Artifact-Centric Business Process Synthesis Framework Using Discrete Event Systems Theory

Yin Wang, Ahmed Nazeem

HP Laboratories
HPL-2011-40

Keyword(s):

Business Process Synthesis, Service Composition, Artifact, Supervisory Control, Discrete Event Systems

Abstract:

Artifact-centric design principle promotes business artifacts to the central role. Services in an artifact system are operations that change the state of these artifacts. A business rule specifies a condition to invoke a service, and a set of rules form a business process. Analogous to the service composition problem in Service Oriented Architecture, one can synthesize an artifact-centric process automatically from a given set of artifacts and services. However, handling uncontrollable events such as user behavior and conditional effects is a challenge. With a few exceptions that target at specific models, existing composition algorithms either assume deterministic outcome of uncontrollable events, or use the execution monitoring and replanning technique. Replanning may be late and miss the opportunity to achieve the goal. In this paper, we introduce a branch of control theory, called Discrete Event Systems (DES) theory, for process synthesis. This theory applies discrete state space models such as automata and Petri nets. The objective is to synthesize the control logic that achieves a given specification. With automaton model, the control logic is represented by state-action pairs that closely resemble rules in artifact systems. While planning techniques are usually optimistic in the sense that they search for the optimal path to achieve the goal, DES theory considers all possible paths and tries to guarantee the specification under the worst sequence of uncontrollable events. We use Google Checkout Service to illustrate our process synthesis technique.

External Posting Date: April 6, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: April 6, 2011 [Fulltext]

Artifact-Centric Business Process Synthesis Framework Using Discrete Event Systems Theory

Yin Wang[1] and Ahmed Nazeem[2]

[1] Hewlett-Packard Labs, Palo Alto
[2] Georgia Institute of Technology, Atlanta

Abstract. Artifact-centric design principle promotes business artifacts to the central role. Services in an artifact system are operations that change the state of these artifacts. A business rule specifies a condition to invoke a service, and a set of rules form a business process. Analogous to the service composition problem in Service Oriented Architecture, one can synthesize an artifact-centric process automatically from a given set of artifacts and services. However, handling *uncontrollable* events such as user behavior and conditional effects is a challenge. With a few exceptions that target at specific models, existing composition algorithms either assume deterministic outcome of uncontrollable events, or use the *execution monitoring and replanning* technique. Replanning may be late and miss the opportunity to achieve the goal. In this paper, we introduce a branch of control theory, called Discrete Event Systems (DES) theory, for process synthesis. This theory applies discrete state space models such as automata and Petri nets. The objective is to synthesize the control logic that achieves a given specification. With automaton model, the control logic is represented by state-action pairs that closely resemble rules in artifact systems. While planning techniques are usually optimistic in the sense that they search for the optimal path to achieve the goal, DES theory considers all possible paths and tries to guarantee the specification under the worst sequence of uncontrollable events. We use Google Checkout Service to illustrate our process synthesis technique.

1 Introduction

The principle of artifact-centric business process model has received increasing attention in the past decade [6, 11, 21, 8]. The formal definition has been proposed [6] and the principle has been applied to business processes in large enterprises [10]. An artifact system consists of artifacts, services, and business rules. Artifacts are data objects with states and attributes. Services are operations that change these states and attributes. The semantics of services are typically described by preconditions and effects that are logic formulae over the states and attributes. Business rules specify the conditions on which the services are invoked. A set of rules define an artifact-centric business process. Given a set of artifacts and services, without or with an incomplete set of rules, we want to automatically derive the full set of rules to achieve a given goal. This is referred

to as the business process synthesis problem [11]. In the Service-Oriented Architecture, this problem is closely related to the concept of service composition, which is heavily investigated [23, 7, 4, 3, 22, 27].

Usually the service composition problem takes as input a set of services and a composition goal. The outcome is a composite of services that satisfies the goal. The composite organizes services into control flow structures, typically in the form of a workflow. A fully automated service composition approach needs a formal model that captures the service semantics. This model restricts the choice of composition algorithms. Since there is no standard service model widely adopted in practice, a large variety of service composition algorithms exist. These algorithms can be largely divided into input/output model based, precondition/effect model based, and stateful model based [34]. The increasing popularity of the artifact-centric model could lead to the convergence of service modeling and composition methods. The goal of this paper is to propose a process synthesis framework for the artifact model.

Considerable amount of service composition work under precondition/effect models exist [23, 3, 22], and planning is a popular solution [24]. Planning algorithms usually search for one solution to achieve the goal, based on optimization criteria such as shortest path. In the presence of *uncontrollable* events such as user behavior or conditional effects, planning typically resorts to the *execution monitoring* and *replanning* mechanism. Replanning may be too late as the path chosen at the beginning is usually optimistic. Alternatively, one can examine all possible paths and synthesize a safe process that guarantees reaching the goal. This problem has been addressed under specific models, e.g., game-structure model checking for "Roman Model" [15], and customized control synthesis algorithms for open workflow Petri nets [2]. The problem has also been studied under the artifact model recently [11]. In this paper, we introduce Discrete Event Systems theory [9] to address this problem and propose an artifact-centric process synthesis framework.

Discrete Event Systems (DES) is a branch of control theory developed since the early 1980s. While classical control theory uses differential equations to model system dynamics, the theory of DES addresses systems with discrete state spaces and event driven dynamics. Popular DES modeling formalisms include automata and Petri nets. The control principle in DES is the same as in traditional control theory. First we model the system to be controlled as an automaton or a Petri net. The control specification is given in accordance to the model, e.g., regular expressions for automata or bad markings for Petri nets. The control synthesis step calculates the control logic to achieve the specification. Under the automaton model, the control logic is specified as state-action pairs, which closely resemble business rules in the artifact model. The runtime execution utilizes the classical observe-action loop to guide the system. Recently, DES control theory has been successfully applied to the safe execution of possibly flawed workflows [32], and deadlock avoidance in multithreaded programs [31, 33].

Comparing with ad-hoc composition methods, business process synthesis using DES theory has many benefits. First, the synthesis result is correct by con-

struction. If the goal cannot be achieved, the violation path is generated. Second, the synthesized process is *maximal permissiveness* in the sense that it does not restrict the system behavior unless absolutely necessary. Third, as a model-based approach, control synthesis addresses changing business requirements automatically by new control specifications. Finally, existing business rules can be modeled as control specifications. Therefore, as byproduct, our control synthesis algorithm can analyze artifact systems and verify properties such as reachability and deadlock freedom. DES theory also handles partial observability and decentralized control, which we briefly discuss in this paper.

We make the following contributions in this paper: i) the systematic introduction of the DES theory to the business process synthesis problem, ii) the development of a business process synthesis framework and the entire procedure, iii) the conversion of Google Checkout Service (GCS) into artifact models, iv) demonstration of the applicability of our framework using the GCS example, and v) a preliminary implementation of the framework in the Web2Exchange platform [30].

The rest of this paper is organized as follows. Section 2 introduces the artifact-centric model and the DES control theory. Section 3 models GCS using the artifact-centric model. Section 4 develops the process synthesis framework and procedure, using GCS as a running example. Section 5 introduces other capabilities of DES theory that are relevant to the process synthesis problem. Section 6 discusses related work and Section 7 concludes the paper with a summary.

2 Background

This section introduces a simplified artifact-centric models, automaton composition operations, and the specific DES theory we exploit in this paper, called Supervisory Control Theory.

2.1 Artifact-Centric Model

We present a simplified artifact-centric model here, based on the definition introduced in [6]. Notably, we do not include states in the artifact definition. Instead we use `enum` type attribute to capture states. This allows multiple `enum` attributes in one artifact. In accordance with this simplification, we allow attribute value checking and value assignment in preconditions and effects. But business rules cannot change the value of attributes. In addition, we aggregate the *read* and *write* sets into one *access* set for services.

Our type system consists of primitive types $\mathcal{T}_p$ that includes the enumeration type, and $\mathcal{C}$ of artifact class types. A type is an element in the union $\mathcal{T} = \mathcal{T}_p \cup \mathcal{C}$, and the domain of each type $t \in \mathcal{T}$ is $\textsc{dom}(t)$. We denote attribute set by $\mathcal{A}$, and the identifier for each class $C \in \mathcal{C}$ by $\textsc{id}_C$.

Definition 1. *(artifact) An artifact class is a tuple $(C, \boldsymbol{A}, \tau)$ where $C \in \mathcal{C}$ is a class type, $\boldsymbol{A} \subseteq \mathcal{A}$ is a finite set of attributes, $\tau : \boldsymbol{A} \to \mathcal{T}$ is a total mapping. An*

artifact object of class $(C, \boldsymbol{A}, \tau)$ *is a pair* (o, u) *where* $o \in \mathbf{ID}_C$ *is an identifier, and* u *is a partial mapping that assigns each attribute* $A \in \boldsymbol{A}$ *an element in its domain* $\textsc{dom}(\tau(A))$.

An attribute A of an artifact object x is referenced as $x.A$.

Definition 2. *(schema) A schema is a finite set* Γ *of artifact classes with distinct names such that every class referenced in* Γ *also occurs in* Γ.

We define the set of terms over a schema Γ to be: i) objects of classes in Γ, and ii) $x.A$ where x is a term and A is an attribute.

Definition 3. *(atom) An atom over a schema* Γ *is one of the following:*

1. $t_1 = t_2$, *where* t_1, t_2 *are terms in* Γ,
2. $\textsc{def}(t.A)$, *where* t *is a term in* Γ *and* A *an attribute,*
3. $\textsc{new}(t.A)$, *where* t *is a term in* Γ *and* A *an artifact typed attribute, and*
4. $t.A = val$, *where* $val \in \textsc{dom}(\tau(t.A))$ *is a value for the term* $t.A$.

A condition over Γ is a conjunction of atoms and negated atoms, whereas an effect is a set of conditions that describe different conditional outcome.

Definition 4. *(service) A service* s *over a schema* Γ *is a tuple* (n, V, P, E) *where* n *is the service name,* V *is the finite set of variables of classes in* Γ, P *is a condition over* Γ *that does not contain the atom* NEW *and* E *is the effect.*

Various semantics exist for the precondition/effect service model. We adopt the semantics defined in [6] and omit the discussion here. Notably, the semantics are not fully consistent with the OWL-S standard. However, these discrepancies are technicalities that affect only the modeling. As long as we can translate the semantics into automaton model, our process synthesis algorithm can proceed.

Definition 5. *(artifact system) An artifact system is a triple* (Γ, S, R), *where* Γ *is a schema,* S *is a set of services over* Γ, *and* R *is a set of business rules. A business rule is a pair of a condition and a service over* Γ. *The service is invoked if the condition is true.*

2.2 Automaton Model and Composition Operations

We model the system to be controlled as an automaton $G = (X, E, f, x_0, X_m)$, where X is the set of states, E is the set of event labels, partial function $f : X \times E \to X$ is the transition function, x_0 is the initial state, and X_m is the set of terminal states. We denote the regular language generated by G as $\mathcal{L}(G)$, and the language marked by G is $\mathcal{L}_m(G)$; namely, $\mathcal{L}_m(G)$ consists of those strings in $\mathcal{L}(G)$ that end at a state in X_m. Event set E is partitioned into controllable and uncontrollable events $E = E_c \cup E_{uc}$. Controllable events can be prevented or postponed at run time, but uncontrollable ones cannot. Examples of controllable events include charging a credit card. On the other hand, the outcome of the charging action should be modeled as uncontrollable events, e.g., successful charge or invalid card number.

Two automaton composition operations are relevant to our discussion.

Definition 6. *(product) The product of automata* $G_1 = (X_1, E_1, f_1, x_{01}, X_{m1})$ *and* $G_2 = (X_2, E_2, f_2, x_{02}, X_{m2})$ *is an automaton* $G_1 \times G_2 := (X_1 \times X_2, E_1 \cap E_2, f, (x_{01}, x_{02}), X_{m1} \times X_{m2})$

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \textit{if both are defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

We have $\mathcal{L}(G_1 \times G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ and $\mathcal{L}_m(G_1 \times G_2) = \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2)$.

Definition 7. *(parallel composition) The parallel composition of automata* G_1 *and* G_2 *is an automaton* $G_1 || G_2 := (X_1 \times X_2, E_1 \cup E_2, f, (x_{01}, x_{02}), X_{m1} \times X_{m2})$

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), x_2) & \textit{if } f_1(x_1, e) \textit{ is defined and } e \notin E_2 \\ (x_1, f_2(x_2, e)) & \textit{if } f_2(x_2, e) \textit{ is defined and } e \notin E_1 \\ (f_1(x_1, e), f_2(x_2, e)) & \textit{if both are defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The above definitions extend to more than two automata in a natural way.

2.3 Supervisory Control Theory

Supervisory control theory (SCT) is a branch of DES control theory that deals with systems modeled as finite state automata and control specifications expressed as regular languages. It was originally proposed in the 1980s [26] and has been thoroughly studied and extended in the past three decades. This section outlines the theory and discusses its capabilities that we exploit in this paper; see [9] for a more detailed exposition of SCT.

A supervisor S of G is a function $S : \mathcal{L}(G) \to 2^E$, i.e., S maps a string in $\mathcal{L}(G)$ to a set of events to be *disabled*. These events must be defined by f and must not include any uncontrollable events. Following the control logic of S, we get a sublanguage of $\mathcal{L}(G)$, denoted as $\mathcal{L}(S/G)$. We call S/G the controlled system. The set of marked strings that survive in the controlled system is $\mathcal{L}(S/G) \cap \mathcal{L}_m(G)$ and it is denoted by $\mathcal{L}_m(S/G)$.

In general, the control specification is a regular language $K \subseteq \mathcal{L}_m(G)$ over the event set E. The supervisory control problem is to find a supervisor S such that $\mathcal{L}_m(S/G) = K$ and $\mathcal{L}(S/G) = \overline{K}$, where the overbar notation denotes the operation of prefix-closure over strings. This means that all strings of K are achieved under control and that all strings in the controlled behavior can be extended to a string in K; the latter condition is called the *non-blocking* condition. Subject to the technical condition that $K = \overline{K} \cap \mathcal{L}_m(G)$, the necessary and sufficient condition for the existence of the supervisor in the presence of uncontrollable events is:

$$\overline{K} E_{uc} \cap \mathcal{L}(G) \subseteq \overline{K} \tag{1}$$

This condition means that if a string is allowed by both G and K, any extension by uncontrollable events that are defined in G should be included in K, because we cannot block any uncontrollable event. The specification K is said to be *controllable* if (1) is satisfied.

Algorithm 1 Iterative control synthesis algorithm

Input: $G = (X, E, f, x_0, X_m)$ and specification K represented by $H = (Y, E, h, y_0, Y_m)$
Output: supremal controllable non-blocking sublanguage $K^{\uparrow C}$

1: Let $H_0 := (Y_0, E, h_0, (y_0, x_0), (Y_m \times X_m)) = H \times G$, where $Y_0 \subseteq Y \times X$
2: $i = 0$
3: **repeat**
4: $\quad i = i + 1$
5: $\quad H_i := (Y_i, E, h_i, (y_0, x_0), (Y_m \times X_m)) = H_{i-1}$
6: $\quad$ remove every state $(y, x) \in Y_i$ where $f(x, e)$ is defined for some $e \in E_{uc}$ but $h_i((y, x), e)$ is not defined
7: $\quad$ trim dead paths of H_i such that $\overline{\mathcal{L}(H_i)} = \overline{\mathcal{L}_m(H_i)}$
8: **until** $H_i = H_{i-1}$
9: $K^{\uparrow C} = \mathcal{L}_m(H_i)$

If K is controllable, we construct the supervisor S as $\forall \sigma \in \mathcal{L}(G), S(\sigma) = \{e | e \in E, \sigma e \in \mathcal{L}(G) \setminus \overline{K}\}$. If K is not controllable, we want to find the maximal subset of K such that (1) is satisfied. It can be shown that the maximal subset is unique and computable. It is called the *supremal controllable non-blocking sublanguage* of K and denoted as $K^{\uparrow C}$. This sublanguage has four key properties: (i) It satisfies (1) by construction and therefore it is achievable under control by its corresponding supervisor. (ii) It does not prevent successful termination and therefore it is non-blocking. (iii) It is *maximally permissive*, disabling transitions only when necessary to satisfy the control specification K. (iv) It is a regular language. Next we describe the algorithm that computes this language.

The control synthesis algorithm is an iterative process. Algorithm 1 illustrates this process. We start with the product automaton H_0 for automaton G and control specification K represented by automaton H, where $\mathcal{L}(H) = \overline{K}$ and $\mathcal{L}_m(H) = K$. The product operation allows mapping of states of H to those of G. The subsequent iterative process prunes H_0 until it is both controllable and non-blocking.

Controllability, as stated in (1), is satisfied at Step 6. If some uncontrollable transition is present in G but it is not permitted by K, the corresponding state in the product automaton must be removed as we cannot prevent it from firing and therefore violating the specification K. Once we remove these states, some other states in the product automaton may no longer reach any terminal state. Therefore we further trim the automaton in Step 7. Similarly, trimming the automaton can remove (uncontrollable) transitions in the product automaton and violate controllability condition (1), hence we need to iterate.

Note that Algorithm 1 may output an empty language if there are too many uncontrollable transitions or if K is too restrictive. One can modify the algorithm to output the pathological paths where K is violated. In addition, while we consider control specifications for the *maximally permitted* behavior so far, as a dual problem, the supervisory control theory can handle control specifications for the *minimally required* behavior. For example, instead of describing the entire

order processing system and capturing all corner cases, one can specify only rules to be enforced, e.g., returned order must be refunded.

Control synthesis requires time quadratic in the size of $G \times K$ in the worst case. However, control synthesis in practice usually converges in a few iterations. In addition, control synthesis is an offline operation, and it does not increase execution time. Finally, business processes are typically small, a few dozens of tasks at most. Our experience shows that automaton models and the control synthesis algorithm scale to real-world business processes [32].

3 Google Checkout Service Example

We use Google Checkout Service (GCS) as a running example in this paper. This section discusses artifacts and services we discover from the online developer's guide. There are numerous ways to model a service using the artifact model. Our intention is to demonstrate the applicability of control theory using a real business process synthesis example. Therefore we pick the simplest model and omit certain technical details.

Google Checkout is an online payment processing service that helps merchants manage their sales orders. It has around 30 RESTful style APIs that communicate between Google and the merchant through HTTP PUT and GET commands. The parameters of each API can be sent through name value pair in the HTTP request, or in a separate XML message. These APIs are designed with extreme flexibility such that merchants of various size and complexity can use the same service. The simplest case could be a lump sum payment, while the complicated case includes a fully customizable calculation system for shipping, tax, coupon, and gift certificate, and handles a complete range of operations such as credit authorization, declined payment, back order, shipping, return, and refund. This flexibility results in an inflated set of APIs and an utterly complicated system. As a result, there is a steep learning curve on using these APIs. Google estimates up to four weeks to integrate GCS with a merchant's shopping portal [1]. Maintenance is even more difficult as both the merchant's order processing system and GCS are evolving. Our strategy is to model GCS as an artifact system but with an incomplete set of business rules. These rules capture only Google's behavior, according to their specification document. We then use DES control theory to automatically synthesize the rules for the merchant, thus complete the integration.

First we derive artifacts from the XML schema. The process can be automated if we consider each XML element as an artifact, and its XML attributes naturally become the attributes of the artifact. However, for simplicity and better readability, we aggregate some of these XML elements and present only the high-level artifacts here. Figure 1 shows the top level artifact class Order and an incomplete set of its attributes. Some of these attributes are of primitive types like `String` and `enum`, the others are artifact class types. Details of three of these second level artifact classes are displayed in Fig. 1 as well.

Class Order	Class MerchantCalc	Class RiskInformation
GoogleOrderNumber: `String`	URL: `String`	avsResponse: `enum`
shoppingCart: ShoppingCart	acceptCoupon: `bool`	billingAddress: Address
merchantCalc: MerchantCalc	acceptGiftCertificate: `bool`	buyerAccountAge: `int`
orderAdjustment: OrderAdjustment		cvnResponse: `enum`
riskInformation: RiskInformation	Class OrderAdjustment	eligibleForProtection: `bool`
financialOrderState: `enum`	adjustmentTotal: `double`	ipAddress: `String`
fulfillmentOrderState: `enum`	merchantCalcSuccess: `bool`	partialCCNumber: `String`
...	...	

Fig. 1. Some artifact classes in Google Checkout Service

Each API is considered as a service in the artifact model. Its precondition and effects must be logical formulae in the form described in Definition 3. Deriving these formulae from the online document is straightforward. There are roughly two categories of APIs, information calculation and order state manipulation. Information calculation APIs change the status of certain attributes from undefined to defined. Figure 2 shows two examples. Order state manipulation APIs issued by the merchant are displayed in Table 1. These APIs affect the financialOrderState attribute and the fulfillmentOrderState attribute, abbreviated as fiState and ffState in the table, respectively. We derive this table directly from the "Financial Order States" and "Fulfillment Order States" tables online.

Service *CheckoutShoppingCart*
Access: {x: Order }
Pre: $\neg$def(x.shoppingCart) $\wedge$
$\neg$def(x.merchantCalc)
Effects:
- new(x.shoppingCart)
- new(x.shoppingCart) $\wedge$
new(x.merchantCalc)

Service *NewOrderNotification*
Access: {x: Order}
Pre: $\neg$def(x.GoogleOrderNumber)
Effects:
- def(x.GoogleOrderNumber) $\wedge$
x.financialOrderState = reviewing $\wedge$
x.fulfillmentOrderState = new

Fig. 2. Example service definition for Google Checkout Service

We can now describe the mainline checkout process. When the online shopping customer clicks the checkout button, the merchant calls *CheckoutShoppingCart* and redirects the customer to GCS page. If the merchant optionally defines the MerchantCalc artifact, Google responds with the *MerchantCalculationCallback* API immediately. After collecting customer information and optionally the merchant-specified tax and shipping cost, Google calls *NewOrderNotification* and *RiskInformationNotification*. The merchant can either charge and ship the order or cancel the order. The merchant can also cancel the order after a charge failure. If the customer rejects or returns shipping, the merchant can cancel the order after issuing a refund. According to this process, services in the artifact model are invoked by either Google or the merchant. Google specifies precisely when to invoke its commands, which we capture by business rules in the artifact

SERVICE	PRECONDITION	EFFECTS
AuthorizeOrder	x.fiState = CHARGEABLE	– x.fiState = CHARGEABLE – x.fiState = PAYMENT_DECLINED
ChargeAndShipOrder	x.fiState = CHARGEABLE	– x.fiState = CHARGED – x.fiState = PAYMENT_DECLINED
RefundOrder	x.fiState = CHARGED	
CancelOrder	x.fiState = CHARGEABLE $\vee$ x.fiState = PAYMENT_DECLINED	x.fiState = CANCELED $\wedge$ x.ffState = WILL_NOT_DELIVER
ProcessOrder	x.ffState = NEW	x.ffState = PROCESSING
DeliverOrder	x.ffState = NEW $\vee$ x.ffState = PROCESSING	x.ffState = DELIVERED

Table 1. Merchant-issued services relevant to the financial and fulfillment order states

model. Figure 3 shows two examples. The merchant must define its own rules for its APIs, in order to complete the integration. We use DES control theory to automatically synthesize these rules.

if DEF(x.merchantCalc) **invoke** *MerchantCalculationCallback*
if x.financialOrderState= REVIEWING **invoke** *RiskInformationNotification*

Fig. 3. Example business rules for Google Checkout Service

4 Artifact-Centric Business Process Synthesis Framework

This section describes the procedure of synthesizing artifact-centric business processes using the supervisory control theory. The entire procedure is implemented in the Web2Exchange platform [30]. Web2Exchange is an object-oriented platform for service management. To incorporate artifact models in the platform, we store artifacts as data objects, and services as functions. The Common Information Model (CIM) based annotation system in Web2Exchange facilitates the use of precondition/effect descriptions. For the purpose of illustration, we use Google Checkout Service (GCS) as a running example throughout this section.

4.1 Architecture

Figure 4 is the architecture of our synthesis procedure. First, we translate the artifacts and services into a set of automata. Each automaton represents some attribute of an artifact. Based on the business goal, our composition algorithm finds and integrates the relevant set of automata using the parallel composition operation. This composite automaton captures all possible behaviors in the artifact system but does not provide any business rules to guide the execution toward the goal. The control synthesis step calculates these rules automatically and represents them as state-action pairs in a supervisor. The business process

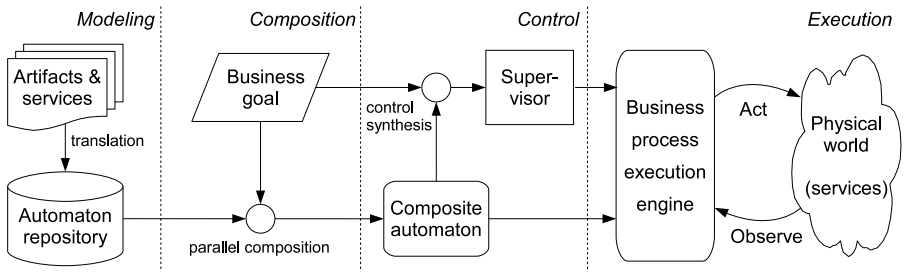

Fig. 4. Architecture

execution engine enforces these rules through the observe-act feedback cycle. Next we explain each module in detail.

4.2 Modeling

Each attribute that appears in the precondition or effect of some service is modeled by an automaton. States of the automaton represent possible values of the attribute, while transitions are services. An attribute may have an infinite number of values. Fortunately we only need to represent those that appear in the preconditions and effects. For example in GCS, most attributes such as GoogleOrderNumber have only two states, defined or undefined. Numerical types with mathematical operations can be discretized using interval arithmetics. Artifact class typed attributes can refer to any artifact instances of the same type, but typically there is a fixed number of artifact instances in the system. In the case of GCS, there is exactly one instance for every artifact class we consider. Therefore each artifact typed attribute has only two values: undefined or referring to the only instance.

Transitions are added to the automaton based on the precondition and effect of the corresponding service. For example, if a service changes an attribute from undefined to defined, we add a transition of the service's name to connect the two states. The attribute financialOrderState exhibits the most complicated automaton in our system, displayed in Fig. 5. Transitions drawn by dashed lines are uncontrollable by the merchant, which are issued by Google or controlled by the shopping customer. For example, in the PAYMENT_DECLINED state, the controllable transition is the *CancelOrder* command, while uncontrollable transitions can lead to CANCELED_BY_GOOGLE state if the customer fails to provide a valid credit card in time. Another source of uncontrollable transitions is conditional effects. For example, the *ChargeAndShipOrder* command may result in either CHARGED state or PAYMENT_DECLINED state. The command itself is controllable but the outcome is not. In this case, we add an intermediate state and split the transition into two stages. The first stage connects the source state to the intermediate state by the controllable transition that represents the service. The second stage connects the intermediate state to a set of states through un-

controllable transitions. Each of these destination states represents a possible outcome of the conditional effect.

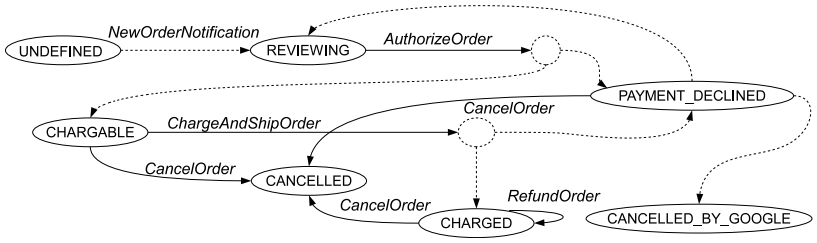

Fig. 5. The automaton for the financialOrderState attribute

As a preliminary implementation, we allow enumeration type and artifact class type attributes in Web2Exchange. Integer types are allowed but we allow only equality checking in precondition/effect formulae. This is sufficient for GCS.

4.3 Composition

The composition phase takes the business goal as the input, selects relevant automata from the repository, and uses parallel composition operation to build the composite automaton. We allow two types of business goal specifications. The first is language based, as described in Section 2.3. In this case, the specification is given as a regular expression over the alphabet of services in the artifact system. For example, *RefundOrder* must precede *CancelOrder*. The second type is state based. For example, we want to reach the state where attribute financialOrderState=CHARGED and fulfillmentOrderState=DELIVERED. Let $\mathcal{G}$ denote the set of component automata in the repository. We pick the initial set of automata $\mathcal{G}' \subseteq \mathcal{G}$ and expand the set until all relevant component automata are included for composition. In the case of language based specifications, we start with the attributes used in the preconditions and effects of the services in the regular expression. For state based specifications, we pick the attributes used to express the desirable state.

Parallel composition synchronizes automata on shared events, therefore all automata that share events with those in $\mathcal{G}'$ must be included, i.e., $\mathcal{G}' = \mathcal{G}' \cup \{G | G \in \mathcal{G} \setminus \mathcal{G}', \exists H \in \mathcal{G}', E_g \cap E_h \neq \emptyset\}$, where E_g and E_h denote the event sets of G and H, respectively. We continue expanding $\mathcal{G}'$ until no new automaton can be added. This is the basic automaton selection procedure. There is an optional pruning step that can reduce the number of component automata selected in exchange for less flexible solutions. For example, we can prune *dead states* in each component automaton, which are states not reachable from the initial state or states that cannot reach the goal state. This pruning does not reduce alternative paths in the composite to reach the goal, but the composite may become undefined should the execution lead to those dead states unexpectedly. In addition,

we can sacrifice alternative paths for a small composite automaton. As an analogy, with a composition task like map navigation, we may want only one path rather than numerous alternatives.

The computational complexity of the above algorithm depends on the size of the final composite. The parallel composition constructs the Cartesian product for the state sets of all automata involved in the operation, which dominates the computation. With many shared events among components, in practice, the state space is much smaller than the full Cartesian product. Pruning further reduces the number of automata in the final composite. With the goal of financialOrderState=CHARGED and fulfillmentOrderState=DELIVERED for GCS, our composition algorithm picked 20 component automata for the parallel composition. The composite automaton has 98 states and 134 transitions.

4.4 Control Synthesis

Given a control specification as a regular language K, we synthesize the supremal controllable non-blocking sublanguage $K^{\uparrow C}$ automatically using Algorithm 1. Here the system G is the parallel composition automaton. State-based control specifications can be translated into language-based specifications. Multiple control specifications can be unified by language intersection. Next we present a few control specification examples for GCS.

The merchant's primary goal state is financialOrderState=CHARGED and fulfillmentOrderState=DELIVERED. The control specification simply marks the corresponding states as terminal. But this specification is not controllable according to equation (1), because financialOrderState can go to the CANCELED_BY_GOOGLE state unavoidably. Using Fig. 5 as an illustration, Algorithm 1 will iteratively remove states CANCELED_BY_GOOGLE, REVIEWING, UNDEFINED, and output an empty language. We need to mark the state financialOrderState=CANCELED_BY_GOOGLE and fulfillmentOrderState$\neq$DELIVERED as terminal too. The new specification is controllable. In contrast, with the same goal, planning algorithms may choose to ship the order before charging the customer, or before the payment is confirmed, because charging and shipping are independent services. Replanning after reaching the PAYMENT_DECLINED state is too late. We may add the CHARGED state as a precondition for shipping, but this is more restrictive than the real semantics are. It is more flexible to model the semantic as is and handle various business objectives using control specifications. Another example is the execution ordering between *RefundOrder* and *CancelOrder*. The API reference allows both APIs to take place at the CHARGED state, but somewhere else in the document there is a note that *RefundOrder* must precede *CancelOrder*. This document inconsistency can be addressed by a regular expression $\{*RefundOrder*CancelOrder*\}$.

Business rules can be translated into language specifications too. We need to find the states in the composite automaton that satisfy the condition of the rule, and remove all of its outgoing transitions except the one representing the service to be invoked. The resulting automaton is the specification. If the specification is controllable, the artifact system guarantees successful termination. Otherwise

Algorithm 1 can complete the system by synthesizing a supervisor. Therefore, as byproduct, our framework can analyze the soundness of existing artifact systems, and complete the rule set if needed.

4.5 Runtime Execution

The runtime execution engine observes the system state, and invokes services as needed according to $K^{\uparrow C}$. We have discussed in Section 2.3 on how to derive a supervisor automaton from $K^{\uparrow C}$. The runtime execution need only to follow the transition rules of the supervisor. Supervisory Control Theory guarantees that the supervisor never blocks any uncontrollable transitions.

5 Extensions

This section discusses *partial observability* [19] and *decentralized control* [28] in DES theory. We do not exploit these capabilities in this paper but they are relevant to the business process synthesis problem. These concepts have been discussed in the literature with respect to different service models [14, 2].

We assume throughout this paper that all transitions are observable to the execution engine. This is referred to as *full observability* in DES theory. Extensions to the control synthesis methods we employ in this paper exist to address *partial observability*. In a partially observable system, transitions in G are either *observable* or *unobservable*. In the business process domain, we may assume that transitions within a business organization are observable, but external organizations are not, except those that are communicated intentionally. The typical solution to the control synthesis problem under partial observability involves the construction of an observer automaton that, based on observable transitions, estimates the set of states the system could possibly be in (called belief state in AI literature). Then, for every state in the estimate set, the controller disables transitions that violates the specification. Similarly to the case with only uncontrollable transitions, we desire non-blocking execution, permissive control, and other properties. After building the observer, the complexity of control synthesis is polynomial to the number of observer states. With partial observability, the maximally permissive controllable non-blocking sublanguage is no longer unique. Different control actions may result in different incomparable sublanguages.

Decentralized control is another well studied topic in DES theory. It allows decentralized entities to make their own control decisions using local observations, yet the global behavior satisfies the given control specification. This approach can be applied to business process synthesis that involve multiple organizations. Various flavors of decentralized models exist. The *communicating automata* model in the AI literature partitions the system and selects communicating events between local models. In the DES literature, the method is usually based on a global system model. The global model can be built from component models automatically via composition operations. After the control synthesis, online control decisions can be computed on-the-fly and a global model is not necessary.

6 Related Work

Existing work on artifact-centric models exhibits significant interest on the static analysis and verification of business processes [6, 12, 13]. Properties including reachability, dead path, and persistence have been studied. Recently model checking techniques have been applied to the process verification problem and decidability results are obtained [8]. The choreography problem of artifact-centric models has been studied too [21], which bears similarity with the decentralized control problem in DES. The goal in [11] is the closest to ours, which is to synthesize an artifact-centric business process with one artifact class. The paper defines the concepts γ-safe and maximal γ-safe rule sets where γ is a given synthesis goal. These concepts map to the notions *controllability* and *maximal permissiveness* in DES, respectively.

In the more general problem of service composition, there is a considerable amount of work using the precondition/effect model [23, 3, 22, 25]. The semantic markup language for web services, OWL-S, is using the precondition/effect model too. As there is no widely adopted industrial standard, the semantics of these models are slightly different. This discrepancy affects only the translation of these models into automata. The overall process synthesis procedure applies to any precondition/effect model as long as the translation exists. There are other service composition work based on the input/output model [27, 29] or stateful models [7, 5, 17].

Typical composition algorithms handle uncontrollable events by the assumption of deterministic outcome or techniques similar to runtime replanning. Notable exceptions include the work from two research groups. The first group considers the "Roman Model" that is first introduced in [5]. This model captures service dynamics by automaton, and uses a central orchestrator to *delegate* an action to a local automaton. Composition with partial controllability has been studied [15], and the work extends to partial observability as well [14]. The approach is based on the game-structure model checking, where the synthesis problem is equivalent to finding the winning strategy in a safety game. Another group considers the open workflow nets, which are acyclic workflow Petri nets augmented by input and output places. The controller is allowed to place tokens in the input places to guide the execution of the net. Centralized and decentralized controllability have been studied [2], and various flavors of decentralized control synthesis problems are discussed [20, 16]. Albeit the use of Petri net model, the synthesis algorithm is close to ours as it is based on the reachability graph and state space exploration. Comparing with these work, we consider the control synthesis problem in the artifact model. The intention is to systematically introduce Supervisory Control Theory (SCT) and propose a process synthesis framework for real services.

Our previous work applied SCT to the safe execution of possibly flawed workflows [32]. The control specification was limited to the special case of avoiding undesirable states in the automaton. In addition, the control logic must not violate the semantics of the manually composed workflow, which is often narrow. Using Petri net model, we applied a different branch of DES theory, called

Supervision Based on Place Invariants, to avoid deadlocks in multithreaded programs [31, 33]. The method successfully found and avoided real deadlock bugs in large-scale open-source software. Recently, there is considerable interest in applications to programming using control theory [18].

7 Conclusion

In this paper, we introduced the Supervisory Control Theory to address the business process synthesis problem under the artifact-centric service model with uncontrollable or nondeterministic events. We developed a process synthesis framework that translates the artifact model into a set of automata, selects relevant automata for composition, synthesizing a supervisor for a given specification, and enforces the supervisor at runtime. The synthesized process is provably correct and maximally permissive. In addition to process synthesis, the framework can analyze the correctness of existing artifact systems. Although we restrict our attention to the artifact-centric model because of its popularity and the relatively well-defined formal model. Our framework and the synthesis procedure extends to other service models, as long as the translation of the model into automata exists.

References

1. Google checkout service. `http://code.google.com/apis/checkout/developer/index.html`.
2. K. S. 0004. Controllability of open workflow nets. In *EMISA*, pages 236–249, 2005.
3. V. Agarwal, K. Dasgupta, N. M. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A service creation environment based on end to end composition of Web services. In *WWW*, pages 128–137, 2005.
4. P. Albert, L. Henocque, and M. Kleiner. A constrained object model for configuration based workflow composition. In *BPM Workshops*, pages 102–115, 2005.
5. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *ICSOC*, pages 43–58, 2003.
6. K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, pages 288–304, 2007.
7. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.
8. P. Cangialosi, G. D. Giacomo, R. D. Masellis, and R. Rosati. Conjunctive artifact-centric services. In *ICSOC*, pages 318–333, 2010.
9. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.
10. T. Chao, D. Cohn, A. Flatgard, S. Hahn, M. H. Linehan, P. Nandi, A. Nigam, F. Pinel, J. Vergo, and F. Y. Wu. Artifact-based transformation of ibm global financing. In *BPM*, pages 261–277, 2009.
11. C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *ICDT*, pages 225–238, 2009.
12. C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *SOCA*, pages 133–140, 2007.

13. C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, pages 181–192, 2007.
14. G. D. Giacomo, R. D. Masellis, and F. Patrizi. Composition of partially observable services exporting their behaviour. In *ICAPS*, 2009.
15. G. D. Giacomo and F. Patrizi. Automated composition of nondeterministic stateful services. In *WS-FM*, pages 147–160, 2009.
16. C. Gierds, A. J. Mooij, and K. Wolf. Reducing adapter synthesis to controller synthesis. *IEEE Transactions on Services Computing* , 99(PrePrints), 2010.
17. R. R. Hassen, L. Nourine, and F. Toumani. Protocol-based Web service composition. In *ICSOC*, pages 38–53, 2008.
18. M. V. Iordache and P. J. Antsaklis. Petri nets and programming: a survey. In *Proceedings of American Control Conference* , ACC'09, pages 4994–4999, Piscataway, NJ, USA, 2009. IEEE Press.
19. F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.
20. N. Lohmann and K. Wolf. Realizability is controllability. In *WS-FM*, pages 110–127, 2009.
21. N. Lohmann and K. Wolf. Artifact-centric choreographies. In *ICSOC*, pages 32–46, 2010.
22. H. Meyer and M. Weske. Automated service composition using heuristic search. In *BPM*, pages 81–96, 2006.
23. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.
24. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice* . Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
25. A. Ragone, T. D. Noia, E. D. Sciascio, F. M. Donini, and S. Colucci. Fully automated Web services orchestration in a resource retrieval scenario. In *ICWS*, pages 427–434, 2005.
26. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.* , 25(1):206–230, 1987.
27. A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In *WWW*, pages 775–784, 2008.
28. K. Rudie and W. M. Wonham. Think globally, act locally: decentralized supervisory control. 37(11):1692–1708, November 1992.
29. Z. Shen and J. Su. On completeness of Web service compositions. In *ICWS*, pages 800–807, 2007.
30. V. Srinivasmurthy, S. Manvi, R. Gullapalli, D. Sathyamurthy, N. Reddy, H. Dattatreya, S. Singhal, and J. Pruyne. Web2exchange: A model-based service transformation and integration environment. pages 324 –331, Sept. 2009.
31. Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI'08*, pages 281–294, 2008.
32. Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of IT automation workflows. In *EuroSys*, 2007.
33. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL'09*, pages 252–263, New York, NY, USA, 2009. ACM.
34. Y. Wang, H. R. Motahari-Nezhad, and S. Singhal. A language-based framework for analyzing service representation models and service composition approaches. In *IEEE International Conference on e-Business Engineering* , 2010.