, consider the case that T is a controllable transition and leads to a bad state. It follows that T is not allowed in $\mathcal{T}^W$ at all or the guards on both transitions do not agree. Now since T is a controllable transition, Algorithm 1 would have disabled T when the guards were not satisfiable preventing it from occurring. Hence, it contradicts the hypothesis that T is in C. We formally prove the above two cases in the following two steps, respectively.

- Recall that $g_n^A(v)$ denote a guard g_n with a variable v whose values depend on an output of an atomic operation A of a given transition system. Suppose that T is a dynamic type 2 transition where $g_n = g_n^A(v)$ and let $\delta_n \in \Sigma_{uc}$ *(e.g., in Example 6.1 take T to be the transitions*

$(s_4, t_4) \xrightarrow{atom_2(x,y)[y=a]} (BAD_2)$ *of Figure 10(c)).* Then, from (2) we have,

$$((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n^A(v)]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}$$

$$\implies (s_n, t_n) \xrightarrow{\delta_n[g_n^A(v)]} (s_{n+1}, t_{n+1}) \in \Gamma_C \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}$$

(from the definition of $\otimes$)

$$\implies \nexists (t_n \xrightarrow{\delta_n[(g_n^A(v))']} t_{n+1}) \in \Gamma_{\mathcal{T}^W} \textit{ such that } ((g_n^A(v))' \wedge g_n^A(v)) \textit{ is satisfiable}$$

(since $\mathcal{T}^W$ is controllable and from the construction of C)

$$\implies (\delta_n, g_n^A(v)) \notin E^{\mathcal{D}}_{\mathcal{T}^W}(t_n) \qquad (T \text{ is not enabled at the state } t_n \text{ of } \mathcal{T}^W)$$

$$\implies \exists\, \delta_n' : \delta_n' \in E^{\mathcal{D}}_{\mathcal{T}^W}(t_n) \textit{ and } \delta_n' \in \Sigma_f$$

by Corollary 4.1, i.e., from the controllability of dynamic transition of type 2 using event enforcement. Otherwise Algorithm 1 (Lines 46-47) would have eliminated $(s_n, t_n) \xrightarrow{\delta_n[g_n^A(v)]} (s_{n+1}, t_{n+1})$ from C. Since there is an enforceable event δ_n' also enabled at state (s_n, t_n) of the controller, the state (s_{n+1}, t_{n+1}) is not reachable because δ_n' would be used to preempt T at runtime

$$\implies \textit{It will be a contradiction to say that } T \in \Gamma_{C\otimes\mathcal{G}_W}$$

This can be seen from Lines 39-51 of algorithm 5, if there is a state (s_n, t_n) in the set of states of C with a transition T whose guard is given by $g_n^A(v)$ and leads to a bad state, it implies that there must be an enforceable transition T' exiting (s_n, t_n), otherwise the state (s_n, t_n) is removed from the states of C. Hence, it will be a contradiction to say that $T \in \Gamma_{C\otimes\mathcal{G}_W}$. Algorithm 5 monitors the variable v at runtime and ensures that δ_n' is enforced.

- Similarly, suppose that T is a dynamic transition of type 2 where $g_n = g_n^A(v)$ and let $\delta_n \in \Sigma_c$, then from (2) we have

$$((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n^A(v)]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C\otimes\mathcal{G}_W} \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}$$

$$\implies (s_n, t_n) \xrightarrow{\delta_n[g_n^A(v)]} (s_{n+1}, t_{n+1}) \in \Gamma_C \textit{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}_C^{Bad}$$

(from the definition of $\otimes$)

$$\implies \nexists (t_n \xrightarrow{\delta_n[(g_n^A(v))']} t_{n+1}) \in \Gamma_{\mathcal{T}^W} \textit{ such that } ((g_n^A(v))' \wedge g_n^A(v)) \textit{ is satisfiable}$$

(a dynamic type 2 controllable transition that leads to a bad state in C means either the transition is in $\mathcal{T}^W$ but the guards are not satisfied for all values or the transition is not in $\mathcal{T}^W$ at all)

$\implies$ *T will be disabled by Algorithm 6 (Line 4)* (Since $\delta_n \in \Sigma_c$)

$\implies$ *It will be a contradiction to say that* $T \in \Gamma_{C \otimes \mathcal{G}_W}$

(IV) Finally, in the computation of C by Algorithm 1 (Line 13), any new transition that was created as a result of the checking for controllability is removed.

From cases I, II, III, IV, we have shown that any transition $T \in \Gamma_{C^0}$ that leads to a bad state is eliminated from the set of transitions of C^0 upon termination of Algorithm 1. Thus, the final output of Algorithm 1 given by C has no bad states and has no transitions that lead to a bad state. This implies that every transition in the controlled system $C \otimes \mathcal{G}_W$ leads to a good state which can be matched by $\mathcal{T}^W$. Hence, $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$

In the following we prove the reverse direction of the theorem.

Part 2
Only if: Assume $\mathcal{T}^W \preceq \mathcal{G}_W$ and $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$
To prove:
$\mathcal{T}^W$ is controllable

To prove the claim we shall consider two cases based on the type of transitions and the given assumption $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W$ to prove controllability. We note that no matter, what algorithm is used to construct C the theorem must be satisfied.

- Static Case:
  Here we prove the first part of controllability (i.e., controllability of static transitions) by using the fact that if there is a simulation relation between $C \otimes \mathcal{G}_W$ and $\mathcal{T}^W$, then it implies every static transition that is enabled in $C \otimes \mathcal{G}_W$ is also enabled in $\mathcal{T}^W$, which proves static controllability. We show this formally as follows.

  Given $C \otimes \mathcal{G}_W \preceq \mathcal{T}^W \implies$
  [*there exists a relation* $R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W}$ *such that* $\forall((p,r),q) \in R$,
  if $(p,r) \xrightarrow{\delta} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists\, q'$ *such that* $q \xrightarrow{\delta} q' \in \Gamma_{\mathcal{T}^W}$ *and* $((p',r'),q') \in R$]
  (from the definition of simulation relation)

  $\implies$ $[\forall \delta \in \Sigma, ((p,r),q) \in R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W}$:
  $(p,r) \xrightarrow{\delta} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists\, q'$ *such that* $q \xrightarrow{\delta} q' \in \Gamma_{\mathcal{T}^W}$
  and $((p',r'),q') \in R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W}]$

  $\implies$ $[\forall \delta \in \Sigma_{uc} : (p,r) \xrightarrow{\delta} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists\, q'$ *such that* $q \xrightarrow{\delta} q' \in \Gamma_{\mathcal{T}^W}]$ (since $\Sigma_{uc} \subseteq \Sigma$)

  $\implies$ $[\forall \delta \in \Sigma_{uc} : \delta \in E^{\mathcal{S}}_{C\otimes\mathcal{G}_W}(p,r) \implies \delta \in E^{\mathcal{S}}_{\mathcal{T}^W}(q)]$

- Dynamic Case:
  Similarly, we prove the second part of controllability (i.e., dynamic controllability) by using the fact that if there is a simulation relation between $C \otimes \mathcal{G}_W$ and $\mathcal{T}^W$, then it implies every dynamic transition that is enabled in $C \otimes \mathcal{G}_W$ is also enabled in $\mathcal{T}^W$ and the guards are satisfied. In this case, no matter how C was constructed and whether the construction uses event enforcement techniques or not, it must ensure that the guards on both transitions of $C \otimes \mathcal{G}_W$ and $\mathcal{T}^W$ are satisfiable. We show this formally as

follows by considering both dynamic 1 and dynamic 2 transitions together.

$$\text{Given } C \otimes \mathcal{G}_W \preceq \mathcal{T}^W \implies$$

$$[\textit{there exists a relation } R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W} \textit{ such that } \forall((p,r),q) \in R,$$

$$\textit{if } (p,r) \xrightarrow{\delta[g_1]} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists q' \textit{ such that } q \xrightarrow{\delta[g_2]} q' \in \Gamma_{\mathcal{T}^W}, \textit{where } g_1 \leq g_2$$
$$\textit{and } ((p',r'),q') \in R]$$
(from the definition of simulation relation)

$$\implies [\forall \delta \in \Sigma, ((p,r),q) \in R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W} :$$
$$(p,r) \xrightarrow{\delta[g_1]} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists q' \textit{ such that } q \xrightarrow{\delta[g_2]} q' \in \Gamma_{\mathcal{T}^W}, \textit{ where } g_1 \leq g_2$$
$$\textit{and } ((p,r),q) \in R \subseteq \mathcal{S}_{(C\otimes\mathcal{G}_W)} \times \mathcal{S}_{\mathcal{T}^W}]$$

$$\implies [\forall \delta \in \Sigma_{uc} : (p,r) \xrightarrow{\delta[g_1]} (p',r') \in \Gamma_{C\otimes\mathcal{G}_W} \implies \exists q' \textit{ such that } q \xrightarrow{\delta[g_2]} q' \in \Gamma_{\mathcal{T}^W} \textit{ where } g_1 \leq g_2]$$
(since $\Sigma_{uc} \subseteq \Sigma$)

$$\implies [\forall \delta \in \Sigma_{uc} : (\delta, g_1) \in E^{\mathcal{D}}_{C\otimes\mathcal{G}_W}(p,r) \implies [(\delta, g_2) \in E^{\mathcal{D}}_{\mathcal{T}^W}(q) \textit{ such that } (g_1 \leq g_2)]$$

From the above two steps it implies that controllability is satisfied, which completes the proof of the theorem. ■

7. Related Work

In this section, we will discuss related work and highlight the differences between existing work and our work. As there exists an extensive body of work on automatic service composition, due to space limitations we discuss only the work most relevant to ours.

We will start by looking at work that applies supervisory control theory to the SOC paradigm. As stated earlier in this paper, supervisory control theory has been applied to software systems such as concurrency in multithreaded programs and component based software systems [16, 4, 52, 51].

7.1. DES and Services

With respect to applying DES to SOA, the work of Wang et al. [53] investigates the use of supervisory control in the artifact-centric design paradigm. They present a framework to synthesize an artifact-centric process from a given set of artifacts such that a correct execution is guaranteed by properly handling uncontrollable events. However, this work relies on the standard SCT which utilizes finite state machines for modeling. In addition, this approach does not deal with data and how messages are actually exchanged among component services. Another related work is the paper by Balbiani et al. [5] which applies a variant of supervisory control theory in which system requirements are specified in modal logic to model an abstract form of service composition where non-deterministic communicating automata are used to represent Web services. The composition synthesis problem considered here is, given a community of services and a goal service, to synthesize a mediator such that the triplet client/mediator/community is equivalent to the goal service. The composition problem considered in their paper is restricted to synthesizing a specification (mediator) that realizes a given goal, but does not show how to actually orchestrate the services in terms of data and control flow requirements during execution time.

7.2. AI and Service Composition

Next, we look at work that employs AI planning techniques. A lot of AI planning based approaches have been proposed to solve the problem of automatic service composition. Pistore et al. [27, 11, 39, 24, 13] present a model-checking based planning approach that uses transition systems to model Web services that communicate by exchanging messages. The authors adopt symbolic model-checking techniques into planning in order to deal effectively with non-determinism, partial observability, and complex goals. They use these techniques to find a parallel composition of all the available services and then synthesize a controller that ensures that the composed service satisfies the given requirement by controlling it. That is, given a set of available services $\mathcal{W} = \mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_n$ and ρ describing the goal specification, they compute a controller (plan) $\mathcal{W}_c$ such that $\mathcal{W}_c \triangleright (\mathcal{W}_1 \parallel \mathcal{W}_2 ... \parallel \mathcal{W}_n) \models \rho$, where $\parallel$ is the composition operator. Their work converts OWL-S processes to state transition systems and then goals are expressed using a requirement specification language called EAGLE. Both the state transition systems and the goals are fed into an MBP planner. Even though this approach can produce correct plans, it suffers from scalability problems partly due to the way goals are expressed. Pistore et al. [41, 38] tried to solve the scalability issues in the aforementioned approach by defining an appropriate model for providing a knowledge level description of the component services which uses BPEL workflows instead of OWL-S process models. The work has been incorporated into a well-known automated service composition project called the ASTRO framework [30]. The main difference between the work by Pistore et al. and our work is that Pistore et al. make use of AI techniques in generating a controller while in our work we make use of supervisory control theory. In addition, our approach makes use of Labelled Transition Systems augmented with guards and variables while the approach by Pistore et al. make use of State Transition Systems.

Another AI planning approach is the work in [43] which presents a mixed initiative framework for semantic Web service discovery and composition which does not attempt to automate all decisions, but assumes that the users should retain close control over many decisions while having the ability to selectively delegate tedious aspects of their tasks. They used an AI planning algorithm known as GraphPlan to build their composition engine which combines rule-based reasoning on OWL ontologies with Jess (a rule engine and scripting environment for Java platforms) and planning functionalities. The main use of planning here is to provide suggestions of composition schemata to the user, instead of enforcing decisions which form the ultimate goal of this work.

Wu et al. [56] also employed graph-based planning to solve the service composition problem. The approach considers both process heterogeneity and data heterogeneity problems. They implemented their own definition of an abstract semantic Web service built on top of SAWSDL and WSDL-S. Then, they extended GraphPlan that automatically generates the control flow of a Web process. The system automatically generates an executable BPEL process from a given specification of the initial state, the goal state and a semantically annotated Web service description in SAWSDL. Data mediation is done using assignment activities in BPEL or by a data mediator which may be embedded in a middleware. At runtime the data mediator converts the available service into the format of the input message of the operation which is invoked when called by the BPEL process.

Sirin et al. [46] attempt to leverage the Hierarchical Task Network (HTN) planning techniques for the automated composition of semantic Web services. The authors are motivated to use this technique based on the fact that the concept of task decomposition in HTN planning is very similar to the concept of composite process decomposition in OWL-S process ontology. They built a system that translates OWL-S service descriptions into SHOP2 (a domain-independent HTN planning system for HTN) [33] and then they provided a method to automatically synthesize a feasible composition plan. The system is also capable of executing information-providing Web services during the planning process. They went ahead to prove the correctness of their algorithm/approach by showing the correspondence to the situational calculus semantics of OWL-S.

Peer [37] shows how the Partial Order Planner known as Versatile Heuristic Partial Order Planner (VHPOP) can be combined with re-planning algorithm for automatic service composition. They provide their own definition of semantic Web services which is then translated into PDDL as an input for VHPOP. The PDDL description of the Web service is fed into VHPOP as well as a set of links between tasks to avoid. One or more plans are automatically generated which may be partially defined. During runtime, execution is done one step at a time since the generated plan(s) does not necessarily ensure correct execution. Hence, if a plan fails, a re-planning is performed and a new plan is produced, given the conditions of the failure; however, if the execution of a plan is successful, there is no need to re-plan and one can move on to the next task.

Klusch et al. [29] present an approach similar to that of Peer. However, they built their framework on OWL-S descriptions rather than developing their own ontology service language. Similarly, the OWL-S descriptions are converted into PDDL descriptions and then fed into their AI planner. They used a hybrid AI planner known as Xplan which combines the benefits of both graph based planning and HTN. Their approach increases planning efficiency in two ways. The graph-plan based FastForward-planner always finds a composition/solution if it exists in the action state space, whereas HTN planning provides decomposition planning techniques. In addition, their planner supports re-planning components which automatically updates or reacts to changes during the composition planning process.

Recently, Sohrabi and McIlraith [48] presented an approach that supports customization, optimization and regulation enforcement during composition construction time by incorporating preferences and regulations into HTN planning. This work builds on the work in [1] by extending and customizing Golog [20] to support personalized constraints and nondeterminism in sequential executions and then, they redesigned ConGolog, the interpreter of Golog to take care of these changes. Interestingly, this development took place alongside the development of the definition of OWL-S and was one of the first works to use semantic Web services as an input to planners through translation to PDDL.

Continuing in the AI approaches, Zou and his group [60] use numerical temporal planning to tackle the problem of dynamic Web service composition which considers quality of service properties. One unique feature of this approach is that it does not rely on existing predefined workflows but it automatically generates temporal and numeral specifications from a composition task. This approach is basically made up of two steps. Firstly, a quality of service aware composition task is translated into a PDDL which is further transformed into a cost-sensitive temporally-expressive planning problem. This stage presents the service composition problem as a numeral planning problem involving time and cost optimization. Finally, the temporally expressive planning problem is solved using a SAT-based cost planning solver developed by the group. This solver deals with logical reasoning, temporal planning and optimization of composition. Other recent works using AI planning to deal with Web service composition can be found in the literature [45, 19, 32, 2, 34]. However, most of the AI planning techniques assume that the behaviour of services is deterministic and, hence, these approaches fail when unexpected events occur [6].

7.3. Other Categories of Service Composition Approaches

Other categories of Web service composition approaches are those that exploit transition systems and formal modeling languages such as Petri nets, UML and FSMs to model service composition. In the following paragraphs we discuss a number of them.

One of the earliest works in this category is by Berardi et al. [9, 8, 7, 10] who present a formal framework in which execution trees are used to describe the exported behaviour of services (an abstraction for its possible executions). These execution trees are represented using Finite State Machines. In the approach, a service is modeled using two schemata, an external and internal schema which are represented using FSMs. The external schema specifies the exported behaviour (externally-visible) of services, whereas the internal

schema contains information on which service instances execute a given action within the community of services. Their approach reduces the problem of composition synthesis into the satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL). That is, both the FSM models of the available services and the target service are encoded into DPDL, and a target service exists if and only if the set of formulas are satisfiable and then an FSM is automatically synthesized. The resulting FSM is further translated into a BPEL process and executed in a BPEL engine.

Pathak et al. [36] propose a Framework for Modeling Service Composition and Execution (MoSCoE) where both the available services and the goal service exhibit infinite-state behaviour. The approach employs Symbolic Transition Systems (STSs) to model services that are associated with guards over infinite domain variables. They use refinement analysis to guide users to refine their composition goal in the case of a failure. Typically, the framework consists of three steps, abstraction, composition and refinement. Both component services and the goal service are described using UML state machines and are translated into STSs. To this end, they apply their composition algorithms to synthesize a composition if it exists. In the case that it does not exist, the users refine their requirements and then try again.

Skogan et al. [47] also proposed an approach for semantic Web service composition using Model-Driven Development (MDD). UML is used to model Web services. They first translate WSDL descriptions into UML models. This allows existing services to be modeled using UML platforms designed for building compositions. They apply MDD techniques to generate a composition based on the UML models of the Web services, which in turn can be translated into executable BPEL specifications. Furthermore, this paper presents an open-source implementation that realizes their technique.

Jingjing et al. [26] solve the service composition problem using timed automata. A formal model built on timed automata is used to model Web services and provides an approach for automatic Web service composition. They implemented an algorithm that automatically generates a timed automaton model for each Web service interface which are all put together by synchronizing them through their branches and end tags. In this case, the equivalent graph is a topology which connects each Web service interface by an equivalence relation. The algorithm has been implemented in a composition automation engine which uses the Web service interface description language (WSIL). The Web service interface description language is a context-free grammar language developed to describe Web service interfaces. The engine is basically a compiler which takes inputs in the form of WSIL and produces outputs via semantic analysis in the form of a graph or an equivalent tree for Web service interface. The equivalent tree represents a data structure without loops which can be obtained by performing a breadth-first traversal of the equivalent graph described above. The output is then verified with a verification tool known as UPPAAL.

The work of Wang et al. [50] presents an approach in which conditional branch structures are used to model the problem of service composition. This approach supports user preferences as well as the ability to adapt to changes in a dynamic real-world environment. In order to model conditional branch structures accurately, they employ activity diagrams in UML to represent the dependencies in composite services. They consider two types of user preferences during composition synthesis. “One type is that a user prefers a class of services over another according to certain conditions (e.g., Lucy prefers to go by air over car, if the driving time is greater than 4 hours). The other type is that the user assigns priorities over services with similar functionalities” [50]. Based on these user preferences and a set of services (each specified by an activity diagram) they provide an algorithm that generates all the feasible composite services.

In another direction, the work by Yang et al. [57] adopts the extended BDI (Belief-Desire-Intention) logic to deal with the problem of Web service composition in the case where a user’s goal is not consistent with the composition goal. The Belief-Desire-Intention model is used to specify a service’s belief, desire and intention, which are mapped into the environment of BDI, the goal of the web service (and user) and composition schemes respectively. A process model is then used to characterize the results. In order to allow

for dynamic evolution of their workflow, they use AgentSpeak(L) (a communication language) to express it.

Khoumsi [28] casts the service composition problem as a control problem using a simple input-output automata-based method. The problem is to synthesize an orchestrator *Orch* from a given set of Web service $S_1, ..., S_n$ and a desired goal S_0 such that *Orch* coordinates the available services to achieve S_0. However, this approach is limited to only the input and output parameter descriptions of services and does not capture the behavioural constraints of services.

In a nutshell, relative to supervisory control theory, none of the above approaches is able to prevent a system from violating its requirements or guarantees that the system requirement would always be satisfied until a violation occurs.

8. Conclusion and Future Work

In this paper, we have developed a supervisory control framework for modeling Web service composition. We have provided a formalism based on SLTS and have also formalized the problem of Web service composition. We have described a technique to generate a composition from a given set of Web services and a specification specified in SLTS.

In future work, we will show that the controller C generated by Algorithm 1 is in fact, minimally restrictive. In this framework, we assume full observability of events but it will be of great interest to model partial observability aspects of services. That is, sometimes a service could progress from one state to another after executing some sequence of internal events or actions which cannot be observed by the controller. Hence, a new control mechanism is needed in order to prevent the system from violating any system requirements. Another direction is to research into how non-functional requirements could be incorporated into the framework. One can also look into how to efficiently represent the SLTSs for our formalism using data structures such as BDDs and process algebra and to also provide a prototype and evaluations of the proposed approach. Another general extension of our approach could be the use use decentralized control of DES to model automated choreography synthesis Web services.

9. Bibliography

[1] S. A. McIlraith and T. C. Son. Adapting Golog for Composition of Semantic Web Services. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25*, pages 482–496. Morgan Kaufmann, 2002.

[2] A. Abdullah and l. Xining. An Efficient I/O Based Clustering HTN in Web Service Composition. In *International Conference on Computing, Management and Telecommunications (ComManTel)*, pages 252–257, Ho Chi Minh City, Vietnam, Jan 2013.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. `http://msdn.microsoft.com/en-us/library/ee251594(v=bts.10).aspx#feedback`, May 2003. [Online; accessed 10-June-2014].

[4] A. Auer, J. Dingel, and K. Rudie. Concurrency Control Generation for Dynamic Threads Using Discrete-Event Systems. *Science of Computer Programming*, 82:22–43, March 2014.

[5] P. Balbiani, F. Cheikh, and G. Feuillade. Composition of Interactive Web Services Based on Controller Synthesis. In *2008 IEEE Congress on Services - Part I*, pages 521–528. IEEE Computer Society, 2008.

[6] P. Bartalos and M. Bielikov. Automatic Dynamic Web Service Composition: A Survey and Problem Formalization. *Computing and Informatics*, 30(4), 2012.

[7] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, M. Lenzerini, and M. Mecella. Modeling Data & Processes for Service Specifications in Colombo. In *Proceedings of the Open Interop Workshop on Enterprise Modeling and Ontologies for Interoperability (EMOI-INTEROP'05) Co-located with CAiSE'05 Conference*, Portugal, Porto, 13th-14th June 2005.

[8] D. Berardi, D. Calvanese, G. D. Giuseppe, R. Hull, and M. Mecella. Automatic Composition of Web Services in Colombo. In *Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems, (SEBD)*, pages 8–15, Brixen-Bressanone (near Bozen-Bolzano), Italy, 19-22 June 2005.

[9] D. Berardi, D. Calvanese, G. D. Giuseppe, R. Hull, and M. Mecella. Automatic Service Composition Based on Behavioral Descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005.

[10] D. Berardi, F. Cheikh, G. D. Giuseppe, and F. Patrizi. Automatic Service Composition via Simulation. *International Journal of Foundations of Computer Science*, 19(2):429–451, 2008.

[11] P. Bertoli, M. Pistore, and P. Traverso. Automated Composition of Web Services via Planning in Asynchronous Domains. *Artificial Intelligence: An International Journal*, 174(3-4):316–361, 2010.

[12] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, April 1983.

[13] A. Bucchiarone, M. de Sanctis, and M. Pistore. Domain Objects for Dynamic and Incremental Service Composition. In *Service-Oriented and Cloud Computing - Third European Conference, ESOCC*, pages 62–80, Manchester, UK, 2-4 September 2014.

[14] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2007.

[15] R. Diekmann and D. Weidemann. Event Enforcement in the Context of the Supervisory Control Theory. In *18th International Conference on Methods and Models in Automation and Robotics (MMAR), Midzyzdroje, Poland*, pages 783–788, Aug 2013.

[16] C. Dragert, J. Dingel, and K. Rudie. Generation of Concurrency Control Code using Discrete-Event Systems Theory. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume FSE-16, pages 146–157, 2008.

[17] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, New Jersey, 09 2004.

[18] M. Fabian and B. Lennartson. On Non-deterministic Supervisory Control. In *Proceedings of the 35th IEEE Conference on Decision and Control*, volume 2, pages 2213–2218 vol.2, Kobe, Japan, Dec 1996.

[19] M.Y. Fayyad, A. Kamel, and A. Salah. ACUAI Framework for Automatic Composition of Web Services using Gaming AI. In *Fifth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 1–6, Lebanese University, Lebanon, April 2015.

[20] A. Ferrein, S. Schiffer, and G. Lakemeyer. Embedding Fuzzy Controllers in Golog. In *IEEE International Conference on Fuzzy Systems, FUZZ-IEEE'09*, pages 894–899, Istanbul, Turkey, Aug. 2009.

[21] X. Fu, T. Bultan, and J. J. Suc. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 621–630, New York, NY, USA, 2004. ACM.

[22] T. Le Gall, B. Jeannet, and H. Marchand. Supervisory Control of Infinite Symbolic Systems using Abstract Interpretation. In *44th IEEE Conference on Decision and Control and European Control Conference. CDC-ECC '05*, pages 30–35, Dec 2005.

[23] O. Hatzi, M. Nikolaidou, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas. Semantically Aware Web Service Composition Through AI Planning. *International Journal on Artificial Intelligence Tools*, 24(01):1450015, 2015.

[24] J. Hoffmann, P. Bertoli, M. Helmert, and M. Pistore. Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection. *Computing Research Repository*, abs 1401.3470, 2014.

[25] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[26] H. Jingjing, Z. Wei, M. Pesic, Z. Xing, and Z. Dongfeng. Web Service Composition Automation Based on Timed Automata. *Applied Mathematics and Information Sciences: An International Journal*, 8(4):2017–2024, 2014.

[27] R. Kazhamiakin, A. Marconi, M. Pistore, and H. Raik. Data-Flow Requirements for Dynamic Service Composition. In *IEEE 20th International Conference on Web Services*, pages 243–250, Santa Clara, CA, USA, June 28 - July 3 2013.

[28] A. Khoumsi. A Simple Formal Method to Synthesize an Orchestrator in Web Service Composition. In *American Control Conference (ACC)*, pages 107–112, Washington, DC, USA, June 2013.

[29] M. Klusch and A. Gerber. Semantic Web Service Composition Planning with OWLS-XPlan. In *Proceedings of the 1st International Association for the Advancement of Artificial Intelligence (AAAI) Fall Symposium on Agents and the Semantic Web*, Technical Report FS-05-01, pages 55–62, Arlington VA, USA, 2005. AAAI Press.

[30] A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: The ASTRO Approach. *IEEE Data Engineering Bulletin Issues*, 31(3):23–26, 2008.

[31] Sajed Miremadi, Knut Åkesson, and Bengt Lennartson. Extraction and Representation of a Supervisor using Guards in Extended Finite Automata. In *9th International Workshop on Discrete Event Systems, WODES 08*, pages 193–199, Gothenburg, Sweden, 2008. IEEE.

[32] P. Na-Lumpoon, M.-C. Fauvet, and A. Lbath. Toward a Framework for Automated Service Composition and Execution. In *8th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, pages 1–8, Kathmandu, Nepal, December 2014.

[33] D. Nau, T. Au, O. Ilghami, U.Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20(1):379–404, December 2003.

[34] Artur Niewiadomski, Wojciech Penczek, and Jaroslaw Skaruz. A Hybrid Approach to Web Service Composition Problem in the PlanICS Framework. In A. Irfan, Y. Muhammad, F. Xavier, and Q. Carme, editors, *Mobile Web Information Systems*, volume 8640 of *Lecture Notes in Computer Science*, pages 17–28. Springer International Publishing, 2014.

[35] M. P. Papazoglou, P. Traverso, D. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007.

[36] J. Pathak, S. Basu, R. Lutz, and V. Honavar. MOSCOE: An Approach for Composing Web Services through Iterative Reformulation of Functional Specifications. *International Journal on Artificial Intelligence Tools*, 17(1):109–138, 2008.

[37] J. Peer. A POP-Based Replanning Agent for Automatic Web Service Composition. In *Proceedings of the Second European Conference on The Semantic Web: Research and Applications, Heraklion, Greece*, ESWC'05, pages 47–61, Berlin, Heidelberg, 2005. Springer-Verlag.

[38] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1252–1259, Edinburgh, Scotland, UK, July 30-August 5 2005. Professional Book Center.

[39] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *IEEE International Conference on Web Services (ICWS'05)*, pages 293–301, Orlando, FL, USA, 11-15 July 2005. IEEE Computer Society.

[40] V. Portchelvi and V. Prasanna Venkatesan. A Goal-Directed Orchestration Approach for Agile Service Composition. *International Journal of Information Technology and Computer Science (IJITCS)*, 7(3):60–67, 2015.

[41] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. ASTRO-CAptEvo: Dynamic Context-Aware Adaptation for Service-Based Systems. In *2012 IEEE Eighth World Congress on Services (SERVICES)*, pages 385–392, Hyatt Regency Waikiki Resort and Spa, Honolulu, Hawaii, USA, 24-29 June 2012. IEEE Computer Society.

[42] P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization (SICON)*, 25(1):206–230, January 1987.

[43] J. Rao, D. Dimitrov, P. Hofmann, and N. M. Sadeh. A Mixed Initiative Approach to Semantic Web Service Discovery and Composition: SAP's Guided Procedures Framework. In *IEEE International Conference on Web Services (ICWS'06)*, pages 401–410, Chicago, Illinois, USA, September 2006.

[44] G. Rodrguez, A. Soria, and M. Campo. Artificial Intelligence in Service-Oriented Software Design. *Engineering Applications of Artificial Intelligence*, 53:86 – 104, 2016.

[45] P. Rodriguez-Mier, M. Mucientes, and M. Lama. Automatic Web Service Composition with a Heuristic-Based Search Algorithm. In *IEEE International Conference on Web Services (ICWS)*, pages 81–88, Washington DC, USA, 2011.

[46] E Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377 – 396, 2004. International Semantic Web Conference 2003.

[47] D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*, EDOC '04, pages 47–57, Washington, DC, USA, 2004. IEEE Computer Society.

[48] S. Sohrabi and S. A. McIlraith. Optimizing Web Service Composition while Enforcing Regulations. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 601–617, Berlin, Heidelberg, 2009. Springer-Verlag.

[49] M. Teixeira, R. Malik, J.E.R. Cury, and M.H. de Queiroz. Supervisory Control of DES with Extended Finite-State Machines and Variable Abstraction. *IEEE Transactions on Automatic Control*, 60(1):118–129, Jan 2015.

[50] P. Wang, Z. Ding, C. Jiang, and M. Zhou. Automated Web Service Composition Supporting Conditional Branch Structures. *Enterprise Information Systems*, 8(1):121–146, January 2014.

[51] Y. Wang, H. Cho, H. Liao, A. Nazeem, T. Kelly, S. Lafortune, S. Mahlke, and S. Reveliotis. Supervisory Control of Software Execution for Failure Avoidance: Experience from the Gadara Project. In *International Workshop on Discrete Event Systems (WODES'10)*, volume 43, pages 259 – 266, Technische Universitt Berlin, 2010.

[52] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *Proceedings of the 36th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 252–263, New York, NY, USA, 2009.

[53] Y. Wang and A. Nazeem. Artifact-Centric Business Process Synthesis Framework Using Discrete Event Systems Theory. Technical report, HP Laboratories, April 2011.

[54] W.M. Wonham. Supervisory Control of Discrete-Event Systems. 2012. [`http://www.control.utoronto.ca/cgi-bin/dldes.cgi`, Online; accessed 10-Sept-2015].

[55] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

[56] Z. Wu, A. Ranabahu, K. Gomadam, A. P. Sheth, and J. A. Miller. Automatic Composition of Semantic Web Services using Process and Data Mediation. Technical report, KNO.E.SIS Center, Wright State University, Dayton, Ohio, USA, 2 2007.

[57] J. Yang, X. Zhou, J. Wang, and X. Zhu. A Novel Method for Web Service Composition Based on Extended BDI. In *IEEE 11th International Conference On Networking, Sensing and Control*, (ICNSC'14), pages 310–315, Miami, FL, USA, IEEE, April 7-9 2014.

[58] P. Zafiropulo, C. West, H. Rudin, D. Cowan, and D. Brand. Towards Analyzing and Synthesizing Protocols. *IEEE Transactions on Communications*, 28(4):651–661, Apr 1980.

[59] C. Zhou and R. Kumar. Control of Nondeterministic Discrete-event Systems for Simulation Equivalence. *IEEE Transactions on Automation Science and Engineering*, 4(3):340–349, 2007.

[60] G. Zou, Q. Lu, Y. Chen, R. Huang, Y. Xu, and Y. Xiang. QoS-Aware Dynamic Composition of Web Services Using Numerical Temporal Planning. *IEEE Transactions on Services Computing*, 7(1):18–31, Jan 2014.