University of California
Santa Barbara

Management of Data and Collaboration for Business Processes

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Yutian Sun

Committee in charge:

Professor Jianwen Su, Chair
Professor Tevfik Bultan
Professor Xifeng Yan
Doctor Richard Hull

September 2015

The Dissertation of Yutian Sun is approved.

Tevfik Bultan

Xifeng Yan

Richard Hull

Jianwen Su, Committee Chair

July 2015

Management of Data and Collaboration

for Business Processes

by

Yutian Sun

Acknowledgements

I would like to express my deep gratitude and thanks to my advisor Prof. Jianwen Su; you have been a tremendous mentor for me. I would like to thank you for your valuable and constructive suggestions and for allowing me to grow as a researcher. Your advice on both research as well as on my career have been priceless. I would also like to thank my committee members, Prof. Tevfik Bultan, Prof. Xifeng Yan, and Dr. Richard Hull for their helpful advice and suggestions in general.

Also, I would like to express my special appreciation to my colleagues in IBM Research, Roman Vaculin, Terry Heath, David Boaz, and Lior Limonad. My completion of the project during the internship could not have been accomplished without the support of you.

In addition, I am particularly grateful for the assistance given by Prof. Jian Yang and Prof. Liang Zhang, who have been supporting and collaborating with me in the area for years.

I also want to thank to National Science Foundation (IIS-0812578), IBM, and Bosch for their financial support. Moreover, my special thanks are extended to UCSB Graduate Division for granting me the dissertation fellowship.

Curriculum Vitæ

Yutian Sun

Education

2015	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2014	M.S. in Computer Science, University of California, Santa Barbara.
2010	B.S. in Information Security, Fudan University

Publications

- **Yutian Sun** and Jianwen Su: "Conformance for DecSerFlow Constraints". In *Proc. of the 12th Intl. Conf. on Service Oriented Computing* (ICSOC), Paris, France, 2014, LNCS 8831, pp. 139-153 (full paper acceptance rate: 24/160 = 15%)
- **Yutian Sun**, Jianwen Su, and Jian Yang: "Separating Execution and Data Management: A Key to Business-Process-as-a-Service (BPaaS)". In *Proc. of the 12th Intl. Conf. on Business Process Management* (BPM), Haifa, Israel, 2014, LNCS 8659, pp. 375-383 (short paper) (full paper acceptance rate: 21/123 = 17%; overall: 31/123 = 25%)
- David Boaz, Terry Heath, Manmohan Gupta, Lior Limonad, **Yutian Sun**, Richard Hull, and Roman Vaculín: "The ACSI Hub: A Data-Centric Environment for Service Interoperation". In *Proc. of the 12th Intl. Conf. on Business Process Management* (BPM) *Demo Session*, Haifa, Israel, 2014, CEUR Workshop Proceedings, Vol 1295, pp. 11-15 (demo paper)
- **Yutian Sun**, Jianwen Su, Budan Wu, and Jian Yang: "Modeling Data for Business Processes". In *Proc. of the 30th Intl. Conf. on Data Engineering* (ICDE), Chicago, USA, 2014, pp. 1048-1059 (acceptance rate: 89/446 = 20%)
- Rik Eshuis, Richard Hull, **Yutian Sun**, and Roman Vaculín: "Splitting GSM Schemas: A Framework for Outsourcing of Declarative Artifact Systems". In *Information Systems*, 2014, Vol 46, pp. 157-187
- Jianwen Su and **Yutian Sun**: "Choreography Revisited". In *Proc. of the 10th Intl. Workshop on Web Services and Formal Methods* (WS-FM), Beijing, China, 2013, LNCS 8379, pp. 13-25 (invited paper)
- Terry Heath, David Boaz, Manmohan Gupta, Roman Vaculín, **Yutian Sun**, Lior Limonad, and Richard Hull: "Barcelona: A Design and Runtime Environment for Declarative Artifact-Centric BPM". In *Proc. of the 11th Intl. Conf. on Service Oriented Computing* (ICSOC), Berlin, Germany, 2013, LNCS 8274, pp. 705-709 (demo paper)

- Rik Eshuis, Richard Hull, **Yutian Sun**, and Roman Vaculín: "Splitting GSM Schemas: A Framework for Outsourcing of Declarative Artifact Systems". In *Proc. of the 11th Intl. Conf. on Business Process Management* (BPM), Beijing, China, 2013, LNCS 8094, pp. 259-274 (full paper acceptance rate: 17/118 = 14%)
- Roman Vaculín, Richard Hull, Maja Vuković, Terry Heath, Nathaniel Mills, and **Yutian Sun**: "Supporting Collaborative Decision Processes". In *Proc. of the 10th Intl. Conf. on Services Computing* (SCC), Santa Clara, CA, USA, 2013, IEEE Computer Society, pp. 651-658
- **Yutian Sun**, Wei Xu, and Jianwen Su: "Declarative Choreographies for Artifacts". In *Proc. of the 10th Intl. Conf. on Service Oriented Computing* (ICSOC), Shanghai, China, 2012, LNCS 7636, pp. 420-434 (full paper acceptance rate: 32/185 = 17%)
- **Yutian Sun**, Richard Hull, and Roman Vaculín: "Parallel Processing for Business Artifacts with Declarative Lifecycles". In *Proc. of the 20th Intl. Conf. on Cooperative Information System* (CoopIS), Rome, Italy, 2012, LNCS 7565, pp. 433-443 (short paper)
- **Yutian Sun**, Wei Xu, Jianwen Su, and Jian Yang: "SeGA: A Mediator for Artifact-Centric Business Processes". In *Proc. of the 20th Intl. Conf. on Cooperative Information Systems* (CoopIS), Rome, Italy, 2012, LNCS 7567, pp. 658-661 (poster paper)
- **Yutian Sun** and Jianwen Su: "Computing Degree of Parallelism for BPMN Processes". In *Proc. of the 9th Intl. Conf. on Service Oriented Computing* (ICSOC), Paphos, Cyprus, 2011, LNCS 7084, pp. 1-15 (full paper acceptance rate: 30/184 = 16%)

Abstract

Management of Data and Collaboration

for Business Processes

by

Yutian Sun

A business process (BP) is a collection of activities and services assembled together to accomplish a business goal. Business process management (BPM) refers to the management and support for a collection of inter-related business processes, which has been playing an essential role in all enterprises. Business practitioners today face enormous difficulties in managing data for BPs due to the fact that the data for BP execution is scattered across databases for enterprise, auxiliary data stores managed by the BPM systems, and even file systems (e.g., definition of BP models). Moreover, current data and business process modeling approaches leave associations of persistent data in databases and data in BPs to the implementation level with little abstraction. Implementing business logic involves data access from and to database often demands high development efforts.

In the current study, we conceptualize the data used in BPs by capturing all needed information for a BP throughout its execution into a "universal artifact". The conceptualization provides a foundation for the separation of BP execution and BP data. With the new framework, the data analysis can be carried out without knowing the logic of BPs and the modification of the BP logics can be directly applied without understanding the data structure.

Even though universal artifacts provide convenient data access for processes, the data is yet stored in the underlying database and the relationship between data in artifacts

and the one in database is still undefined. In general, a way to link the data of these two data sources is needed. we propose a data mapping language aiming to bridge BP data and enterprise database, so that the BP designers only need to focus on business data instead of how to manipulate data by accessing the database. We formulate syntactic conditions upon specified mapping in order that updates upon database or BP data can be properly propagated.

In database area, mapping database to a view has been widely studied In recently years, data exchange method extends the notion of database views to a target database (i.e., multiple views) by using a set of conjunctive queries called "tuple generating dependency" (tgd). Tgd is a language that is easy to understand/specify, expressive, and decidable for a wide range of properties, which is ideal as a mapping language. Naturally, if both enterprise database and artifacts are represented as relational database, we can take advantage of data exchange technology to bridge enterprise database and artifacts by using tgd as well. Therefore, we re-visit the mapping and update propagation problem under the relational setting.

In addition to the data management for a single BP, it is equivalently essential to understand how messages and data should be exchanged among multiple collaborative BPs. With the introduction of artifacts, data is explicitly modeled that can be used in a collaborative setting. Unfortunately, today's BP collaboration languages (either orchestration or choreography) do not emphasize on how data is evolved during execution. Moreover, the existing languages always assume each participant type has a single participant instance. Therefore, a declarative language is introduced to specify the collaboration among BPs with data and multiple instances concerned. The language adopts a subset of linear temporal logics (LTL) as constraints to restrict the behavior of the collaborative BPs.

As a follow-up study, we focus on the satisfiability problem of the declarative BP

collaboration language, i.e., whether a given specification as a set of constraints allows at least one finite execution. Naturally, if a specification excludes every possible execution, it should be considered as an undesirable design. Therefore, we consider different combination of the constraint types and for each combination, syntactic conditions are provided to decide whether the given constraints are satisfiable. The syntactic conditions automatically lead to polynomial testing methods (comparing to PSPACE-complete complexity of general LTL satisfiability testing).

Contents

Chapter 1

Introduction

A business process (BP) is a collection of activities and services assembled together to accomplish a business goal. Business process management (BPM) refers to the management and support for a collection of inter-related business processes. The need for BPM is ubiquitous as BPs exist in all types of organizations including governments, healthcare, business, and more. A BPM system (a.k.a. workflow system) is a piece of software to aid BPM through automating many management functions.

Business practitioners today face enormous difficulties in managing data for BPs due to the fact that the data for BP execution is scattered across databases for enterprise, auxiliary data stores managed by the BPM systems, and even file systems (e.g., definition of BP models). Moreover, current data and business process modeling approaches leave associations of persistent data in databases and data in BPs to the implementation level with little abstraction. Implementing business logic involves data access from and to database often demands high development efforts.

In most cases, a process modeling language does not emphasize on how data is accessed. Therefore the needed data in a process is stored in database in an ad-hoc way. Fig. 1.1 (where the upper part is a BPMN process [1] and the lower part is the database)

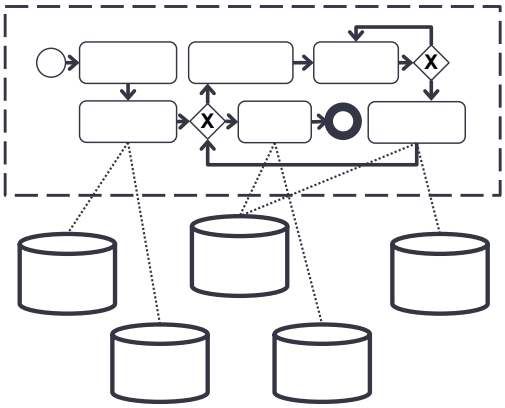

Figure 1.1: Traditional way of data access

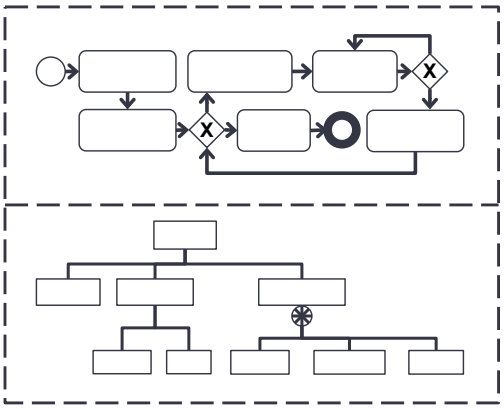

Figure 1.2: Access data through artifacts

demonstrates how in general each activity in a process access data through hard-coded SQL statements. This design suffers several drawbacks including inconvenient to change a process, business-level operation to be implemented deep in programming level, and hard for auditing since data is scattered around.

Regarding the ad-hoc way of data access, if we have a piece of data model that is recording all the needed data in a structured format, then the design would be much cleaner as the process does not need to concern about the database but only on the data model. Another advantage of having a data model is to shift burden of data design from programmers to business people, who are more familiar with what and how data should be used in business processes. Moreover, data model also provides a formal structure that helps business people to analyze, manage, and control their business operations from day to day. In 2003, a model called "artifacts" [2], or also known as "business entity with lifecycle" was proposed to provide a data model for a process. Essentially, an artifact is a marriage of processes (i.e., lifecycle) and a piece of information (i.e., business entity) that records all needed business data so that the execution of the process only needs to focus on this piece of data instead of the database. Fig. 1.2 presents the high-level view of

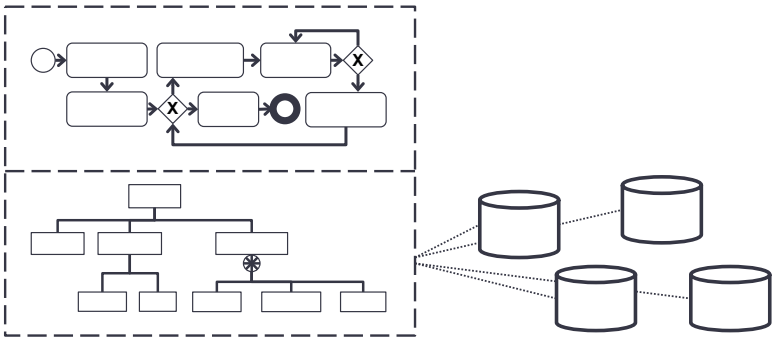

Figure 1.3: Bridge database and artifacts

an artifact, where the lower part is an XML-like data structure, called "business entity", with a lifecycle (in this case, a BPMN process) associated. The business entity collects all the data in a structured format for its corresponding lifecycle to read and write. Note that in general, there is no restriction on what format a business entity or a lifecycle should be. Actually through out the chapter, we do not have a specific lifecycle model and the business entity can be both hierarchical or relational.

However, with the introduction of artifacts, several issues are raised, which are also the main focus of this thesis. In the following, we briefly introduce each of the problems and how we approach them.

1. (Mapping between database and artifacts) Even though artifacts provide convenient data access for processes, the data is yet stored in the underlying database and the relationship between data in artifacts and the one in database is still undefined. In general, a way to link the data of these two data sources is needed (Fig. 1.3). In Chapter 3, we propose a data mapping language aiming to bridge BP data and enterprise database, so that the BP designers only need to focus on business data instead of how to manipulate data by accessing the database. We formulate syntactic conditions upon specified mapping in order that updates upon database or BP data can be properly propagated.

2. (Mapping between relational databases) In database area, mapping database to a view has been widely studied ([3, 4, 5]). In recently years, data exchange ([6, 7]) method extends the notion of database views to a target database (i.e., multiple views) by using a set of conjunctive queries called "tuple generating dependency" (tgd). Tgd is a language that is easy to understand/specify, expressive, and decidable for a wide range of properties, which is ideal as a mapping language. Naturally, if both enterprise database and artifacts are represented as relational database, we can take advantage of data exchange technology to bridge enterprise database and artifacts by using tgd as well. In Chapter 4, we re-study the mapping and update propagation problem under the relational setting.

3. (Universal artifacts) The previous two problems concern how business data (the data in a business entity) is managed for processes. However, the concept of data in a process can be much broader than business data. Data like execution status, schema definitions, correlation information, or even BP engine data is still scattered around within BP engine or local database. The ad-hoc way of storing these types of data combines a process with its engine tightly that is difficult for providing services, data analysis, or changes. In Chapter 5, we conceptualize the data used in BPs by capturing all needed information for a BP throughout its execution into a "universal artifact". The conceptualization provides a foundation for the separation of BP execution and BP data. With the new framework, the data analysis can be carried out without knowing the logic of BPs and the modification of the BP logics can be directly applied without understanding the data structure.

4. (Collaboration for artifacts) In addition to the data management for a single BP, it is equivalently essential to understand how messages and data should be exchanged among multiple collaborative BPs. With the introduction of artifacts, data is explicitly modeled that can be used in a collaborative setting. Unfortunately, today's BP collab-

oration languages (either orchestration or choreography [8]) do not emphasize on how data is evolved during execution. Moreover, the existing languages always assume each participant type has a single participant instance. Therefore, in Chapter 6, a declarative language is introduced to specify the collaboration among BPs with data and multiple instances concerned. The language adopts a subset of linear temporal logics (LTL) [9] as constraints to restrict the behavior of the collaborative BPs.

5. (Collaboration satisfiability) As a follow-up study, we focus on the satisfiability problem of the declarative BP collaboration language, i.e., whether a given specification as a set of constraints allows at least one finite execution. Naturally, if a specification excludes every possible execution, it should be considered as an undesirable design. Therefore, in Chapter 7, we consider different combination of the constraint types and for each combination; syntactic conditions are provided to decide whether the given constraints are satisfiable. The syntactic conditions automatically lead to polynomial testing methods (comparing to PSPACE-complete complexity of general LTL satisfiability testing).

The thesis addresses the above 5 topics in 5 different chapters (3 - 7), where Chapter 2 introduces the preliminary concepts of relational database and artifacts, Chapter 8 is the related work, and Chapter 9 concludes the thesis. Though there is a loose linkage among the chapters, they should be rather self-contained and can be skipped when necessary only with an exception of the pre-requisite of Chapter 2. Fig. 1.4 presents a recommended dependency and reading sequence of this thesis by chapter numbers.

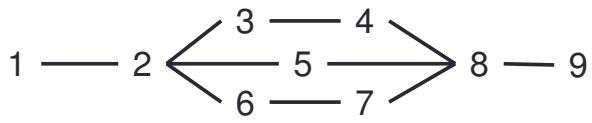

Figure 1.4: Dependency of chapters

Chapter 2

Database, Artifacts, and Business Entities

This section provides key notions of the relational data model [10] [11] that has been widely used in enterprises and the information model (a.k.a. business entity) of an artifact-centric business process [2] that elevates data as the first-class citizen for process design (our formalism is closer to the one defined in [12]).

For the technical development, we assume a totally ordered set of *names* that are used as attribute names, relation names, and names in the artifact model presented later in the section. Every (finite) set of names is enumerated according to this total order. Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^+ = \mathbb{N} - \{0\}$.

2.1 Database Models

The database model introduced in this section may or may not include "keys" and "foreign keys' (depeding on the context) with cardinality bounds that resemble the entity-relationship model [13] with cardinalities. Fig. 2.1 shows a high-level view of the concepts

Concepts	Meaning
relation symbol	a relation "type" that only has a name and arities
relation schema	a relation symbol with attribute names
database schema	a set of relation symbol or schemata
relation/database schema with keys	a relation/database schema with keys, where their instances should satisfy the key constraints
relation/database schema with keys and foreign keys	a relation/database schema with keys and foreign keys, where their instances should satisfy the key, foreign key, and cardinality constraints

Figure 2.1: High-level description of database models

in this section.

Relational Models

A *relation symbol* R is a name, which has a fixed *arity* in $\mathbb{N}$.

Definition: A *relation schema* is a tuple (R, A), where R is a relation symbol, A a finite ordered set of names for (*primitive*) *attributes*, such that $R \notin A$ and $|A|$ equals to the arity of R.

Definition: A *database schema* (or a *database schema with attributes*) $\mathbb{R}$ is a finite set of relation symbols (resp, relation schemata).

Let two disjoint countably infinite sets Const and Var be a set of *constants* and a set of (*labeled*) *nulls* respectively, where a labeled null is used to denote uncertain values [7]. Let $\textbf{Dom}$ be $\mathsf{Const} \cup \mathsf{Var}$.

Definition: Given a relation symbol R or a relation schema (R, A), a *tuple* τ (of R) is a sequence (i.e., ordered bag/multiset) of values in $\textbf{Dom}$, such that $|\tau|$ equals to the arity of R.

Given a relation schema (R, A), the i^{th} attribute $a \in A$ ($i \in [1..|A|]$), and a tuple τ of R, denote $\tau(a)$ to be the i^{th} value in τ.

Given a relation symbol R or a relation schema (R, A) A *relation* (*instance*) (of R) is a finite set of tuples.

Definition: Given a database schema $\mathbb{R} = \{R_1, R_2, ..., R_n\}$, a *database* (*instance*) I is a set $\{R_1^I, R_2^I, ...R_n^I\}$, such that for each $i \in [1..n]$, R_i^I is a relation of R_i. A ground database is a database that only contains values in Const.

Key, Foreign Key, and Cardinality Constraints

Definition: A *relation schema with keys* is a tuple (R, A, K), where (R, A) is a relation schema and $K \subseteq 2^A$ a set of *keys* over R such that no single key is properly contained in another. Each attribute in a key in K is called *prime*.

The tuple for a relation schema with keys is defined the same as the one without keys.

Given R as a relation schema with keys, the concept of "relation (instance)" is formulated in the same way (as for relation schema/symbol), such that the key values are pairwise distinct (unique).

Given a relation schema (R, A) (or a relation schema with keys (R, A, K)), we may denote a relation schema it as $R(A)$ (resp. $R(A, K)$) or simply R when it is clear from the context, A as $\textsc{Att}(R)$, and K as $\textsc{Keys}(R)$. We omit types of attributes (that can be easily added). Thus we assume the existence of a universal domain **Dom**.

The definition of *database schema with keys*, together with its (ground or non-ground) *database* (*instance*) *with keys* is the similar to the ones for database schema and database.

In the remainder of this section, we incorporate foreign keys and cardinality constraints to database.

A database schema with keys and foreign keys consists of a set of relation schemata with keys, and a set of "foreign keys", each having a cardinality bound. Similar to cardinality constraints in the ER model [13], a bound limits the number of occurrences of a foreign key value.

Definition: A *database schema with keys and foreign keys* is a triple $(\mathbb{R}, F, \lambda)$, where

- $\mathbb{R}$ is a set of relation schemata with keys with distinct names and pairwise disjoint attribute sets. Let $\textsc{Keys}(\mathbb{R}) = \cup\{\textsc{Keys}(R) \mid R \in \mathbb{R}\}$ be the set of all keys in $\mathbb{R}$.
- $F \subseteq (\bigcup_{R\in\mathbb{R}} 2^{\textsc{Att}(R)}) \times \textsc{Keys}(\mathbb{R})$ is a set of *foreign keys* such that (1) for each pair (S, κ) in F, $|S| = |\kappa|$, S is not a proper superset of a key in $\textsc{Keys}(\mathbb{R})$, and if S is a subset of attributes in a relation schema R and $\kappa \in \textsc{Keys}(R')$, then $R' \neq R$, and (2) the graph $(\bigcup_{R\in\mathbb{R}} 2^{\textsc{Att}(R)}, F)$ is acyclic.
- $\lambda\colon F \to \{1, ?, +, *\}$ is a total mapping assigning a *cardinality bound* to each foreign key, where for each foreign key f, $\lambda(f)$ limits the number of occurrences of each f-value with 1 stands for exactly once, ? for at most once, + for at least once, and $*$ for unrestricted.

Definition: Given a database schema with keys and foreign keys $\mathbf{S} = (\mathbb{R}, F, \lambda)$, a *database* (*instance*) *with keys and foreign keys* of $\mathbf{S}$ is a total mapping d from $\mathbb{R}$ to instances of $\mathbb{R}$ such that for each $R \in \mathbb{R}$, $d(R)$ is an instance of R and d satisfies all foreign key constraints in F and the cardinality bounds in λ. Let $\mathit{inst}(\mathbf{S})$ be the set of all databases of $\mathbf{S}$.

The definition omits conditions for satisfaction of foreign keys and cardinality bounds: the former can be found in, e.g. [11], and the latter means that each value for a key must occur in the referencing relation for the specified number of times.

Example 2.1.1 Kingfore Corporation (KFC) [14] in Beijing is a company that repairs the heating equipment in some residential area.

Fig. 2.2 shows (a part of) the database schema used by KFC, where keys are underlined, foreign keys italicized with references as arrows. The database includes six relation schemas: `tUser` (information of customer and staff including repair-persons) with key

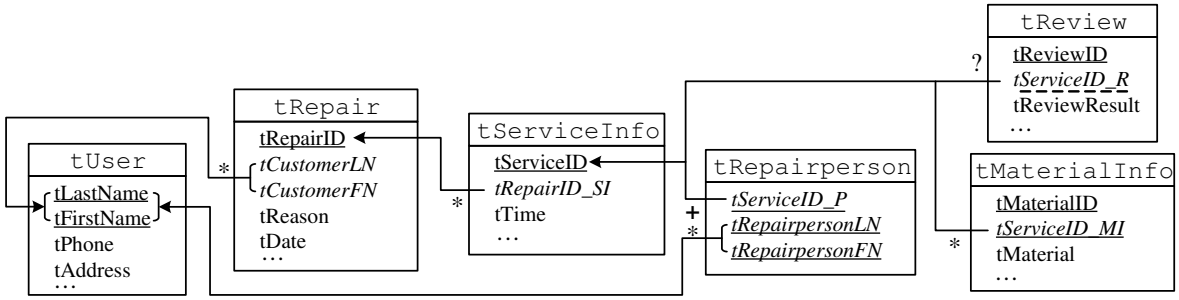

Figure 2.2: A Part of the Database Schema for KFC

(*tLastName*, *tFirstName*), `tRepair` (customer repair requests) with key *tRepairID* and a foreign key (*tCustomerLN*, *tCustomerFN*) to `tUser` (the requesting customer), `tServiceInfo` (individual on-site repair services performed by repair-persons), `tRepairperson` (the assigned repair-persons in a single service), `tReview` (service reviews) with two keys *tReviewID* and *tServiceID_R*, and `tMaterialInfo` (replacement parts used in a service) whose foreign key *tServiceID_MI* is also in a key. ∎

In the remainder of this thesis, for relation or database schema/instances, we may drop the term "keys" or "foreign keys" to refer to the concept of "database (schema) with keys" or "database (schema) with foreign keys" if the context is clear.

2.2 Artifacts and Business Entities

Artifact-centric models [2, 15, 16] specify a process with a data model (called "business entity") and a lifecycle. In the remainder of the section, we formulate the key concepts of the business entities. Since most of the technical results in this thesis focus on the business entities only, we may not introduce the lifecycle in this section. Essentially, An artifact model can have different types of lifecycle, which will be introduced separately when they are needed in the remainder of this thesis. Fig. 2.3 presents a high-level view of the important concepts that will be used in this thesis.

Concepts	Meaning
complex attribute	a hierarchical data attribute with sets
business entity	a complex attribute with key, local key, and dependency
artifact	a business entity with lifecycle
business entity/artifact enactment	the status/assignment of a business entity/artifact case at a time instant
business entity/artifact instance	a finite execution of a business entity/artifact case, i.e., a sequence of enactments

Figure 2.3: High-level description of artifact models

Unlike a database that is for storing all the data in an organization, a business entity (instance) only focuses on all the needed data for a process case or instance. Naturally, a business entity only contains small portion of what has been stored in a database and could be in different format, e.g., XML or relational models. This section we introduce a business entity that is hierarchically structured.

The following formalizes hierarchical structures for business entities.

Definition: The family of (*complex*) *attributes* is recursively defined as follows.

- each primitive attribute is an attribute,
- "$a\colon (a_1, ..., a_n)$" is a *tuple* attribute and "$a\colon \{(a_1, ..., a_n)\}$" is a *set* attribute, if $n \geqslant 1$, a_i's are attributes, and a is a name not occurring in any of the a_i's.

Let $\textsc{Att}(a)$ denote the set of all attributes used in a (including the name for a), and $\textsc{Pm}(a)$ the set of primitive attributes in a. A *value* of an attribute a is defined as follows.

- Each element in **Dom** is a value of a primitive attribute a,
- $(a_1\colon v_1, ..., a_n\colon v_n)$ is a value of a tuple attribute $a\colon (a_1, ..., a_n)$ if each v_i is a value of a_i,
- Each finite (possibly empty) set $\{(a_1\colon v_{1,1}..., a_n\colon v_{1,n}), ..., (a_1\colon v_{k,1}..., a_n\colon v_{k,n})\}$ is a value of a set attribute $a\colon \{(a_1, ..., a_n)\}$ if $k \in \mathbb{N}$ and for each $i \in [1..k]$ and each $j \in [1..n]$, $v_{i,j}$ is a value of a_j.

Given a tuple attribute "$a\colon(a_1, ..., a_n)$" or a set attribute "$a\colon\{(a_1, ..., a_n)\}$", for each $i, j \in [1..n]$, a_i is a *child* (*attribute*) of a and a *sibling* (*attribute*) of a_j, a is the *parent* (*attribute*) of a_i. Let a be an attribute. A set κ of attributes in $\textsc{Att}(a)$ is a *key* (in a) if attributes in κ are pairwise siblings of each other. In our model, key values must be unique among all artifact "enactments" in a "snapshot". A *local key* (in a) is a pair (κ, b) where κ is a key in a, and b an attribute in a and an ancestor of every attribute in κ. Intuitively, b provides the "context" within which a local key value is unique.

Given a set V of values for a complex attribute a, a key κ_1 in a, a local key (κ_2, b) in a, V *satisfies* the key κ_1 if there are no undefined values for attributes in κ_1 and each value of κ_1 attributes occurs at most once (i.e., is unique in V), V *satisfies* the local key (κ_2, b) if there are no undefined values for attributes in κ_2 and each value of κ_2 attributes occurs at most once within each value of b (but may occur multiple times in different values of b in V) and the b value is defined.

In order to define the notions of business entities, we introduce "functional paths" (for retrieving an attribute from another attribute).

Given a complex attribute a, a *functional* (or *fun-*) *path* p (from a_1 to a_n) is an expression of form "$a_1.a_2.\cdots.a_n$", where $n>0$, $a_i \in \textsc{Att}(\mathsf{a})$ for each $i \in [1..n]$, and for each $i \in [1..(n-1)]$, a_{i+1} is a sibling, parent of a_i, or a child of a_i if a_i is not a set attribute. Also, a_1 and a_n are the *head* and *tail* of p, resp. Note that if $a_1.\cdots.a_n$ is a fun-path, so is $a_i.\cdots.a_j$ for all $1 \leqslant i \leqslant j \leqslant n$.

Intuitively, a functional path from attribute a to b denotes that given an attribute-value pair of a, a unique value of b can be determined. In general, not every pair of attributes have a functional path as they may go through set attributes, where multiple values of b can be obtained given a.

A "business entity" is a complex attribute with (local) keys and "access dependencies". The latter specifies a partial order on all primitive attributes in the complex

attribute to denote that an attribute can be read/written after other attributes have been written. Although access dependencies could be divided into "write-write" and "write-read" dependencies, we combine them since they do not affect the technical results.

Definition: A *business entity* is a tuple $(\Omega, \mathsf{a}, K, L, dep)$, where

- Ω is a (unique) name,
- a is a (complex) attribute with name Ω and a contains the primitive attribute "ID" as its child,
- K is a set of *keys* in a and L a set of *local keys* in a such that (1) $\{\text{ID}\} \in K$, (2) each (local) key in $(K \cup L)$ other than $\{\text{ID}\}$ has a set attribute as its parent, and (3) each set attribute in a has exactly one element in $K \cup L$ as its child(ren), and
- $dep \subseteq \text{PM}(\mathsf{a}) \times \text{PM}(\mathsf{a})$ is a set of *access dependencies* such that the graph induced by dep is a directed acyclic graph rooted at ID and for each edge (u, v) in this graph, there exists a "functional path" from v to u.

When it is clear from the context, we may conveniently denote $(\Omega, \mathsf{a}, K, L, dep)$ as $\Omega(\mathsf{a}, K, L, dep)$, or simply Ω.

Example 2.2.1 Continue with Example 2.1.1; Fig. 2.4 shows the business entity of a repair business process (i.e., a case for a repair request). Each repair has an ID (renamed to *aID* to avoid confusion), repair information (*aRepair_Info*), and several services (*aService_Info*). Each service may require multiple replacement parts (*aReplacement_Parts*), 0 or 1 review (*aReview_Info*). Set attributes (e.g., *aService_Info* and *aReplacement_Parts*) are attached with a circled-star. There are three keys, *aID*, {*aRP_Last_Name*, *aRP_First_Name*} and *aServiceID* (indicated by **UNIQUE**) and a local key *aPartID* within

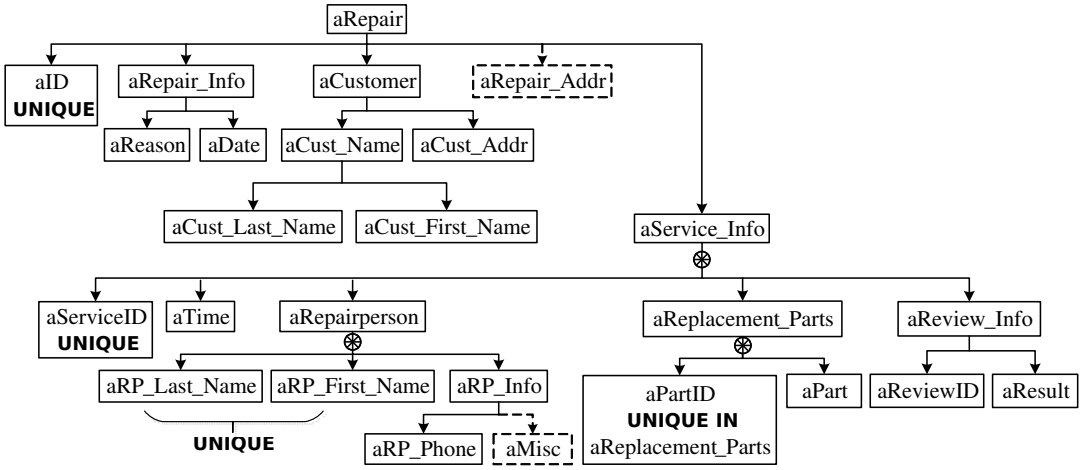

Figure 2.4: A business entity for the repair process (artifact)

the context of *aReplacement_Parts* (**UNIQUE IN**). Primitive attributes include *aID*, *aRepair_Info*, *aCust_Name*, *aServiceID*, etc. An access dependency example could be that before writing *aCust_Name*, *aID* should be written, or writing *aCust_Name* precedes reading *aCust_Addr*. An example fun-path can be "*aRP_Last_Name*.*aReprairperson*.*aService_Info*.*aID*", which means that given the last name of a repair person in a business entity instance, their is only one *aID* we can obtain. On the other hand, given an *aID*, there could possibly have multiple last names (of *aReprairperson*). ∎

Definition: An *enactment* of a business entity $\Omega(\mathsf{a}, K, L, dep)$ is a value of a that satisfies all local keys in L. An *instance* of Ω is a finite sequence (i.e., ordered bag) of enactments of Ω that satisfies each key in K. We denote by $Ent(\Omega)$ (or $inst(\Omega)$) the set of all enactments (resp. instances) of Ω.

Remark 2.2.2 In the remainder of this thesis, we use term "enactment" to refer to the status of an artifact or business entity "case" at a single time stamp; while "instance" to denote a finite execution of an artifact or business entity "case" (i.e., a sequence of enactments).

Chapter 3

Data Mapping for Artifacts

An important omission in current development practice for business process management systems is *modeling of data & access* for a business process, including relationship of the process data and the persistent data in the underlying enterprise database(s). This chapter develops and studies a new approach to modeling data for business processes: representing data used by a process as a hierarchically structured *business entity* with (*i*) *keys*, *local keys*, and *update constraints*, and (*ii*) a set of *data mapping rules* defining exact correspondence between entity data values and values in the enterprise database. This chapter makes the following technical contributions: (1) A data mapping language is formulated based on path expressions, and shown to coincide with a subclass of the schema mapping language Clio. (2) Two new notions are formulated: *Updatability* allows each update on a business entity (or database) to be translated to updates on the database (or resp. business entity), a fundamental requirement for process implementation. *Isolation* reflects that updates by one process execution do not alter data used by another running process. The property provides an important clue in process design. (3) Decision algorithms for updatability and isolation are presented, and they can be easily adapted for data mappings expressed in the subclass of Clio.

The remainder of the chapter is organized as follows. Section 3.1 introduces and motivates the framework and technical problems using a real example. Section 3.2 introduces the mapping language, and establishes the equivalence of the language and a subclass of Clio. Section 3.3 formulates and studies the updatability concept. Section 3.4 focuses on the isolation property. Summary is provided in Section 3.5.

3.1 Need for a Mapping Language

Two key components in modern enterprise systems are data management and business process management (BPM). Data refer to the *persistent* data managed by DBMS. Business processes (BPs) prescribe how *business operations* should be conducted. BPs consume, manipulate, and generate data; their interoperation is often accomplished through sharing data access. Current data and BP modeling approaches (e.g., ER [13], BPMN [1]) leave *associations* of persistent data in databases and data in BPs to the implementation level with little abstraction. Implementing business logic involves data access from and to database and often demands high development efforts.

Recognizing the importance of integrating data with process, the BPM community is embracing a shift from traditional activity-centric BP models (e.g. [17]) to data-centric modeling. *Artifact* models [2, 12] lead this trend by using an information model for data in a BP and a lifecycle model to capture how the business data evolve through business operations in the BP. In object-centric models [18, 19], process logic was modeled as object behaviors and object coordination. However, the modeling and design of the connection between databases and BPs is still *missing*. The challenge in system support for "linking" database and BP has two aspects: **1.** A formal approach that models the business process behavior and its associated data, captures its running status relevant to database updates, and maintains its connection with the database at all time. **2.**

Modeling and tool support is needed for automation and to ensure that every BP runs on its own data and maintains data consistency with the database.

To address the challenges, we study an *entity-database mappings* to manage data accesses and updates between database and BPs. The approach allows us to achieve:

- Model and manage BPs and database separately: a BP only accesses data in its entity, the database is connected the BP through the entity rather than directly (without abstraction).
- BP data and the database are conceptually separated but remain "connected" (via ***mapping rules***) and consistent (through the ***updatability*** property).
- ***Isolation*** property reflects the situation that data updates in one BP will not affect another BP execution in relation to the propagated database change.

This chapter makes the following technical contributions: (1) based on formal models for database and artifact (BP) design, a language is developed for specifying mapping rules between database and artifacts. It is shown that the language coincides with a subclass of Clio [6]. (2) Two new notions are formulated. "Updatability" allows each update on a business entity (or database) to be translated to updates on the database (or resp. business entity), a fundamental requirement for (business) process implementation. "Isolation" requires that updates by one process execution does not alter data used by another running process. The property provides an important clue in process design. (3) Decision algorithms for updatability and isolation are presented, which can be adapted for data mappings in the subclass of Clio.

In the remainder of this section, we describe an application development that demands a framework to model data accessed by a BP and automate database accesses by the BP through specified "mappings" between the BP data and the underlying database. Since the database is modified by BP executions indirectly through the data mappings, two

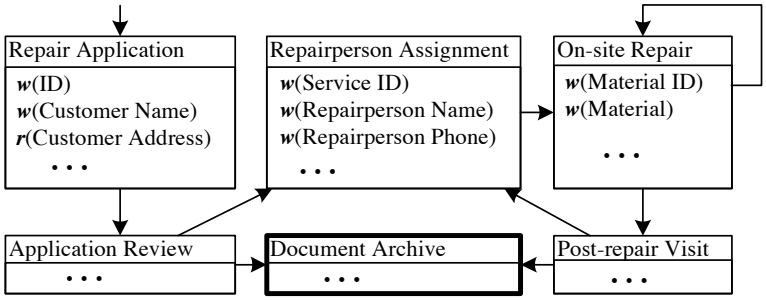

Figure 3.1: A Business Process for RBPS

technical problems arise: (1) "updatability", the ability to translate data modifications by BPs to database updates, and (2) "isolation", a property that data modifications by one process execution do not *implicitly change* the data used by another.

Kingfore Corporation (KFC) [14], mentioned slightly in Chapter 2) in Beijing was developing a repair management system (RMS) to manage their business of heating equipment repairs. Fig. 3.1 shows the (simplified) primary process with six activities (shown as boxes); inside each box are attributes to be ***r***ead or ***w***ritten by the activity. Actual values of all attributes are stored in a single database.

The process is initiated by a repair request from a customer via the RMS web front; a KFC operator is notified and approves the application if an on-site repair is needed. The application is then sent to the corresponding heating center manager for the customer's residential location. The manager reviews the request, and assigns one or more service-persons, a.k.a. repairpersons for the case. After the repair is completed (with one or more site visits) a representative from KFC visits the customer and closes the case. However, if the service-persons are unable to fix the problem or the post-repair visit receives an unsatisfactory response, the request is sent back to the manager and new service-persons will be assigned.

RMS was developed in jBPM (`www.jboss.org/jbpm`) with about 18,900 lines of hand-written code (for backend business logic, interface, and database access). In addition

to error-prone development, **a key weakness is that every small change on the process requires fairly significant code rewriting**. For example, when two Service Form Generation activities were deleted from the initial process, about 800 lines (or 4%) of the code were rewritten. A close examination shows that much of the rewriting can be avoided: If we have a *conceptual data model* for each process (schema) and the *mapping* between the process data schema and the database, the code lines concerning database accesses can be automatically generated. Indeed, the RMS code base contains more than $3,800$ or 20% lines for accessing the database that could be automatically generated.

Artifact-centric process modeling approach [12] (also [2, 15]) represents process data as a "business entity" and a process schema as a business entity with a "lifecycle". Fig. 3.1 shows a lifecycle as a finite state machine [16]. Fig. 2.4 shows part of the business entity for the repair process in Fig. 3.1. Each repair has an ID (renamed to *aID* to avoid confusion), several services (*aService_Info*), and a customer (*aCustomer*). Each service may require multiple replacement parts (*aReplacement_Parts*) and repairpersons (*aRepairperson*).

Once business entities are modeled, associations of business entities and the database can be defined as "data mappings". With such mappings between business entities and the database, the process designer only needs to focus on business entities in process modeling/design, while ignoring the database. (In general, the database may contain much more data than what is needed by a single process.)

Consider as an example where the *aID* value of each instance of the business entity (Fig. 2.4) for the repair process is mapped to the *tRepairID* value of some tuple in the table `tRepair` (Fig. 2.2). Then, the *aCust_LN* and *aCust_FN* values in the business entity instance with $aID = 101$ should correspond to *tCustomerLN* and *tCustomerFN* values (resp.) of the tuple with matching *tRepairID* (i.e., $= 101$). This can be specified using a path expression for *aCust_LN*: "*aCust_LN.aCust-omer.aID*@`tRepair`(*tRepairID*).*tCustomerLN*", where the first half of the expression navigates in the business entity (Fig. 2.4) and the

second half in the database (Fig. 2.2). More specifically, from the attribute-value pair *aCust_LN* in a business entity instance, we are able to uniquely locate the attribute-value pair *aCustomer*, and consequently uniquely locate the attribute-value pair *aID*. With the value of *aID*, we can match the value of *tRepairID* in table *tRepair* and then find the value of *tCustomerLN* (in the corresponding tuple). The above expression means that the given value of *aCust_LN* should be identical to the value of *tCustomerLN*. Furthermore, the values of *aCust_LN* and *aCust_FN* in the business entity instance with $aID = 101$ can be fetched by an SQL query:

```
SELECT tCustomerLN, tCustomerFN FROM tRepair WHERE tRepairID=101
```

This example hints that one can generate SQL expressions for database accesses from the data mappings. A further advantage of the mappings is that modification of a process (or database) can be made locally on the process (resp. database) first and then the mappings are adjusted. The current practice is to consider both at the same time, a more complex task.

During the RMS development, a problem encountered was that some updates on a business entity could not be translated into database updates. Consider the following mapping expressed using Clio [6] from the database (Fig. 2.2) as the source to the business entity (Fig. 2.4) as the target.

$$\begin{aligned} \forall i_s\ i_m\ n_l\ n_f\ m\ t\ \ &\text{tServiceInfo}(i_s, 101, t), \\ &\text{tRepairperson}(i_s, n_l, n_f), \text{tMaterialInfo}(i_m, i_s, m) \\ &\rightarrow \exists R\, p\ \text{aRepairperson}(n_l, n_f, R), R(p, m) \end{aligned} \qquad (*)$$

Rule $(*)$ creates an *aRepair* instance. Each entry in *aRepairperson* records for each service the replacement parts used by a service-person (using a join of `tMaterialInfo` and

`tRepairperson` on the service ID). Suppose there are two tuples (a_1, b, c_1) and (a_2, b, c_2) in `tMaterialInfo`, and two tuples (b, d_1, e_1) and (b, d_2, e_2) in `tRepairperson`. Then *aRepairperson* should have four entries $(d_1, e_1, (p_1, c_1))$, $(d_1, e_1, (p_2, c_2))$, $(d_2, e_2, (p_3, c_1))$, and $(d_2, e_2, (p_4, c_2))$, where p_1, p_2, p_3, are p_4 are some phone number values. If the process execution is to delete the entry $(d_2, e_2, (p_3, c_1))$ in *aRepairperson*, it is not possible to update the database so that the mapping rule $(*)$ would give the updated *aRepairperson*. The reason that rule $(*)$ is not "updatable" is due to poor design, which makes a cross product of tables `tMaterialInfo` and `tRepairperson`, and then stores the result in the same tuple *aRepariperson*. A solution to this problem is to move the attribute *aMisc* (a dashed box in Fig. 2.4) that stores the information of replacement parts to another attribute *aPart* under the tuple *aReplacement_Parts*. For the remainder of this chapter, we consider the attribute *aMisc* removed.

Another situation that can lead to an inconsistent update is that two attributes in the same business entity (instance) are mapped to the value of the same tuple in the database.

Example 3.1.1 Consider an instance of a business entity in Fig. 2.4, where both attributes *aRepair_Addr* and *aCust_Addr* are mapped to the attribute *tAddress* in the same tuple in table `tUser` (Fig. 2.2). When the value of *aRepair_Addr* is updated to the one that is different from the value of *aCust_Addr*, the database is unable to capture this change. The problem lies in the redundancy of information stored in the business entity. A quick fix is to remove a redundant attribute *aRepair_Addr*. Similarly, for the remainder of this chapter, we also consider attribute *aRepair_Addr* removed. ∎

Since the data in a process are mapped into the database, the above discussion suggests a critical "updatability" property that every business entity modification by a

process should always be translated into database updates. In this chapter, we argue that each specified mapping should be "updatable".

Another interesting property is independence between two process executions, called "isolation", i.e., the two process instances will not update the same attribute of the same tuple in the database. Consider the entity in Fig. 2.4. Suppose there are two running repair process instances requested by the same customer; if *aCust_Addr* is updated by one of the instances, the other repair process instance for the same customer will "see" the changed address even though it makes no updates on the address. In this situation, the artifact (in Fig. 2.4) is not "isolated" (with itself). For this example, it is desirable to have the address change made by one process to be immediately visible by the other running instance. However, Such implicit changes are not always helpful.

For example, an address change in the repair process of a customer will alter the address in an ongoing "Customer Profile Update" instance by the same customer. This failure to isolate is counter-intuitive and not desirable. To avoid this situation, one can restrict attribute *aCust_Addr* in Fig. 2.4 to be "read-only" for the repair process.

Isolation property is important to BP designers. Unlike updatability, we do not require artifacts to be always "isolated".

3.2 Entity-Data Mapping Rules

Process data are typically stored in a database (Fig. 3.2). When a process instance updates its business entity, the corresponding updates on the database should be performed. This section introduces "Entity-Data" mapping rules, or ED rules, a rule-based data mapping language for specifying correspondence between BP enactments and databases, and shows that ED rules are equivalent to a subset of Clio with respect to the expressiveness of mapping databases to enactments.

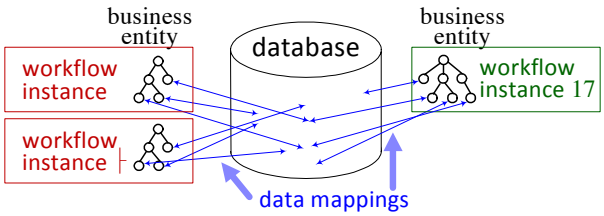

Figure 3.2: Process execution and database

A business entity can be naturally seen as a "view" on the database, when treating data mapping as queries. In this case, finding database updates for a process update resembles the view update problem studied in the relational databases [3, 4]. There are a few important differences.

First, the view update problem often has no solutions [3], and is hard in restricted cases when solutions exist, the presence of key and foreign key constraints further complicates the problem. However, a process instance acts on one entity instance at a time; even without data modeling as business entities, appropriate database updates are always found during process implementation. It suggests that business entities are more restrictive than views in practice and its update problem is generally solvable.

Second, business entities are hierarchically structured but not relational. Most view mechanisms are not suitable, with an exception of schema language languages such as Clio [6]. In this case, the database is the source and the entity is the target. However, the (view or) target update problem has not been studied. Our path expression based language is restrictive as a query language but more natural for specifying data mappings on process data. More importantly, the language facilitates solutions to the problem of propagating process updates to the database. In §3.2.1, we identify a subclass of Clio rules that are equivalent to ED rules.

To define the mapping language, we introduce the notions of "functional multi-paths" (retrieving attributes from another attribute in a business entity), "reference paths" (re-

trieving attributes from another attribute in a database), "cross-reference paths" (retrieving attributes in a database from an attribute in a business entity), and "key-mapping rules" (matching a (local) key in a business entity with (part of) a key in a database).

For each $m > 0$, an *m-ary functional multi-*(or *fun-m*)*path* is an expression of form "$p_0.[p_1, ..., p_m]$", where (1) for each $i \in [1..m]$, $p_0.p_i$ (concatenation p_0 and p_i) is a fun-path and the tail of p_i is a primitive attribute, (2) the head of p_0 is primitive, and (3) for each $i \in [1..m]$, there is a path from the tail of p_i to the head of p_0 in graph ($\textsc{Pm}(\mathsf{a})$, *dep*) (i.e., complying with the access dependencies to prevent that a referenced attribute is undefined). When $m = 1$, we may drop "[" "]" for convenience.

Example 3.2.1 In Fig. 2.4, an example fun-mpath could be "*aPart*.[*aPartID*, *aReplacement_Parts*.*aServiceID*]". ∎

Similarly, in the database side, a "reference paths" traverses through a chain of foreign key references to retrieve a single non-primary attribute. Given a database schema $(\mathbb{R}, F, \lambda)$ and $n > 0$, a *reference* (or *ref-*)*path* is an expression of form "$R_1(\kappa'_1).\kappa_1@R_2(\kappa'_2).\kappa_2@...@R_{n-1}(\kappa'_{n-1}).\kappa_{n-1}@R_n(\kappa'_n).a$", where for each $i \in [1..(n-1)]$, $\kappa_i \subseteq \textsc{Att}(R_i)$ and $(\kappa_i, \kappa'_{i+1}) \in F$ (the value of κ_i should be the same as κ'_{i+1}), and $a \in \textsc{Att}(R_n)$.

Example 3.2.2 In Fig. 2.2, a reference path is

`tMaterial-Info`(*tMaterialID*, *tServiceID_MI*).*tServiceID_MI*
@`tServiceInfo`(*tServiceID*).*tRepairID_SI*
@`tRepair`(*tRepairID*).*tReason*

that denotes the *tReason* value of the tuple in `tRepair` corresponding to a unique tuple in `tMaterialInfo` by matching foreign keys. ∎

A "cross-reference path" defined below concatenates a fun-mpath and a ref-path to set

up a reference from a non-key (i.e., neither a key or a local key) business entity attribute to a database attribute. Given Ω as a business entity and $\mathbf{S}$ as a database schema a *cross-reference* (or *cref-*)*path* is an expression of form "$p_0.[p_1, ..., p_n]@R_1(\kappa'_1).\kappa_1@...@R_m(\kappa'_m).a$", where $p_0.[p_1, ..., p_n]$ is a fun-mpath of Ω and $R_1(\kappa'_1).\kappa_1@...@R_m(\kappa'_m).a$ is a ref-path of $\mathbf{S}$, $n = |\kappa'_1|$, and the head of p_0 is not a part of any (local) key.

A cref-path $p_0.[p_1, ..., p_n]@R_1(\kappa'_1).\kappa_1@...@R_m(\kappa'_m).a$ is used to establish an "equality" between the head of p_0 in a business entity and attribute a in a database schema. The linkage is established by matching the last (ordered set of) attribute(s) in the fun-mpath and the key of the first relation schema in the ref-path.

Example 3.2.3 For the database schema in Fig. 2.2 and the business entity in Fig. 2.4, a cref-path can be '*aReason*.*aRepair_Info*.*aID*@`tRepair`(*tRepairID*).*tReason*' denoting that the *tReason* attribute can be retrieved from the *aReason* attribute by matching the values of *aID* and *tRepairID*. A more complicated example is

aPart.[*aPartID*, *aReplacement_Parts*.*aServiceID*]
@`tMaterial-Info`(*tMaterialID*, *tServiceID_MI*).*tMaterial*

whose linkage is matched by a pair of attributes (i.e., *aPartID* to *tMaterialID*, and *aServiceID* to *tServiceID_MI*). ∎

In our framework, a cref-path establishes the relationship between a non-key attribute in a business entity and an attribute in a database. For attributes in a key or local key, a "key-mapping rule" is used, which establishes the relationship between a key (or local key) in a business entity and a key (resp. part of a key) in a database.

Let $\mathbf{S} = (\mathbb{R}, F, \lambda)$ be a database schema and $\Omega(\mathtt{a}, K, L, dep)$ a business entity. Further suppose that γ is a (local) key of Ω, $R \in \mathbb{R}$ a relation schema, and $\kappa \subset \textsc{Att}(R)$ where $|\gamma| = |\kappa|$. A *key-mapping rule* of γ is an expression of form "$R.\kappa$" or "$R.\kappa$ **WHEN** φ"

such that

- if γ is the ID of Ω, then κ is a key of R and the key-mapping rule must have the form "$R.\kappa$",
- if γ is not the ID of Ω, "$R.\kappa$ **WHEN** φ" must be used; moreover, κ is a key of R if γ is a key,
- if γ is a local key in Ω, κ is contained in a key of R and specifically $\kappa = \kappa' - \{f \mid \exists \kappa'' \in \textsc{Keys}(\mathbb{R}), (f, \kappa'') \in F\}$ where κ' is a key of R,
- φ is an expressions with form $p_0.[p_1, ..., p_n] = R.\kappa'$, where $p_0.[p_1, ..., p_n]$ is a fun-mpath, the head of p_0 is the first element of γ to denote that the all the referenced attributes (based on $p_0.[p_1, ..., p_n]$) should start from γ, and κ' is a foreign key of R; the tails of $p_1, ..., p_n$ (i.e., the referenced attributes from γ) form the (local) key in the "upper level": if γ is the (local) key for a set attribute a, and b is an ancestor of a in the business entity and a set attribute (or the root) with no set attribute between a and b, then the tails of $p_1, ..., p_n$ form the (local) key of b, and
- for each $p_0.[p_1, ..., p_n] = R.\kappa'$ in φ, if $\lambda(\kappa')$ is "?" or "1", then for each $i \in [0..n]$, p_i contains no set attributes; otherwise (i.e., $\lambda(\kappa')$ is "$*$" or "+"), p_i contains a set attribute for some $i \in [0..n]$.

Intuitively, a key-mapping rule defines an equality between a (local) key γ (in a business entity) and (part of) a key κ (in a database). The "**WHEN**" conditions are needed for the situations when γ equals to κ under a context (presence of a foreign key) either not in a nested set (a key in the business entity) or within a nested set (a local ley).

Example 3.2.4 Consider the database schema in Fig. 2.2 and the business entity in Fig. 2.4, the key-mapping rule `tRepair`.*tRepairID* for *aID* denotes that there exists a tuple

of `tRepair` whose value of *tRepairID* is equivalent to the value of *aID* in a business entity. Moreover, the key-mapping rule for *aServiceID* could be "`tServiceInfo`.*tServiceID* **WHEN** *aServiceID.aService_Info.aID* = `tServiceInfo`.*tRepairID_SI*" to denote that there should exist a tuple of `tServiceInfo` whose *tServiceID* value is equal to the value of *aServiceID* in a business entity only under the circumstance where in the same business entity and the tuple, the value of *aID* is the same as the value of *tRepairID_SI*. ∎

Definition: Given a business entity $\Omega(\mathtt{a}, K, L, dep)$, a database schema $\mathbf{S} = (\mathbb{R}, F, \lambda)$, if γ is a (local) key, or a primitive attribute not in a (local) key in Ω, an *entity-data (ED) mapping rule* of γ is either a key-mapping rule if γ is a key or local key, otherwise an expression "$= p$" where p is a cref-path whose head is γ.

An ED rule for a primitive attribute or (local) key a may have different forms. If a is not a key or a local key, then a cref-path can uniquely match the value of a in a business entity to the value of an attribute in a database (these two values should always be the same). Otherwise, if a is a (local) key, then two cases can be obtained: (1) if a is the ID, then there should exist a key κ in the corresponding relation schema R, such that the value of a in a business entity is the same as the value of κ in some instance of R. (2) otherwise, the value of a should be "scoped" by the "**WHEN**" condition.

Example 3.2.5 Fig. 3.3 shows the mapping rules for some attributes based on Figs. 2.2 and 2.4 (not including the attributes in dashed boxes). The meaning of the mapping rules for *aID* and *aServiceID* is the same as explained in Example 3.2.4. And, comparing with Example 3.2.3, the mapping rule for *aReason* means that the value of *aReason* is the same as the value of *tReason* in a tuple of `tRepair` whose key (*tRepairID*) has the value of *aID*. ∎

Attributes	Mapping rules
aID	tRepair.tRepairID
aReason	= aReason.aRepair_Info.aID @tRepair(tRepairID).tReason
aDate	= aDate.aRepair_Info.aID @tRepair(tRepairID).tDate
aCust_Last_Name	= aCust_Last_Name.aCust_Name.aCustomer.aID @tRepair(tRepairID).tCustomerLN
aCust_First_Name	= aCust_First_Name.aCust_Name.aCustomer.aID @tRepair(tRepairID).tCustomerFN
aCust_Addr	= aCust_Addr.[aCust_Last_Name,aCust_First_Name] @tUser(tLastName, tFirstName).tAddress
aServiceID	tServiceInfo.tServiceID **WHEN** aServiceID.aService_Info.aID = tServiceInfo.tRepairID_SI
aTime	= aTime.aServiceID@tServiceInfo(tServiceID).tTime
(aRP_Last_Name, aRP_First_Name)	tRepairperson.[tRepairpersonLN, tRepairpersonFN] **WHEN** aRP_Last_Name.aRepairperson.aServiceID = tRepairperson.tServiceID_P
aRP_Phone	= aRP_Phone.aRP_Info.[aRP_Last_Name, aRP_First_Name] @tUser(tLastName, tFirstName).tPhone
aPartID	tMaterialInfo.tMaterialID **WHEN** aPartID.aReplacement_Parts.aServiceID = tMaterialInfo.tServiceID_MI
aPart	= aPart.[aPartID, aReplacement_Parts.aServiceID] @tMaterialInfo(tMaterialID, tServiceID_MI).tMaterial
aReviewID	= aReviewID.aReview_Info.aServiceID @tReview(tServiceID_R).tReviewID
aResult	= aResult.aReviewID @tReview(tReviewID).tReviewResult

Figure 3.3: Entity-Data mapping rules

Given a business entity Ω and a database schema $\mathbf{S}$, a mapping rule r (with respect to Ω and $\mathbf{S}$) for a (set of) attribute(s) A in Ω is *satisfied* by an enactment $\sigma \in \mathit{Ent}(\Omega)$ and a database $d \in \mathit{inst}(\mathbf{S})$, denoted as $(\sigma, d) \models r$, if one of the following conditions is satisfied.

- r is of form "$R.\kappa$" and there exists a tuple τ of R in d, such that the value of κ in τ

is the same as the value of A in σ. (Notice that the correspondence between the ID A and the key κ is one-to-one.)

- r is of form "$R.\kappa$ **WHEN** φ", for each tuple τ of R in d that satisfies φ wrt σ, there exists a value for A identical to the value of κ in τ, and for each value v for A in σ, there exists a tuple τ of R in d such that the value of κ in τ is v and τ and σ satisfy φ.
- r is of form "$= p@R_1(\kappa'_1).\kappa_1@...@R_j(\kappa'_j).a$", where p is a functional multi-path, for each value of all tails of p in σ, there exists a value v for A in σ such that v is the value of κ'_j in a tuple of R_j that is "retrieved" according to the cref-path, and vice versa.

Definition: Given a business entity Ω and a database schema $\mathbf{S}$, a set of ED rules is an *ED cover* if it contains exactly one mapping rule for each (local) key or non-key primitive attribute of Ω.

The set of mapping rules in Fig. 3.3 is an ED cover for the business entity in Fig. 2.4 (ignoring the attributes in dashed boxes). The following property confirms that an ED cover yields a well defined data mapping.

Lemma 3.2.6 Let Ω be a business entity, $\mathbf{S}$ a database schema, M an ED cover. Then, for each database $d \in \mathit{inst}(\mathbf{S})$ and a value v_{ID}, there exists at most one enactment $\sigma \in \mathit{Ent}(\Omega)$ holding v_{ID} as its ID value such that for each $r \in M$, $(\sigma, d) \models r$.

Proof: (Sketch) Let Ω, $\mathbf{S}$, M, σ, d, and v_{ID} be as stated in the Lemma. As M is a tight cover of Ω, each primitive attribute of Ω is "covered" by exactly one mapping rule. Suppose that if there exists a distinct enactment σ' from σ of Ω, where σ' holds the same ID value v_{ID} as σ; and without loss of generality, assume that σ contains an attribute-value pair $(a : v_a)$, where a is a primitive attribute of Ω and $v_a \in \textsc{Dom}$, and σ' does not. Further, suppose that the only mapping rule for a is r. Then it can be shown that either

v_a is not a value for the corresponding attribute of a in d (which means that $(\sigma, d) \not\models r$) or v_a is a value for the corresponding attribute of a in d (which means that $(\sigma', d) \not\models r$), which leads to a contradiction. Notice that if the given ID does not have a corresponding value in d according to the mapping rules in M, then σ does not exist. ∎

3.2.1 Clio and an equivalence result

In this subsection, we view ED rules and Clio [6] as "queries" mapping database instances to enactments, define the notion of "equivalence" of such queries, and then formulate a syntactic subclass of Clio, called "entity maps". The main result shows that ED covers are equivalent to Clio entity maps.

We fix $\mathbf{S}$ to be a database schema, Ω a business entity, and x_{ID} an enactment ID (used as a variable), unless otherwise specified.

Given an enactment of Ω with ID x_{ID}, an ED cover M specifies all values in the database that correspond to values in the enactment x_{ID}. The rules can also be used to "fetch" the values for the enactment x_{ID} using the correspondence. For an instance $d \in \mathit{inst}(\mathbf{S})$, let $M(d)$ be the output enactment x_{ID}.

Clio is a schema mapping language using tgd-like (tuple generating dependency) rules to transform hierarchical source data to hierarchical target data. For our purpose, we consider only Clio rules that map relational databases to hierarchical data representing enactments. In order to compare with ED rules that produce one enactment, we allow the variable x_{ID} as the only free variable (holding the ID of the resulting enactment). We focus on constant-free Clio rules of the following form with only x_{ID} occurring free:

$$\forall \bar{x}\ \Phi_{\mathbf{S}} \rightarrow \exists \bar{Y}\ \Psi_{\Omega} \qquad (\dagger)$$

where $\bar{x}$ is a sequence of first-order variables, $\Phi_{\mathbf{S}}$ a conjunction of atomic formulas of form "$R(\bar{z})$" with R a relation in $\mathbf{S}$ and $\bar{z}$ variables in $\bar{x} \cup \{x_{\text{ID}}\}$, $\bar{Y}$ variables for (nested) tuple/set constructs in Ω (i.e., $\bar{Y}$ has no first-order variables), Ψ_Ω a conjunction of atomic formulas with Ω or variables in $\bar{Y}$ as relations each first-order variable occurs at most once, and x_{ID} occurs in both $\Phi_{\mathbf{S}}$ and Ψ_Ω. In addition, equality (two occurrences of the same variable) is only allowed between attributes for a foreign key constraint in $\mathbf{S}$.

Example 3.2.7 Equation $(*)$ is a Clio mapping from the relational database in Fig. 2.2 to the nested structure in Fig. 2.4. ∎

Given a set of Clio rules of form (i), we use the semantics similar to the one in [20] that produces a single enactment from a database (rather than the semantics in [7]. In particular, for nested sets, this semantics produces enactments that satisfy partition normal form (PNF) [21].

We introduce further syntactic restrictions on Clio rules. The overall goal is to have a set of Clio rules to map a database to a single business entity instance with the ID x_{ID}.

We construct a graph G_r for each rule r of form (i) whose nodes are occurrences of relations in $\Phi_{\mathbf{S}}$ and contains edges (R_1, R_2) if there is an equality in $\Phi_{\mathbf{S}}$ for a foreign key (a) from R_2 to R_1, or (b) R_1 to R_2 with a "1-1" constraint.

CONDITION 1. A rule r is *contributing* if (1) after collapsing each strongly connected component of G_r into a node, the resulting graph is a tree whose root is the relation containing the variable x_{ID}, and (2) each strongly connected component in G_r contains a node whose corresponding formula contains a variable occurring in Ψ_Ω of r.

Condition 1 requires that the left-hand side of a Clio rule is connected and join can only happen between keys-foreign keys, thus avoiding arbitrary cross products.

For each business entity Ω, we construct its *normalization* Ω^{norm} by recursively apply-

ing the following operations:

(1) Collapse two consecutive tuple constructs,

(2) If a set construct τ has a local key within a scope, we

duplicate in τ (attributes of) the key of the scope.

Clearly, each attribute in Ω^{norm} corresponds to one attribute in Ω and conversely, each attribute in Ω corresponds to one attribute in Ω^{norm}, except for key attributes in a scope of local key(s).

Note that Ω^{norm} has one tuple construct (the root) and 0 or more nested set constructs. For each tuple/set construct τ in Ω^{norm}, let $\text{KEY}(\tau)$ be the set of attributes in the key of τ.

Let $F_{\mathbf{S}}$ be the set of functional dependencies on $\mathbf{S}$ obtained by turning each key, foreign key dependency into a functional dependency and each foreign key with 1-1 constraint into two functional dependencies (both directions). For an atomic attribute a in Ω, let $r(a)$ be the set of attributes in $\mathbf{S}$ contributing values to a, $r(a)$=$\{b \mid b$ an attribute in $\mathbf{S}$ and there is a variable x corresponding to attribute a in Ψ_Ω and b in $\Phi_{\mathbf{S}}\}$. For a sequence $a_1...a_n$ of attributes in Ω, let $r(a_1...a_n)$ denote $\times_{i=1}^{n} r(a_i)$.

CONDITION 2. For each tuple/set construct τ in Ω^{norm}, the rule r is τ-*full* if (1) each attribute in $\text{KEY}(\tau)$ corresponds to an attribute (variable) of Ω occurring in Ψ_Ω, and (2) there is a sequence $B \in r(\text{KEY}(\tau))$ of attributes of $\mathbf{S}$ such that (i) there is an occurrence $R(\bar{z})$ of a relation R in $\Phi_{\mathbf{S}}$ such that B is a key of R and z contains variables corresponding to B, and (ii) there is a sequence of attributes $C \in r(\text{PM}(\tau))$ such that $F_{\mathbf{s}}$ logically implies the functional dependency $B \rightarrow C$ (recall that $\text{PM}(\tau)$ is the set of primitive attributes in τ).

A rule r is *full* if it is τ-full for each set/tuple construct τ in Ω^{norm} with an attribute occurring in Ψ_Ω.

Condition 2 concerns nested sets: in each nested set, attribute values of each tuple

must be uniquely identifiable with the keys in the nested set, avoiding multiple values for a tuple (violating key constraints).

CONDITION 3. A rule r is *consistent* if for each pair of constructs τ_1, τ_2 in Ω^{norm} where τ_1 is a parent of τ_2, (1) if τ_2 has an attribute occurring in Ψ_Ω, so does τ_1, and (2) there exist sequences of attributes $B_i \in r(\text{KEY}(\tau_i))$ $(i = 1, 2)$ such that (i) $F_{\mathbf{S}}$ implies the functional dependency $B_2 \rightarrow B_1$, and (ii) there is an occurrence $R(\bar{z})$ in $\Phi_{\mathbf{S}}$ where $\bar{z}$ contains both variables for B_1 and variables for B_2.

Condition 3 insists that each nested set should be connected to its corresponding context (i.e., the parent tuple).

CONDITION 4. We define $H(r) = \{\tau \mid \tau$ is a tuple/set construct in Ω^{norm} with at least one attribute occurring in $\Psi_\Omega\}$. We construct a tree T_Ω from Ω^{norm} as follows: tuple/set constructs as nodes and child relationships as edges. The rule r is *closed* if for each $\tau \in H(r)$, every ancestor of τ in T_Ω is also in $H(\tau)$.

Condition 4 concerns the right-hand side of a Clio rule: if a rule produces a value for some nested set in a business entity, all of its ancestors should be present (to provide the context).

Definition: A set π of Clio rules of form (i) is an *entity map* if (1) every rule in π is contributing, full, consistent, and closed, (2) for each path p in T_Ω from the root, there is a rule $r \in \pi$ such that $H(r) = \{\tau \mid \tau$ is on $p\}$, and (3) for each pair of rules $r_1, r_2 \in \pi$, $H(r_1) \subseteq H(r_2)$ implies the existence of a 1-1 embedding of r_1 into r_2.

In general, Clio rules may produce enactments with undefined (or null) values. To simplify our presentation, we focus only on databases and enactments with no undefined values.

Definition: Let $\mathbf{S}$ be a database schema, Ω a business entity, π a Clio entity map, and M a ED cover. Then, π and M are *equivalent*, denoted as $\pi \equiv M$, if for each database

$d \in inst\,(\mathbf{S})$, $\pi(d) = M(d)$ when $d, \pi(d), M(d)$ contain no undefined values.

Lemma 3.2.8 *Let $\boldsymbol{S}$ be a database schema without "0-1" constrained foreign keys and Ω a business entity. If π is a Clio entity map and M a ED cover, then for each $d \in inst\,(\boldsymbol{S})$ without undefined values, neither $\pi(d)$ nor $M(d)$ have undefined values.*

Theorem 3.2.9 *Let $\boldsymbol{S}$ be a database schema without "0-1" constrained foreign keys and Ω a business entity. For each Clio entity map π, there is an ED cover M such that $\pi \equiv M$. And the converse also holds.*

The proof for Theorem is illustrated in subsections 3.2.2 and 3.2.3.

3.2.2 ED Mapping Rules to Clio

In this and the next section, we show a forward and backward translation between a set of updatable ED mapping rules and a set of Clio entity map rules, s.t., given a business entity Ω, a database schema $\mathbf{S}$, (1) if a set of Clio entity map rules hold for some instances of Ω and $\mathbf{S}$ then the translated ED mapping rules hold for the same instances, and (2) if the ED mapping rules hold for some instances of Ω and $\mathbf{S}$ then the translated Clio entity map rules hold for the same instances.

The following "Forward Clio Translation Procedure" provides a high-level view of how to translate ED mapping rules to Clio. Each step will be explained informally with examples afterwards.

Forward Clio Translation Procedure

Input: a database schema $\mathbf{S}$, a business entity Ω, an ED cover M of $\mathbf{S}$ and Ω.
Output: a set of Clio entity map rules

A. Build a "dependency graph" G based on M and Ω.

B. Group the directly connected nodes of G from the same relation; and generate a "grouped graph".

C. For each tuple in Ω, compute a set of "associating predicates".

D. For each tuple in Ω, construct a "template" of Clio rule based on its "associating predicates".

E. "Propagate" the equalities among variables for each "template" of Clio rule.

In ***Step A***, a "dependency graph" is needed. A *dependency graph* is a graph (V, U, E), where V and U are two node sets and $E \subseteq (V \cup U) \times (V \cup U)$ is an edge set, which is constructed as follows:

1. Initially V, U, and E are empty.
2. For each (set of) attribute(s) that has mapping rules, create a node in V.
3. For each $u, v \in V$, if the mapping rule of v is of form "$R.\kappa$ **WHEN** φ" and u occurs as a tail in a fun-path in φ, then add edge (u, v) in E.
4. If a node $v \in V$ has mapping rule of form "$= p@R_1(\kappa'_1).\kappa_1@...@R_j(\kappa'_j).a$", then creates nodes κ_1^v, κ_2^v, ..., κ_{j-1}^v, and a^v in U; and for each node $u \in V$ that occurs as a tail in p create edges (u, κ_1^v), (κ_1^v, κ_2^v), ..., (κ_{j-1}^v, a^v), and (a, v) in E.

A dependency graph essentially provides the information that how an attribute is "determined" by another attribute.

Example 3.2.10 Based on Fig. 3.3, a dependency graph can be created, which is shown in Fig. 3.4 (for now, ignore the dashed boxes and the relation names in the parentheses; further, we ignore the subscripts for some nodes as the context is clear). One thing to notice is that as *aRP_Last_Name* and *aRP_First_Name* share the same mapping rule, they will be group into the same node (with grey background).

In Fig. 3.4, we can have an understanding that *aID* can "determine" *tCustomerLN* (in database) and *tCustomerLN* can "determine" *aCust_Last_Name* (in business entity). Also,

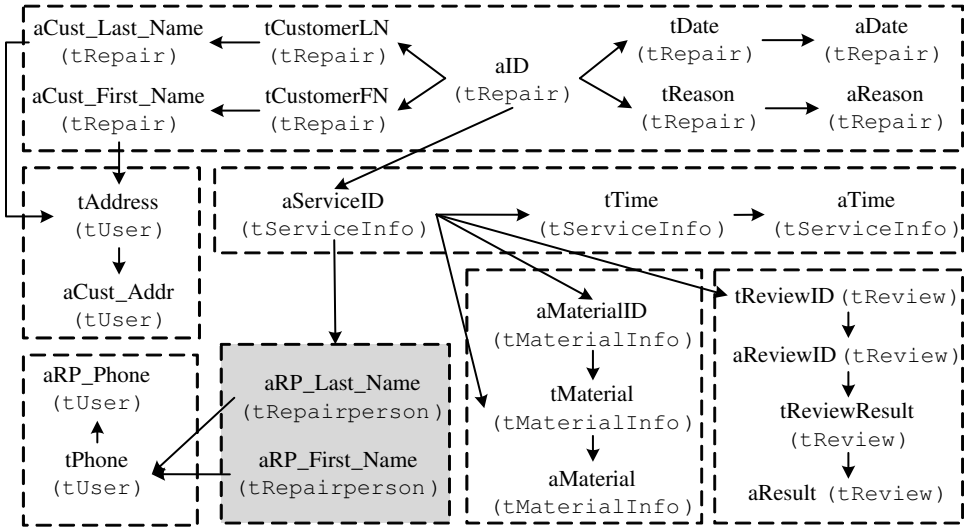

Figure 3.4: A dependency graph

aServiceID can "determine" a set of *aMaterialID*s and each *aMaterialID*, together with the corresponding *aServiceID*, can "determine" *tMaterial*, who can "determine" *aMaterial*. Notice that the dependency is purely based on the ED mapping rules. ∎

For ***Step B***, we need to label the corresponding relation of a node in a dependency graph in order to know what predicates in the database can "contribute to" what attributes in the business entity. The labeling is stated as follows:

1. Let (V, U, E) be the dependency graph.
2. For each $v \in V$ that has mapping rule of form "$R.\kappa$ (**WHEN** φ)", label v with R.
3. For each $v \in V$ that has mapping rule of form "$= p@R_1(\kappa'_1).\kappa_1@...@R_j(\kappa'_j).a$", label v with R_j, $a^v \in U$ with R_j, and each $\kappa^v_i \in U$ (where $i \in [1..(j-1)]$) with R_i.

After the labeling, if two nodes are directly connected and labeled with the same relation, we can group them together to denote that these two attributes are from the same predicate. The result collapsed graph is called the "*grouped graph*".

Example 3.2.11 Continuing with Example 3.2.11, the corresponding relation of a node is labeled within parentheses underneath each node. For example, as the mapping rule for *aID* is "`tRepair`.*tRepairID*", node *ID* corresponds to relation `tRepair`; and since the mapping rule for *aReason* is "= *aReason*.*aRepair_Info*.*aID*@`tRepair`(*tRepairID*).*tReason*", node *aReason* corresponds to `tRepair` as well. The dashed boxes denote the grouping result. ∎

Notice that given a dependency graph, the edges among the grouped nodes are acyclic due to the nature of the mapping rules; and the grouped graph is rooted by a single predicate (i.e., the predicate without incoming edges), called *root predicate*, which contains the ID of the target business entity. For example, the root predicate in Fig. 3.4 is "`tRepair`".

For ***Step C***, we need to identify a set of "associating predicates" for each tuple in the business entity. The intuition is that when constructing the Clio rules, each Clio rule contributes to the information of a single tuple in the business entity (whose common parent attribute could either be a set or not a set). Hence, in order to capture each related predicate in a database that may "contribute to" the formation of a business entity tuple, we need to understand what predicates to use.

The algorithm is informally described as follows:

1. For each tuple τ in the business entity, denote $\mathbf{P}(\tau)$ to be a set of nodes in the grouped graph, where for each $P \in \mathbf{P}(\tau)$, P should contain at least one attribute in τ.
2. If a node p in the grouped graph that is on a path from the root predicate to a node in $\mathbf{P}(\tau)$, then union $\{p\}$ to $\mathbf{P}(\tau)$; repeat this step until $\mathbf{P}(\tau)$ reaches a fixpoint.

Example 3.2.12 Continuing with Example 3.2.11, to specify the information in the tuple of *aRepairperson*, we need attributes *aRP_Last_Name*, *aRP_First_Name*, and *aRP_Phone*.

These three attributes are contained in nodes `tUser` and `tRepairperson` in the grouped graph in Fig. 3.4. Thus, the associating predicates for *aRepairperson* are `tUser` and `tRepairperson`, together with the predicates on the path from the root to these two nodes, i.e., `tRepair` and `tServiceInfo`. ∎

Step D is to build a Clio mapping "template" for each tuple in a business entity. In general, a Clio mapping is divided into two halves: to the left of "→" is the structure of the database with universal quantifiers and to the right of "→" is the structure of the business entity with existential quantifiers. In Step C, we have identified what predicates to put on the left; while for the right, what we need is a nested structure representing a "fragment" of the business entity. By "fragment" we mean that it is not necessary to present the entire business entity by nested relations, since each Clio mapping rule is for a single tuple only.

Example 3.2.13 If we are to construct the Clio rule for tuple *aRepairperson*, then on the left, predicates `tRepair`, `tServiceInfo`, `tRepairperson`, and `tUser` are needed according to Example 3.2.12; while on the right, only the "ancestor" tuples of *aRepairperson* are needed, i.e., *aRepair* and *aService_Info*. The following expression shows a Clio mapping template for *aRepairperson*:

$$\begin{aligned}
&\forall y_1, ..., y_{15}, \text{tRepair}(y_1, y_2, y_3, y_4, y_5), \text{tServiceInfo}(y_6, y_7, y_8),\\
&\qquad \text{tRepairperson}(y_9, y_{10}, y_{11}), \text{tUser}(y_{12}, y_{13}, y_{14}, y_{15}), \rightarrow\\
&\exists \text{RI, C, S, R, M, V},\\
&\qquad \text{aRepair}(x_1, \text{RI, C, S}), \text{S}(x_2, x_3, \text{R, M, V}), \text{R}(x_4, x_5, x_6)
\end{aligned}$$ ∎

Notice that based on Example 3.2.13, all the variables are distinct and all the variables to the right of "→" are not quantified in a "template"; as in Step E, the equalities will be set up among the variables to fix this issue.

The last ***Step E*** is to identify the equalities among variables for each Clio mapping template generated in the last step. In general, there are two types of equalities to be addressed. one is the equality between a variable in a database and a variable in a business entity, which is determined by ED mapping rules. And the other one is between two variables in a database (for matching foreign keys and keys), which is determined by dependency graphs.

Example 3.2.14 Continue with Example 3.2.13. The following provides the examples of how to establish equalities according to the two types.

Type 1 (database variables and business entity variables): As the mapping rule for "*aID*" is "`tRepair`.*tRepairID*", x_1 and y_1 are equal, which can be replaced by a common variable, say "id_r". Further, for attribute "*aTime*", since its mapping rule is "= *aTime.aServiceID*@`tServiceInfo`(*tServiceID*).*tTime*", x_3 (which represents *aTime*) and y_8 (which represents *tTime*) are equal.

Type 2 (database variables and database variables): For predicates tRepair(y_1, y_2, y_3, y_4, y_5) and tServiceInfo(y_6, y_7, y_8), since there is an edge from *aID* to *aServiceID*, which is contained in predicate `tServiceInfo`, y_1 is equal to the foreign key y_7 that is referencing *aID*.

After identifying all the equalities, we can obtain the final the expression based on the one in Example 3.2.13 (where in the following, id_r is a free variable; thus does not need to be quantified):

$$\forall ln_c, fn_c, r, d, id_s, id_r, t, ln_r, fn_r, p_r, a_r,$$
$$\text{tRepair}(id_r, ln_c, fn_c, r, d), \text{tServiceInfo}(id_s, id_r, t),$$
$$\text{tRepairperson}(id_s, ln_r, fn_r), \text{tUser}(ln_r, fn_r, p_r, a_r) \rightarrow$$
$$\exists \text{RI, C, S, R, M, V,}$$

$$\text{aRepair}(id_r, \text{RI, C, S}), \text{S}(id_s, t, \text{R, M, V}), \text{R}(ln_r, fn_r, p_r) \quad \blacksquare$$

For each tuple in the given business entity, a Clio mapping like the one in Example 3.2.14 is needed.

3.2.3 Clio to ED Mapping Rules

In the following "Backward Clio Translation Procedure", we show high-level description of the translation from a set of Clio entity mapping rules to ED mapping rules. Each step will be explained informally with examples afterwards.

Backward Clio Translation Procedure

Input: a database schema **S**, a business entity Ω,
a set of Clio entity map rules from **S** to Ω

Output: an set of ED mapping rules

A. Create the mapping rule of the ID of Ω.

B. Create the mapping rule for each (local) key but the ID.

C. Create the mapping rules for all other primitive attributes.

In ***Step A***, let r be a Clio mapping rule in the given set of Clio rules, *id* be the free variable for the ID attribute in the given business entity specified on the right side of r, and R be the predicate on the left side of r that contains *id* as its key and has no foreign key referencing to other predicates in r. Create an ED mapping rule "$R.\kappa$" for the ID of the given business entity, where κ is (the name of) the key for R.

Example 3.2.15 Consider the Clio mapping rule in Example 3.2.14. id_r is the free variable holding the value for the ID of *aRepair*. And id_r occurs in both predicates `tRepair` and `tServiceInfo`; while only in `tRepair`, does id_r occur as a key. Therefore, the ED mapping rule for the ID of *aRepair* is "`tRepair`.*tRepairID*". $\blacksquare$

Notice that since each Clio mapping rule given are isomorphic in terms of the "common" part, each Clio rule should generate the same mapping rule for the same attribute.

Step B is to identify the ED mapping rules for all the (local) keys but the ID. For each (local) key κ in the given business entity, let r be a given Clio mapping rule that (on the right side of r) holding the nested relation R_1, R_2, ..., R_n, where R_1 contains a variable id_1 representing κ, R_n is a set attribute or the root, for each $i \in [2..n]$, R_i is the parent attribute of R_{i-1}, and for each $i \in [2..(n-1)]$, R_i is not a set attribute. Let κ' be the (local) key of R_n and represented by variable id_2. Thus on the left side of r there should exist two predicates P_1 and P_2, s.t., P_1 contains id_1, P_2 contains id_2, and P_1 has a foreign key κ'' (whose variable is also id_2) referencing P_2. Then the ED mapping rule for κ is $P_1.\kappa$ **WHEN** $P_1.\kappa'' = p$, where p is a functional path from κ to κ' in the given business entity.

Example 3.2.16 Consider the Clio mapping rule in Example 3.2.14. id_s is the variable representing the key *aServiceID* of *aService_Info*. And its parent *aRepair* is the root relation holding key *aID* (with variable id_r). Thus, correspondingly there are two predicates `tRepair` and `tServiceInfo` on the left of the same Clio rule, s.t., they have a matching over id_r on key and foreign key. Thus the ED mapping rule for *aServiceID* is

`tServiceInfo`.*tServiceID* **WHEN**

aServiceID.aService_Info.aID = `tServiceInfo`.*tRepairID_SI* ∎

The last ***Step C*** is to create mapping rules for all the non-key attributes. For each non-key attribute a in the given business entity, let r be the Clio mapping rule holding a variable v for a. Let id be a variable on the right of r, s.t., (1) there is a functional path p from a to id, and (2) there exist predicates P_1, P_2, ..., P_n on the left of r, s.t., id occurs as the key κ_1 in P_1, for each $i \in [1..(n-1)]$, P_i has a foriegn key κ_i' referencing

the key κ_{i+1} of P_{i+1} that match on the same variable, and v occurs as an attribute a' in P_n. Then the ED mapping rule for a is "$= p@P_1(\kappa_1), \kappa_1'@...@P_n(\kappa_n).a'$".

Example 3.2.17 Consider the Clio mapping rule in Example 3.2.14. t is the variable representing the attribute *aTime*. And there exists an attribute *aServiceID* whose variable is id_s, s.t., there is a functional path from *aTime* to *aServiceID*, id_s occurs as the key *tServiceID* in predicate `tServiceInfo`, and t occurs as attribute *tTime* in the same predicate. Thus, the ED rule for *aTime* is

$$= aTime.aServiceID@\texttt{tServiceInfo}(tServiceID).tTime$$ ∎

3.3 Updatability

This section studies the "updatability" of the ED mapping rules. An "updatable" mapping is able to capture the updates on each business entity enactment and propagate the changes to the database, and vice versa. Consider the framework in Fig. 3.2, if an update is applied on process instance 1, then the changes should be reflected in the database. Similarly, if the database is updated, then the corresponding modification should be propagated to each affected process instance. As illustrated in Section 3.1, not every mapping is "updatable".

Although the ED rules are used to specify corresponding data entries in a database for primitive attributes in a business entity, it is more convenient to define and study mappings from databases to enactments since a database may contain more data entries while every attribute (primitive or complex) in an enactment is stored in the database.

Recall that **Dom** is the domain for all primitive attributes. A *permutation* of **Dom** is a 1-1 and onto mapping from **Dom** to **Dom**. Permutations are naturally extended to tuples, relations, databases, and complex attributes.

Definition: Let $\mathbf{S}$ be a database schema and Ω a business entity. A partial mapping μ from $inst(\mathbf{S}) \times \textbf{Dom}$ to $Ent(\Omega)$ is a *database-enactment* mapping (or DE-mapping) if for each permutation π of **Dom**, each $d \in inst(\mathbf{S})$, and each $v \in \textbf{Dom}$, the following conditions hold whenever $\mu(d, v)$ is defined: (a) $\mu(\pi(d), \pi(v)) = \pi(\mu(d, v))$, and (b) v is the value for the ID attribute in the enactment $\mu(d, v)$.

Note that in the above definition, a DE-mapping takes a database and a value as inputs and produces an enactment whose ID is the input value. Condition (a) is also referred as "genericity" in the study of database queries [22], which essentially forbids manipulations on values (e.g., concatenations, arithmetic, etc.) and forces the mapping to treat values as (uninterpreted) symbols.

Let $\mathbf{S}$ be a database schema, Ω a business entity, and M an ED cover. Define the mapping μ_M from $inst(\mathbf{S}) \times \textbf{Dom}$ to $Ent(\Omega)$ as $\mu_M(d, v) = \sigma \in Ent(\Omega)$ if σ has its ID value v and $(\sigma, d) \models r$ for each $r \in M$; $\mu_M(d, v)$ is undefined otherwise. The following is a consequence of Lemma 3.2.6.

Lemma 3.3.1 For each database schema $\mathbf{S}$, each business entity Ω, and each ED cover M of mapping rules, μ_M is a DE-mapping.

In this chapter, three types of database updates are considered: *insertion*, *deletion*, and *modification*. Given a database d, an insertion, deletion, and modification operation on d adds a new tuple to d, removes a tuple from d, and, resp., changes the value of a non-prime attribute in a given tuple in d. We only consider updates whose the result database satisfies all key and foreign key constraints. For a database schema $\mathbf{S}$, let $\Delta_{\mathbf{S}}$ be the set of all possible updates on $inst(\mathbf{S})$. Given a database $d \in inst(\mathbf{S})$, for each $\delta \in \Delta_{\mathbf{S}}$, $\delta(d)$ denotes the resulting database after applying δ on d.

Similar to database updates, three types of business entity updates are considered:

insertion, *deletion*, and *modification*. Given an enactment σ, an insertion, deletion, and modification operation on σ adds a new tuple to a set in σ with its (local) key defined, removes a tuple (in which each set should be empty) from σ, and changes the value of a non-key primitive attribute in σ. We consider only updates whose result is an enactment. Let $\delta(\sigma)$ be the result of applying an update δ on an enactment σ.

Given an enactment σ of a business entity $\Omega(\mathsf{a}, K, L, dep)$, an update on an attribute a is *eligible on* σ if each attribute in the graph induced by *dep* having a path to a has a non-$\perp$ value in σ. For an enactment σ and an instance I of a business entity Ω, the set of all eligible updates on σ, I, is denoted as $\Delta^{\sigma}_{\Omega}, \Delta^{I}_{\Omega}$ (resp.), and conveniently as Δ_{Ω} when clear.

We now define the central notion of "updatability" of this section. Roughly speaking, a DE-mapping is database-updatable if every update δ^d on a database d corresponds to an update δ^e on an enactment σ such that the DE-mapping is preserved, i.e. updating the database d with δ^d and then applying the DE-mapping is identical to applying the DE-mapping first followed by the update δ^e on σ. The converse direction is business entity-updatability. A slight technical problem is that each update δ^d (or δ^e) often corresponds to a sequence of updates on the enactment (resp. database). Thus the following definition allows sequences of updates.

Definition: Let $\mathbf{S}$ be a database schema, Ω a business entity, $\Delta_s \subseteq \Delta_{\mathbf{S}}$ and $\Delta_\omega \subseteq \Delta_\Omega$ classes of updates on $\mathbf{S}$ and Ω (resp.). A DE-mapping μ is said to be

- *database-updatable with respect to* Δ_s, Δ_ω if for each database update $\delta^d \in \Delta_s$, there is a sequence $\overline{\delta^e}$ of business entity updates in Δ_ω such that for all $d \in inst(\mathbf{S})$ and $v \in \mathbf{Dom}$, $\mu(\delta^d(d), v) = \overline{\delta^e}(\mu(d, v))$;
- *business entity-updatable w.r.t.* Δ_s, Δ_ω if for each business entity update $\delta^e \in \Delta_\omega$ there is a sequence $\overline{\delta^d}$ of database updates in Δ_s such that for all $d \in inst(\mathbf{S})$ and $v \in \mathbf{Dom}$,

$\mu(\overline{\delta^d}(d), v) = \delta^e(\mu(d, v))$;

- *updatable w.r.t.* Δ_s, Δ_ω if it is both database-updatable and business entity-updatable w.r.t. Δ_s, Δ_ω.

Updatability states that it is the same effect whether to apply an update followed by a mapping or a mapping followed by an update. With updatability, each update on databases can be propagated to business entities, and vice versa.

The notion of DE-mapping is similar to the schema mapping in [6]; however, the schema mapping studies did not concern updates. For business entity-updatability, it is similar to view updates [3, 4] in relational databases; however, view updates focus on relational models and business entity-updatability focuses on hierarchical models. Further, it is not clear if the view complement approaches [3, 5] can be adopted in this work. Updates on XML views over relational databases were studied in [23] with a focus on complexity and/or testing whether an XML view update can be translated. In comparison, DE-mappings corresponds to very restricted views and we focus on sufficient conditions to ensure that updates can be translated.

Theorem 3.3.2 For each database schema $\mathbf{S}$, each business entity Ω, and each ED cover M, the DE-mapping μ_M is database-updatable with respect to $\Delta_{\mathbf{S}}$ and Δ_Ω.

Theorem 3.3.2 is easy to prove. Since an ED cover identifies a database attribute (value) for each attribute (value) in a business entity, the mapping can be easily expressed as a database query. Therefore database updatability follows immediately. However, Theorem 3.3.2 fails for business entity-updatability, which was illustrated in Example 3.1.1.

Given r as a mapping rule for some primitive attribute or (local) key γ in a business entity, if r is of form "$R.\kappa$ (**WHEN** φ)" then each primitive attribute in γ is said to be

associated with each attribute in κ; otherwise, if r is of form "$= p_0.[p_1, ..., p_n]@R_1(\kappa'_1).\kappa_1 @...@R_m(\kappa'_m).a$", then γ is *associated with* a.

Let M be an ED cover for a database schema $\mathbf{S}$ and a business entity Ω. Two primitive attributes in Ω are *overlapping* if the two sets of database attributes that they are associated with (resp.) are not disjoint.

Theorem 3.3.3 For each database schema $\mathbf{S}$, each business entity Ω, and each ED cover M, if primitive attributes in Ω are pairwise non-overlapping, then the DE-mapping μ_M is business entity-updatable with respect to $\Delta_{\mathbf{S}}$ and Δ_Ω.

Proof: (Sketch) Let $\mathbf{S}$, Ω, M, $\Delta_{\mathbf{S}}$, and Δ_Ω be as stated in the theorem. For each $\delta \in \Delta_\Omega$, suppose a is the attribute to be updated by δ. Let $M(a)$ be the mapping rule in M for a, σ and d an enactment and instance of Ω and $\mathbf{S}$ respectively, s.t., for each $r \in M$, $(\sigma, d) \models r$, and $\hat{\sigma}$ an enactment by applying δ on σ. Two cases are considered.

(1) a does not occur as a tail in each mapping rule in M. Since no two attributes are overlapping, for each $r \in M - \{M(a)\}$, $\hat{\sigma}, d \models r$. Thus, the sequence of updates on d only need to concern with how to satisfy $M(a)$, which will not affect the satisfiability of other mapping rules in M. The details are given below.

- If a is mapped to a non-key and non-foreign-key database attribute a', then modify the value of a' in d to the same value of a, or
- If a is mapped to an attribute in a key κ of relation schema R in $\mathbf{S}$, suppose that the mapped value is v_κ; then either
  - update nothing if there exists a tuple of R in d with the key value v_κ, or
  - insert a new tuple in d with key value v_κ if such tuple did not exist, or
  - if a is mapped to an attribute in a foreign key, similar to the previous cases, a new tuple may be inserted (if needed) and the value of the foreign key may be updated.

(2) a occurs as a tail in another mapping rule $r \in M$ for an attribute a' in Ω, then in additional to the similar considerations in case (1), we also needs to update the value for a' in the corresponding (maybe newly inserted) tuple in d. Moreover, if a' occurs as a tail in some other mapping rules in M as well, then, the same technique can be applied (recursively). ∎

Corollary 3.3.4 Let $\mathbf{S}$ be a database schema, Ω a business entity, M an ED cover, and $\Delta_s \subseteq \Delta_{\mathbf{S}}$ and $\Delta_\omega \subseteq \Delta_\Omega$ classes of updates on $\mathbf{S}$ and Ω (resp.). The DE-mapping μ_M is updatable with respect to Δ_s and Δ_ω if either (1) primitive attributes in Ω are pairwise non-overlapping, or (2) Δ_ω contains no updates on any overlapping attribute.

The set of mapping rules in Fig. 3.3 is updatable as all the primitive attributes in *aRepair* business entity (Fig. 2.4) are pairwise non-overlapping.

Finally, we remark that since ED covers correspond to Clio entity maps, the updatability results on ED covers can be extended to entity maps informally. But formal statements are problematic due to presence of undefined values. Nevertheless, entity maps can always be converted into ED covers for implementation of BP data accesses.

3.4 Isolation

In this section, we study the notion of "isolation" that prohibits any situations when two business entities in a system update their attributes that are mapped to the same entry in a database. Consider the framework in Fig. 3.2, if process instances 1 and 2 apply updates on attributes that are mapped to the same entry in the database, then these two instances are "affecting" each other, which may not be intended for the process design.

As shown in Section 3.1, an artifact is usually associated with a concrete lifecycle.

In order to generalize our result, in the following, we abstract different lifecycles into "update patterns" that restrict how artifacts should progress (i.e., update its business entity), such that as long as the lifecycles comply with the specified update patterns, our results in this chapter hold.

Given a business entity Ω, an *update class* of Ω, is a set $\{\text{"insert"}, \text{"delete"}\} \times A_{\text{s}} \cup \{\text{"modify"}\} \times A_{\text{nk}}$, where A_{s} and A_{nk} are the sets of all set and non-key primitive attributes in Ω respectively. Intuitively, an update class is a "template" of possible updates for enactments. Each element in an update class is called an *update type*.

An "update constraint" is a conjunction of equality or inequality conditions of primitive attributes. If the conjunction is satisfied (by an enactment), all updates of the specified update type are not allowed to apply.

Definition: Given a business entity Ω, an *update constraint* over Ω is of form $(\bigwedge_{i=1}^{n} \varphi_i) \not\rightarrow U$, where (1) $n \in \mathbb{N}^+$, (2) for each $i \in [1..n]$, φ_i is either $a_1 = a_2$ or $a_1 \neq a_2$, where a_1, a_2 are primitive attributes in Ω such that there are functional paths from a_1 to a_2 as well as a_2 to a_1, or values in **Dom**, and (3) U is an update type of Ω, s.t., there is a functional path from the attribute in U to each variable in φ_i ($i \in [1..n]$).

Example 3.4.1 For business entity in Fig. 2.4, update constraint "$aCust\_Last\_Name = \perp \wedge aCust\_First\_Name = \perp \not\rightarrow (\text{modify}, aReason)$" indicates that if a requesting customer has not been determined yet, then reason for the request cannot be filled in. Another example could be "$1 = 1 \not\rightarrow (\text{update}, aCust\_Addr)$", denoting that $aCust\_Addr$ cannot be updated. ∎

Definition: A *business entity system* $\mathcal{A}$ is a finite set of business entities whose attribute sets are pairwise disjoint. A *snapshot* of $\mathcal{A}$ is a total mapping that assigns each business entity Ω in $\mathcal{A}$ an instance in $inst(\Omega)$. Let $\text{SH}(\mathcal{A})$ denote all snapshots of $\mathcal{A}$.

Definition: An *artifact system* Σ is a pair $(\mathcal{A}, \eta)$, where $\mathcal{A}$ is a business entity system, and η is a mapping from each $\Omega \in \mathcal{A}$ to a set of update constraints of Ω. A *snapshot* of Σ is a snapshot of $\mathcal{A}$.

Given an artifact system $(\mathcal{A}, \eta)$, an *artifact* is a pair $(\Omega, \mathcal{C})$, where $\Omega \in \mathcal{A}$ and $\mathcal{C} = \eta(\Omega)$ is a set of update constraints over Ω, called *update pattern*.

Given an enactment σ of Ω, an update $\delta \in \Delta^{\sigma}_{\Omega}$ is *applicable* to σ if δ does not violate each update constraint in $\mathcal{C}$ and δ is eligible to σ. The notion of "applicable" can be naturally extended to the artifact system snapshots. Given a snapshot Π of an artifact system, denote Δ^{Π} to be all the applicable updates to Π.

Section 3.3 focuses on DE-mappings that result in individual enactments. In this section, we study mappings from databases to shapshots. We fix $\mathbf{S}$ to be an arbitrary database schema and $\mathcal{A}$ an arbitrary (business entity) system, unless otherwise indicated.

Definition: A mapping μ from $inst(\mathbf{S})$ to $\text{SH}(\mathcal{A})$ is a *database-snapshot* (or *DS-*)*mapping* if for each permutation π of **Dom**, each $d \in inst(\mathbf{S})$, $\mu(\pi(d)) = \pi(\mu(d))$.

DE-mappings and DS-mappings are closely related. For each DS-mapping μ and each business entity $\Omega \in \mathcal{A}$, we define a mapping μ_Ω derived from μ as follows. For each database $d \in inst(\mathbf{S})$ and each $v \in \mathbf{Dom}$, $\mu_\Omega(d, v) = e$ if e is an enactment in $\mu(d)$ with its ID value v, undefined otherwise. Let $\text{DE}(\mu) = \{ \mu_\Omega \mid \Omega \in \mathcal{A} \}$ be the set of mappings derived from μ. Conversely, for each set $\mu_\mathcal{A} = \{ \mu_\Omega \mid \Omega \in \mathcal{A} \}$ of DE-mapping, we define the mapping $\text{DS}(\mu_\mathcal{A})$ as: $\text{DS}(\mu_\mathcal{A})(d) = \bigcup_{\Omega \in \mathcal{A}} \{ \mu_\Omega(d, v) \mid \text{if } v \in \mathbf{Dom} \text{ and } \mu_\Omega(d, v) \text{ is defined} \}$.

Lemma 3.4.2 Let $\mu_\mathcal{A} = \{ \mu_\Omega \mid \Omega \in \mathcal{A} \}$ be a set of DE-mappings and μ a DS-mapping. Then $\text{DE}(\mu)$ is a set of DE-mappings and $\text{DS}(\mu_\mathcal{A})$ is a DS-mapping.

Definition: A DS-mapping μ is *business entity-updatable* if each DE-mapping in $\text{DE}(\mu)$ is business entity-updatable.

In the remainder of this section, we only focus on business entity-updatable DS- or DE-mappings. While the Lemma 3.4.2 states that DE-mappings can be naturally extended to DS-mappings, and vice versa, additional issues may arise due to "conflicting" updates by two enactments. In some cases, two updates upon two separate enactments (possibly of the same business entity) might be propagated to the same attribute in the same tuple in a database. Under such circumstances, maintenance of business entities as well as the databases becomes more complicated, and their semantic implications may not be clear. In our formal analysis, we intend to identify DS-mappings that do not have such conflicts, which may serve as a guideline for design or the execution engine to manage executions.

Let a be a complex attribute, K a set of keys and local keys in a, and A a (possibly infinite) set of primitive attributes. Let κ be the set of all primitive attributes in (local) keys in K. The *projection* of a on A, denoted as $\Pi_A(a)$, is a complex attribute with all primitive attributes not in $A \cup \kappa$ removed. The projection operation is naturally extended to values of attribute a. We extend projection to Ω and enactments in $Ent(\Omega)$, $\mathcal{A}$ and snapshots in $\text{SH}(\mathcal{A})$ naturally. Given updates $\Delta \subseteq \Delta_\Omega$, let $W(\Delta)$ be the set of all attributes updated by operations in Δ.

Definition: Given $\mathbf{S}$ as a database schema, $\Sigma = (\mathcal{A}, \eta)$ an artifact system, and μ a DS-mapping from $inst(\mathbf{S})$ to $\text{SH}(\mathcal{A})$, μ is *isolating w.r.t.* η, if for each $d \in inst(\mathbf{S})$ and each business entity update $\delta^e \in \Delta^{\mu(d)}$, there is a sequence $\overline{\delta^d}$ of database updates in $\Delta_\mathbf{S}$ such that $\Pi_{W(\Delta^{\mu(d)})}(\mu(\overline{\delta^d}(d))) = \Pi_{W(\Delta^{\mu(d)})}(\delta^e(\mu(d)))$.

An isolating DS-mapping prohibits the situation where two running business entity enactments can both update two attributes respectively that are mapped to the same entry in the same tuple in a database, hence, "affecting" each other in a snapshot. Isolation is needed for the SeGA tool [24] to work properly.

Example 3.4.3 In Fig. 2.4, if an *aRepair* enactment has attribute customer address (*aCust_Addr*) updated, it may affect some other enactment(s), e.g., of *Custemer Info Review*, that may also be able to update the customer address. Therefore, the underlying DS-mapping is not isolating. ∎

Notice that it is not undesirable if a DS-mapping is not isolating. Isolation is only to provide a guideline for the process designers to know that two business entities may interfere each other. By understanding the interference, a designer can add update constraint to prevent such situation when necessary.

Example 3.4.4 Continue with Example 3.4.3. A designer can add update constraint "$1 = 1 \not\rightarrow (\text{update}, aCust\_Addr)$" to prevent an *aRepair* enactment from updating the customer addresses. Adding this constraint is intuitive as the repair process only serves for the repair purpose. ∎

Theorem 3.4.5 Let $\mathbf{S}$ be a database schema, $(\mathcal{A}, \eta)$ an artifact system, and M be a set of ED mapping rules whose derived DS-mapping μ_M (from $inst(\mathbf{S})$ to $\text{SH}(\mathcal{A})$) is business entity-updatable. Then it can be determined in exponential time, whether μ_M is isolating with respect to η.

To prove Theorem 3.4.5, an algorithm is provided. The main idea is that for each pair of business entities in a system, first compute a "conflict set", whose elements are pairs of attributes that are mapped to the same attribute in database. Then is to run a symbolic execution to check if two updates of two business entity enactments can be applied to the corresponding conflict sets.

In general, a non-empty conflict set for two business entities does not imply that corresponding mapping is not isolating. If no two updates can be applied during the

execution upon each pair of elements (resp.) in each conflict set, the mapping is still isolating.

Before computing the "conflict sets" (for two business entities), we first introduce the notion of "nonconflicting" for mapping rules, such that if a database attribute is mapped by such a mapping rule, then this attribute can only be "accessed" by its corresponding primitive business entity attribute or (local) key.

Given a business entity Ω, a database schema $\mathbf{S}$, and an updatable mapping rule set M (with respect to Ω and $\mathbf{S}$), a mapping rule $r \in M$ (for a (local) key or a primitive attribute a in Ω) is *nonconflicting* if one of the following conditions holds.

- a is a key,
- if r is of form "$R.\kappa$ **WHEN** φ", then there exist some tails of some functional multi-paths in φ that form a key, or
- if r is of form "$= p_0.[p_1, ..., p_n]@R_1(\kappa'_1).\kappa_1@...@R_m(\kappa'_m).a$", then some tails of $p_1, ..., p_n$ form a key, and for each $i \in [1..(m-1)]$, either $\kappa'_i \subseteq \kappa_i$, or $\lambda(\kappa_i)$ is "?" or "+".

Example 3.4.6 Based on the mapping rules shown in Fig. 3.3, the mapping rule for attributes "a*Cust_Addr*" and "a*RP_Phone*" are not nonconflicting; and the mapping rules for other attributes are nonconflicting. ∎

Definition: Given two business entities Ω_1 and Ω_2 a database schema $\mathbf{S}$, and two updatable mapping rule sets M_1, M_2 (with respect to Ω_1 and $\mathbf{S}$, Ω_2 and $\mathbf{S}$ respectively), the *conflict set* of Ω_1 and Ω_2, denoted as $cf(\Omega_1, \Omega_2)$, is a set of pairs of non-key primitive attributes or (local) keys, such that for each $(a_1, a_2) \in cf(\Omega_1, \Omega_2)$, (1) a_1 and a_2 have mapping rules in M_1 and M_2 respectively, (2) both the mapping rules for a_1 and a_2 are not nonconflicting, and (3) the two (sets of) attributes associated with a_1 and a_2 have overlapping.

Example 3.4.7 Suppose a system only contains two business entities *aRepair1* and *aRepair2*, whose structures are exactly the same and shown in Fig. 2.4. Suppose both of them adopt the mapping rules defined in Fig. 3.3. The conflict set of *aRepair1* and *aRepair2* only contains two pairs (*aCust_Addr*, *aCust_Addr*) and (*aRP_Phone*, *aRP_Phone*). ∎

Intuitively, a conflict set denotes all the possible attributes in two business entities that can be mapped into the same attribute in the same tuple in a database during the execution.

Given a business entity $\Omega(\mathsf{a}, K, L, dep)$, a *symbolic business entity* of Ω is a complex attribute $\tilde{\Omega}$, where $\tilde{\Omega}$ is obtained from a by replacing each set attribute $a\colon\{(a_1, ..., a_n)\}$ by a tuple attribute $a\colon(a_1, ..., a_n)$.

Given a business entity Ω and a set of update constraints C of Ω, for each $c \in C$, denote $\mathrm{N}(c)$ to be the number of distinct (primitive) attributes and values in c. Let $\mathrm{N}(C)$ be the largest $\mathrm{N}(c)$ for each $c \in C$. Then the (*symbolic*) *domain* of Ω is $\{v_1, v_2, ..., v_{\mathrm{N}(C)}, \perp\}$, where for each $i \in [1..\mathrm{N}(C)]$, v_i is a distinct "defined" value.

A *symbolic enactment* $\tilde{\sigma}$ of a symbolic business entity is an assignment of each primitive attribute in $\tilde{\sigma}$ to an element in the symbolic domain; and $\tilde{\sigma}$ should comply with the access dependency, i.e., an attribute can have a defined value only if each attribute it depends on has a defined value. Notice that given a symbolic enactment $\tilde{\sigma}$ with domain $\{v_1, v_2, ..., v_{\mathrm{N}(C)}, \perp\}$, it is sufficient to check if $\tilde{\sigma}$ can satisfy each constraints in C.

For reading convenience, for each attribute, set of attributes, key, or local key κ in a business entity Ω, we denote the corresponding "attribute", "set of attributes", "key", or "local key" of κ as $\tilde{\kappa}$ in its corresponding symbolic business entity $\tilde{\Omega}$.

Similar to the "insertion", "deletion", and "modification" updates for a business

entity, a symbolic business entity can also have these three updates as well. More specifically, given a business entity Ω, a symbolic business entity $\tilde{\Omega}$ of Ω, and a set of update constraints C of Ω, an insertion of a complex attribute $\tilde{a}$ (based on a symbolic enactment of $\tilde{\Omega}$) is to assign a set of child attributes $\tilde{K}$ of $\tilde{a}$ from $\perp$'s to elements in $\{v_1, ..., v_n\}$, such that a is a set attribute and K forms a (local) key for a in Ω; a deletion is to assign each attribute in a tuple to $\perp$; and a modification is to assign a non-key primitive attribute to an element in $\{v_1, ..., v_n\}$.

An update is *applicable* with respect to a symbolic enactment and a set of update constraints, if no constraint prohibits the applying of the update and after the update, the result assignment still forms a symbolic enactment.

Based on a symbolic enactment, a finite set of "symbolic updates" can be computed. Given business entity Ω, a symbolic business entity $\tilde{\Omega}$ of Ω, an update pattern C of Ω, and a symbolic enactment $\tilde{\sigma}$ of $\tilde{\Omega}$, a *symbolic update set* $\Delta(\tilde{\sigma}, C)$ is a set of all applicable insertions, deletions, and modifications based on $\tilde{\sigma}$.

Given two business entities Ω_1 and Ω_2 a database schema $\mathbf{S}$, and two updatable mapping rule sets M_1, M_2 (with respect to Ω_1 and $\mathbf{S}$, Ω_2 and $\mathbf{S}$ respectively), Ω_1 and Ω_2 are *independent* (with respect to M_1 and M_2) if and only if for each pair of symbolic enactments $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ of $\tilde{\Omega}_1$ and $\tilde{\Omega}_2$ (resp.), each $\delta_1 \in \Delta(\tilde{\sigma}_1, C)$, $\delta_2 \in \Delta(\tilde{\sigma}_2, C)$, and each pair of attributes $(\tilde{a}_1, \tilde{a}_2)$ updated by δ_1 and δ_2 (resp.), there does not exists a pair $(\kappa_1, \kappa_2) \in cf(\Omega_1, \Omega_2)$, such that a_1 occurs in κ_1 and a_2 occurs in κ_2.

A DS-mapping is isolating if and only if each pair of (possibly the same) business entities in the system is independent (with respect to their corresponding mapping rules).

Example 3.4.8 Continue with Example 3.4.7; the mapping from the database in Fig. 2.2 to this system is isolating with respect to constraints "$1 = 1 \not\rightarrow$ (update, *aCust_Addr*)"

and "$1 = 1 \not\rightarrow$ (update, *aRP_Phone*)". ∎

The complexity to determine the conflict sets is polynomial with respect to the size of the database schema, business entities, and mapping rules. While the complexity of the symbolic execution is exponential with respect to to the size of all the business entities.

Notice that the algorithm described above is to check the "write-write" conflicts within a system. It is straightforward to adapt this algorithm to check the "write-read" conflicts, i.e., an update upon an attribute within an enactment will not affect the value of each other enactment in the system.

Similar to the remark at the end of Section 3.3, the isolation results on ED covers can be extended to entity maps.

3.5 Summary

This chapter initiates a study on data mappings between BPs and databases through formalizing the data models and formulating a mapping language. This idea of bridging BPs and databases has a potential to allow management issues to be dealt with separately for BPs and for databases while making sound design decisions. For BPM, it allows many interesting problems to be studied in the presence of data, e.g., process evolution. For databases, it brings a new dimension, i.e. BPs, into the database design, in particular, by including BPs' data needs, database design could avoid problems such as missing data or mismatched semantics.

On the technical front, there are many interesting problems to be addressed. A better understanding is needed for specifying entity-data mappings, alternative languages and relaxing the attribute-attribute mapping requirement are worth considering. Concerning data mapping properties, are there general requirements other than updatability and isolation? For example, general integrity constraints in databases might add more difficulties

to updatability. The problems are more interesting to be studied along with lifecycle for bentities, e.g., the issue of BP independence could cleanly separate the "footprints" on data by two BP executions.

Chapter 4

Data Mappings: Identifying the Source

In Chapter 3, ED mapping was proposed to be a mapping language to bridge the BP data and the enterprise database. However, ED mapping is rather restricted in terms of syntax, which limits the expressive power of the way of transforming one data format into another. On the contrary, Clio mapping, which has been proved to be a superset of ED mapping in Chapter 3 is more expressive as a mapping language. Naturally, it is interesting to understand how updatability can be determined if the data mapping is specified by Clio.

Essentially, Clio mapping is a variant of a mapping language called "source-to-target tuple generating dependency", or simply, tgds. the language of tgds is a subset of first-order logic that has been widely studied in the data exchange community. For example, a tgd mapping can be $A(x, y) \wedge A(y, z) \rightarrow B(x, z)$ and $A(x, y) \wedge A(y, z) \rightarrow C(y)$ to denote that a source table A will be transformed into two target tables B and C based on the mapping. In detail, a tgd mapping is a triple $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ and $\mathbf{T}$ represent the source and target schema and Σ is a set of tgd rules to specify the mapping between $\mathbf{S}$

and $\mathbf{T}$.

A number of data exchange studies [7, 25] focus on how to produce the instance of $\mathbf{T}$ given an instance of $\mathbf{S}$ and $\mathtt{M}$, where "chase" [7, 25] is a widely used algorithm to achieve the goal. Also, there are cases, where target database is updated and consequently, the source database should also be updated. This property is called "updatability" [26], which refers to if a target database is updated, is there a sequence of updates to the source database such that the new target can still be "chased" from the new source? Updatability is essentially equivalent to the "Chase Validity Problem" (CVP), i.e., given a schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance J of $\mathbf{T}$, does there exist an instance I of S, such that J can be chased from I through $\mathtt{M}$? To address this problem, the notion of inverted mapping is proposed [27, 28, 29], which mainly addresses the expressiveness of the inverted mapping language instead of existence of the source database. To the best of what we know, [30] is the first paper to show that this problem is NP-Complete.

Therefore, in this chapter, we study several subclasses of tgd mapping to show when the intractable cases can be solved in PTIME. Specifically, we highlight the following three properties of a tgd mapping:

self-joined: Same source predicate appear at least twice in the same rule

$A(x, y) \wedge A(y, z) \rightarrow S(x, z)$

unioned: Same target predicate appear in at least two different rules

$A(x, y) \wedge B(y, z) \rightarrow S(x, z)$ and $C(x, y, z) \rightarrow S(x, y)$

multi-viewed: There are at least two different target predicates

$A(x, y) \wedge B(y, z) \rightarrow S(x, z)$ and $C(x, y, z) \rightarrow T(x, y)$

In details, we study all 8 combinations of the above 3 properties (i.e., {self-joined, not self-joined} $\times$ {unioned, not unioned} $\times$ {multi-viewed, not multi-viewed}) under 4 different circumstances: {with key constraints, without key constraints} $\times$ {entire source unknown, part of the source unknown} and present the following contributions:

- Theorem 4.2.5: CVP is NP-Complete if the mapping is self-joined, unioned, or multi-viewed; otherwise, is in PTIME (Fig. 4.1).
- Propositions 4.3.1 and 4.3.2: CVP with key constraints is NP-Complete if the mapping is self-joined, unioned, or multi-viewed; otherwise, is in PTIME (Fig. 4.1).
- Theorem 4.3.4: Given a "key-preserving" mapping, CVP is NP-Complete if the mapping is unioned; otherwise, is in PTIME (Fig. 4.2).
- Theorem 4.4.1: Given two or more source relations missing, CVP is NP-Complete even when the mapping is not self-joined, not unioned, and not multi-viewed (Fig. 4.3).
- Theorem 4.4.4: Given only one source relation missing, CVP is NP-Complete if the mapping is self-joined; otherwise, is in PTIME (Fig. 4.4).
- Theorem 4.5.1: Given only one source relation missing, CVP with key constraints is NP-Complete even when the mapping is not self-joined, not unioned, and not multi-viewed (Fig. 4.3).

The remainder of this chapter is organized as follows: Section 4.1 defines the notions and the main problem. Section 4.2 investigates the cases where no additional constraints are enforced, comparing with Section 4.3, where keys are added. Section 4.4 studies the case, where only part of the source relation is unknown. Similarly, Section 4.5 complicates the problem by adding key constraints. Finally, summary is discussed in Section 4.6.

4.1 Preliminaries and Problem Definition

In this section, we review some database notions that have been mentioned in Section 2.1, define the key notion of schema mapping, and state the main problem we will focus

on in this chapter.

Schema and Instances A *schema* $\mathbf{R}$ is a finite set $\{R_1, R_2, ..., R_n\}$ of *relation symbols*, each of which has a fixed *arity*. Let two disjoint countably infinite sets Const and Var be a set of *constants* and a set of *labeled nulls* respectively, where a labeled null is used to denote uncertain values [7]. Let **Dom** be Const $\cup$ Var. An *instance* I of $\mathbf{R}$ is a set $\{R_1^I, R_2^I, ...R_n^I\}$ of *relations* (or *tables*), where each R_i^I is a finite relation of the same arity as R_i taking the values from **Dom**. A *ground* instance is an instance that only contains values from Const. Each row in a relation R^I in I is called a *tuple* of R^I.

Schema Mapping In the following, we assume $\mathbf{S}$ and $\mathbf{T}$ to be two fixed schemas, indicating that data is mapped from a *source* $\mathbf{S}$ to a *target* $\mathbf{T}$. A *schema mapping* (or *mapping* for short) is a triple $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where Σ is a set of constraints describing the relationship between $\mathbf{S}$ and $\mathbf{T}$. Mapping $\mathtt{M}$ essentially defines a set of pairs of instances as follows:

$$\{(I, J) \mid I \text{ and } J \text{ are instances of } \mathbf{S} \text{ and } \mathbf{T} \text{ resp.}, \langle I, J\rangle \models \Sigma\}$$

We call J is a *solution* to I w.r.t. $\mathtt{M}$ if $(I, J) \in \mathtt{M}$.

TGDs and Full Schema Mapping Given a schema $\mathbf{S}$, a *conjunctive formula* over $\mathbf{S}$ is a conjunction of atomic formulas of form "$R(\bar{z})$", where R a relation symbol (or *predicates* in general terms) in $\mathbf{S}$ and $\bar{z}$ is a vector of (possibly duplicated) variables, whose length agrees on the arity of R.

A *source-to-target tuple generating dependency* (or, in short, *tgd*) is a first-order logic formula of form

$$\forall \bar{x}\ \Phi_{\mathbf{S}} \rightarrow \exists \bar{y}\ \Psi_{\mathbf{T}} \tag{i}$$

where $\bar{x}$ and $\bar{y}$ are disjoint sequences of first-order variables, $\Phi_{\mathbf{S}}$ and $\Psi_{\mathbf{T}}$ are conjunctive

formula over $\mathbf{S}$ and $\mathbf{T}$, s.t., there is no free variables in (i), i.e., each variable in $\Phi_{\mathbf{S}}$ should occur in $\bar{x}$ and each variable in $\Psi_{\mathbf{T}}$ should occur in $\bar{x}$ or $\bar{y}$. (Recall that a *free variable* in a first-order logic formula is a variable that is neither universal nor existential quantified.)

A *full* tgd is a formula of form (i) without $\exists\bar{y}$. In this chapter, we focus a mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with Σ a set of full tgds. a schema mapping with a set of full tgds is called a *full schema mapping*. In the remainder of this chapter, we may drop the universal ($\forall$) and the existential ($\exists$) quantifiers for convenience. For notation convenience, denote $|\Phi_{\mathbf{S}}|$ to be the number of (possibly duplicated) relation symbols in $\Phi_{\mathbf{S}}$ and $\Phi^i_{\mathbf{S}}$ to be the i^{th} atomic formula $R(\bar{z})$ in $\Phi_{\mathbf{S}}$, where $i \in [1..|\Phi_{\mathbf{S}}|]$.

For technical development, we introduce *assignment* for a first-order logic formula φ, which is a total mapping from all the free variables in φ to **Dom**. Given an assignment α, a first-order logic formula φ, and a vector of free variables $\bar{x}$ in φ, we denote $\alpha(\bar{x})$ to be the assignment of $\bar{x}$ under α and $\varphi[\alpha]$ to be the formula of replacing each free variable x by $\alpha(x)$.

Example 4.1.1 Consider a source schema $\mathbf{S}$ with a single binary relation E and a target schema $\mathbf{T}$ with a binary relation F and a unary relation G. The following tgd specifies a mapping from $\mathbf{S}$ to $\mathbf{T}$:

$$E(x, z) \wedge E(z, y) \rightarrow F(x, y) \wedge G(z) \tag{ii}$$

Consider a $\mathbf{S}$ instance I that only contains a single relation $E^I = \{(1, 1), (2, 2), (1, 2)\}$ of E. Then the $\mathbf{T}$ instance J with two relations $F^J = \{(1, 1), (2, 2), (1, 2)\}$ of F and $G^J = \{(1), (2)\}$ of G is a solution to I w.r.t. the mapping specified by (ii). However, if we drop any tuple in F^J or G^J, the new instance is no longer a solution. ∎

Chase Given a conjunctive formula $\Phi_{\mathbf{S}}$ over a schema $\mathbf{S}$ and a $\mathbf{S}$ instance I, a *permutation*

of $\Phi_{\mathbf{S}}$ wrt I is a vector of tuples $(\tau_1, \tau_2, ..., \tau_m)$, where $m = |\Phi_{\mathbf{S}}|$ and for each $i \in [1..m]$, τ_i is a tuple of R^I, where R is the relation symbol of $\Phi^i_{\mathbf{S}}$; A permutation is *valid* under assigment α, if for each $i \in [1..m]$, $\tau_i = \Phi^i_{\mathbf{S}}[\alpha]$.

Given a mapping `M` speicifed by a finite set of tgds and its source instance I, a *chase* procedure that takes `M` and I as inputs, can produce a solution to I w.r.t. `M` (or more precisely, a "universal solution", see [7] for details). We use $chase_{\mathtt{M}}(I)$ to denote the corresponding output.

Let I and `M` be as stated above; we briefly describe how chase (for tgds) works. A chase procedure starts with the instance pair $\langle I, J = \varnothing \rangle$. For each tgd $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}}$ in `M` and each valid permutation of $\Phi_{\mathbf{S}}$ wrt I under assigment α, incorporate n tuples $\sigma_1, \sigma_2, ..., \sigma_n$ into J where $n = |\Psi_{\mathbf{T}}|$, such that (1) for each $j \in [1..n]$, $\sigma_j = \Psi^j_{\mathbf{T}}[\beta]$ under the same assignment β, (2) α and β agree on the same variables, and (3) each existential quantified variables (in $\Psi_{\mathbf{T}}$) is mapped to a unique labeled null in β.

Example 4.1.2 Continue with Example 4.1.1. We apply the chase procedure on mapping (ii) and source instance E^I. As the left-hand side of (ii) contains two relation symbols, we consider all the permutations of E^I. For permutation $(E(1,1), E(2,2))$, it is impossible to have an assignment α, such that $E(1,1) = E(x,z)[\alpha]$ and $E(2,2) = E(z,y)[\alpha]$ as there is no assignment that can map z to both 1 and 2; therefore ignored. For permutation $(E(1,1), E(1,2))$, we can have assignment $x = 1$, $y = 2$, and $z = 1$ to be a "witness"; and therefore incorporate $F(1,2)$ and $G(1)$ to the target instance (which originally contains nothing). Similarly, we can apply the same approach to all other permutations of E^I; and target instance J in Example 4.1.1 will be the output of this chase procedure. ∎

In this section, we address the problem of given a mapping and a target instance, check

if this target instance can be chased from some source instance throught the mapping.

Chase Validity Problem (CVP) Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance J of $\mathbf{T}$, does there exist an instance I of S, such that $J = chase_M(I)$?

4.2 Valid Chased Targets

The Chase Validity Problem has been proved to be NP-Complete in [30]. In the remainder of this section, we focus on the same problem regarding some subcases of full tgds (i.e, with self-join, union, or multiple target predicates). It turns out that though some subcases are quite restrictive, they still remain NP-Complete.

In order to have a simpler statement and reasoning, we assume that each full tgd contains exactly one relation symbol on the right-hand side. The following Lemma 4.2.1 and Corollary 4.2.2 show that a set of full tgds with arbitrary right-hand-side relation symbols is equivalent to the one with single right-hand-side relation symbol in terms of chase result.

Lemma 4.2.1 Given a source schema $\mathbf{S}$, a target schema $\mathbf{T}$, and a full tgd of form $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}} \wedge R(\bar{z})$ over $\mathbf{S}$ and $\mathbf{T}$, where $\Phi_{\mathbf{S}}$ and $\Psi_{\mathbf{T}}$ are conjunctions over $\mathbf{S}$ and $\mathbf{T}$ (resp.), R a relation symbol in $\mathbf{T}$, and $\bar{z}$ variables occurring in $\Phi_{\mathbf{S}}$. Let $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \{\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}} \wedge R(\bar{z})\}$, and $\mathtt{M}' = (\mathbf{S}, \mathbf{T}, \{\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}}, \Phi_{\mathbf{S}} \rightarrow R(\bar{z})\}$ be two mappings. Then for every $\mathbf{S}$ instance I, $chase_{\mathtt{M}}(I) = chase_{\mathtt{M}'}(I)$.

Proof: (Sketch) Let I, $\Phi_{\mathbf{S}}$, $\Psi_{\mathbf{T}}$, and $R(\bar{z})$ be as stated in the lemma. For each valid permutation of $\Phi_{\mathbf{S}}$ wrt I under some assigment α, since all variables in $\Psi_{\mathbf{T}}$ and $R(\bar{z})$ occur in $\Phi_{\mathbf{S}}$ due to full tgd, $\Psi^i_{\mathbf{T}}[\alpha]$ ($i \in [1..|\Psi_{\mathbf{T}}|]$) and $R(\bar{z})[\alpha]$ can be incorporated in to

the target instance either based on one rule $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}} \wedge R(\bar{z})$ or two rules $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}}$ and $\Phi_{\mathbf{S}} \rightarrow R(\bar{z})$. ∎

Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, denote $\mathtt{M}^{\mathrm{S}} = (\mathbf{S}, \mathbf{T}, \Sigma^{\mathrm{S}})$ to be a schema mapping, where Σ^{S} is obtained by decomposing each tgd $\Phi_{\mathbf{S}} \rightarrow \bigwedge_{i=1}^{n} R_i(\bar{z}_i)$ in Σ to a set of tgds $\bigcup_{i=1}^{n} \{\Phi_{\mathbf{S}} \rightarrow R_i(\bar{z}_i)\}$, where R_i is a relation symbol in $\mathbf{T}$.

The following corollary is a direct result from Lemma 4.2.1.

Corollary 4.2.2 Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, for each $\mathbf{S}$ instance I, $chase_{\mathtt{M}}(I) = chase_{\mathtt{M}^{\mathrm{S}}}(I)$.

Example 4.2.3 Consider the following two tgds.

$$E(x, z) \wedge E(z, y) \rightarrow F(x, z) \tag{iii}$$

$$E(x, z) \wedge E(z, y) \rightarrow G(z) \tag{iv}$$

The tgd (ii) in Example 4.1.1 is equivalent to (iii) and (iv) in terms of chase result. ∎

Without loss of generality, we always assume that if the given set of tgds are full, the right-hand side of each tgd is singleton.

Definition: Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, $\mathtt{M}$ is (1) *self-joined* if there exists a tgd $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}}$ in Σ, where $\Phi_{\mathbf{S}}$ contains two identical relation symbols, (2) *unioned* if there are two tgds in Σ sharing a common target relation symbol, or (3) *multi-viewed* if there are more than one target relation symbols in Σ.

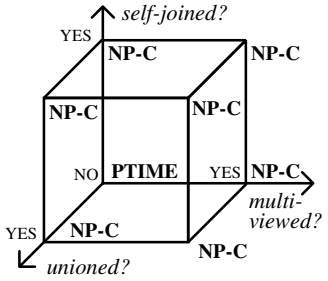

Figure 4.1: No constraints

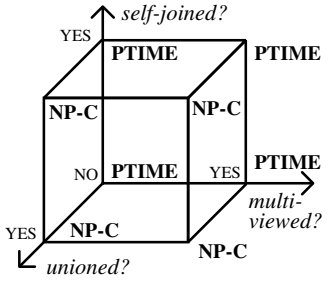

Figure 4.2: Key-preserving

Example 4.2.4 Consider the following two equations and equations (iii) and (iv) in Example 4.2.3.

$$D(x) \wedge E(x, y) \rightarrow F(x, y) \tag{v}$$

$$C(x, z) \wedge E(z, y) \rightarrow H(z) \tag{vi}$$

The equation (iii) is self-joined but not unioned or multi-viewed. The (schema formed by) equations (iv), (v), and (vi) is self-joined and multi-viewed but not unioned. Equation (v) is not self-joined, unioned, or multi-viewed. Equations (iii) and (v) are unioned and self-joined, but not multi-viewed. Equations (iii), (iv), (v), and (vi) are unioned, self-joined, and multi-viewed. ∎

Theorem 4.2.5 If the full schema mapping is self-joined, unioned, or multi-viewed CVP is NP-Complete; otherwise, is in PTIME.

The result of Theorem 4.2.5 is visualized in Fig. 4.1. All combinations of self-joined, multi-viewed, and unioned are NP-Complete expect the one that excludes all three conditions.

In the following, we present four lemmas (4.2.6, 4.2.7, 4.2.8, and 4.2.9) to prove the correctness of Theorem 4.2.5.

Lemma 4.2.6 CVP is NP-Complete if the given full schema mapping is self-joined, but not unioned or multi-viewed.

Lemma 4.2.6 is a direct result from the view consistency problem in [31], where the mapping contains only one full tgd (i.e., a conjunctive query).

Lemma 4.2.7 CVP is NP-Complete if the given full schema mapping is multi-viewed, but not self-joined or unioned.

The membership of NP directly follows from [30]. To prove that CVP is NP-hard given a mapping that contains multiple targets but not union or self-join, we reduce it from the Set Basis Problem [32].

> **Set Basis Problem** Given a finite set S, a collection $C \subseteq 2^S$ of subsets of S, and a positive integer $K \leqslant |C|$, is there a collection $B \subseteq 2^S$ of subsets of S with $|B| = K$, such that for each $c \in C$, there is a subcollection of B whose union is exactly c?

Proof: (NP-hardness of Lemma 4.2.7) Let S, C, B, and K be as stated in the Set Basis Problem. Essentially the Set Basis Problem is to find "bases" B that can form C. Now consider (1) two binary source relation symbols $\hat{B}(s, b)$ to represent if $s \in S$ is in $b \in B$ and $R(b, c)$ to represent if $b \in B$ is a "basis" for $c \in C$, (2) two target relation symbols $\hat{C}(s, c)$ of arity 2 to represent if $s \in S$ is in $c \in C$ and D with arity 1 to indicate the size of B should be no greater than K, and (3) the following two tgds:

$$\hat{B}(s, b) \wedge R(b, c) \rightarrow \hat{C}(s, c) \qquad \hat{B}(s, b) \wedge R(b, c) \rightarrow D(b) \tag{vii}$$

The instances of $\hat{C}$ and D are defined as follows:

$$\hat{C} = \{(s, c) \mid s \in c, \text{ where } s \in S \text{ and } c \in C\} \qquad D = \{(1), (2), ..., (K)\}$$

It can be shown that the Set Basis Problem has a solution if and only if the mapping specified by (vii) with target instance given above has a source database. And the bases B is actually indicated by the source database $\hat{B}$. Note that (vii) is multi-viewed, but not union or self-joined. Therefore Lemma 4.2.7 holds. ∎

Lemma 4.2.8 CVP is NP-Complete if the given full schema mapping is unioned, but not multi-viewed or self-joined.

Proof: The membership of NP directly follows from [30]. To prove that CVP is NP-hard given a mapping that contains union but not multiple targets or self-join, we construct a set of tgds that are "equivalent" to (vii); and the NP-hard result naturally follows Lemma 4.2.7.

Consider the following tgds:

$$\begin{gathered} \hat{B}(s, b) \wedge R(b, c) \wedge F_3(x_3, s, b, c) \rightarrow \hat{C}_{\text{big}}(s, c, x_3) \\ \hat{B}(s, b) \wedge R(b, c) \wedge F_1(x_1, s, b, c) \wedge F_2(x_2, s, b, c) \rightarrow \hat{C}_{\text{big}}(x_1, x_2, b) \end{gathered} \tag{viii}$$

Essentially, $\hat{C}_{\text{big}}$ is a "composed" relation symbol to represent C and D in equation (vii) and F_i's ($i = [1..3]$) are dummy relation symbols only to make (viii) full tgds.

Given a target instance J of $\hat{C}$ and D in (vii), we construct an instance J' of $\hat{C}_{\text{big}}$ in (viii):

1. J' starts with empty content.

2. For each tuple (s, c) in the relation of C in J, we corporate a tuple (s, c, x_3) in the relation of $\hat{C}_{\text{big}}$ in J', where x_3 holds a unique value that is different from each value in the active domain of J.

3. Similarly, for each tuple (s, b) in $\hat{B}$, incorporate a tuple (x_1, x_2, b) in J, where x_1 and x_2 hold unique values that are different from each value in the active domain of J.

It is trivial to show that J has a source instance w.r.t. (vii) if and only if J' has a source instance w.r.t. (viii). Note that (viii) is unioned, but not multi-viewed or self-joined. Therefore Lemma 4.2.8 holds. ∎

Lemma 4.2.9 Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and a ground instance J of $\mathbf{T}$, it is in PTIME to check whether there exists a source instance I of $\mathbf{S}$, s.t., $J = \mathit{chase}_{\mathtt{M}}(I)$.

To prove Lemma 4.2.9, we need a leading Lemma 4.2.10 and the notion of "reversed mapping", which will be used in the proof as well as in the next section.

Lemma 4.2.10 Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, and an instance J of $\mathbf{T}$, if there exists a source instance I of $\mathbf{S}$, s.t., $J = \mathit{chase}_{\mathtt{M}}(I)$, then for each R-tuple τ in I, where $R \in \mathbf{T}$, there exists an assignment α, s.t., $\tau = R[\alpha]$.

Lemma 4.2.10 is straightforward to prove and can serve as a filter to eliminate an apparent "unchaseable" target instance. For example, if the given tgd is $S(x) \rightarrow T(x, x)$ and a target instance J is $T^J = \{(1, 2)\}$, apparently there is no source target can produce T^J through chase.

Definition: Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ that is not unioned, a schema mapping $\tilde{\mathtt{M}} = (\mathbf{T}, \mathbf{S}, \tilde{\Sigma})$ is a *reversed mapping* (*w.r.t* $\mathtt{M}$), where $\Phi_{\mathbf{S}} \rightarrow \Psi_{\mathbf{T}}$ is a tgd in Σ if and only if $\Psi_{\mathbf{T}} \rightarrow \Phi_{\mathbf{S}}$ is a tgd in $\tilde{\Sigma}$.

A reversed mapping is to flip the direction of the arrows in tgds. For example, in Example 4.2.3, the reversed mapping of the one specified by (iii) and (iv) is $F(x, z) \rightarrow E(x, z) \wedge E(z, y)$ and $G(z) \rightarrow E(x, z) \wedge E(z, y)$.

The notion of "reversed mapping" is different from the one in [27, 28, 29], where by simply flipping the arrows will lead to a wrong results in those works.

Moreover, a reversed mapping can be generalized to corporate unions if disjunction is allowed. However, to have a cleaner presentation of this chapter, we only focus on the mapping without union.

Proof: (Lemma 4.2.9) Note that if a given mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ is not multi-viewed, unioned, or self-joined, Σ only contains a single tgd with no repeated relation symbols on the left-hand-side. W.l.o.g., suppose the only tgd is of form $A_1(\bar{x}_1, \bar{y}_1) \wedge A_2(\bar{x}_2, \bar{y}_2) \wedge ... \wedge A_n(\bar{x}_n, \bar{y}_n) \rightarrow B(\bar{x}_1, \bar{x}_2, ..., \bar{x}_n)$, where each A_i ($i \in [1..n]$) is a distinct relation symbol with variables $\bar{x}_i$ and $\bar{y}_i$. Further we assume J is a given target instance of B.

Now consider the reversed mapping $\tilde{\mathtt{M}} = (\mathbf{T}, \mathbf{S}, \tilde{\Sigma})$, where $\tilde{\Sigma} = \{B(\bar{x}_1, \bar{x}_2, ..., \bar{x}_n) \rightarrow A_1(\bar{x}_1, \bar{y}_1) \wedge ... \wedge A_n(\bar{x}_n, \bar{y}_n)\}$. Suppose $I = \mathit{chase}_{\tilde{\mathtt{M}}}(J)$. Based on Lemma 4.2.10, it is easy to see $J \subseteq J' = \mathit{chase}_{\mathtt{M}}(I)$, as it resembles the lossless-join decomposition, where one can only obtain larger or same table by decomposing and join the original table.

Another observation is that J' only contains constants. This is because null values in $I = \mathit{chase}_{\tilde{\mathtt{M}}}(J)$ are only introduced at position y_i ($i \in [1..n]$) in each A_i during the chase procedure with input J and $\tilde{\mathtt{M}}$. Together with that A_i's are distinct, during chase procedure with input I and $\mathtt{M}$ null values will not occur at position x_i in each A_i. Therefore, $J' = \mathit{chase}_{\mathtt{M}}(I)$ does not contain null values.

As a result, to determine if J is chased from some source instance through mapping $\mathtt{M}$ is essentially to check if $J = chase_{\mathtt{M}}(chase_{\tilde{\mathtt{M}}}(J))$. The data complexity is polynomial wrt the size of J due to the reason that the chase procedure is polynomial complexity. ∎

Corollary 4.2.11 CVP is in PTIME if the given full schema mapping is not multi-viewed, not unioned and not self-joined.

Corollary 4.2.11 is a direct result of Lemma 4.2.9 by treating each null value in the given target instance as constants.

Hence, according to Lemmas 4.2.6, 4.2.7, 4.2.8, and Corollary 4.2.11, and the result from [30], Theorem 4.2.5 holds.

4.3 Adding Key Constraints

The results from the previous section indicates that the Chase Validity Problem is intractable under most circumstances even when the tgds have a quite restricted form. Consider that most relations database systems nowadays require keys in the tables, in this section, we focus on relations with key constraints and show that the Chase Validity Problem is in PTIME if the some forms of tgds are "key-preserving".

Given a relation symbol R with arity n, a *functional dependency* (or *FD* for short) over R is of form $X \rightarrow Y$, where X and Y are subsets of $\{1, 2, .., n\}$ (representing columns or "attributes" or R), if for each relation of R, a X value uniquely determines a Y value. The notion of FD over a schema is an extension of the one over a relation symbol.

Now we briefly remind the concepts of keys that have been introduced in Chapter 2.1.

A *key* for a relation symbol R with arity n, is a minimum subset K of $\{1, 2, .., n\}$,

such that K functional determines every column of R, i.e., $K \rightarrow [1..n]$. A *super key* is a subset of $\{1, 2, .., n\}$ and a superset of a key. W.l.o.g., in this section, we always assume each relation of a (database) schema has a key.

Definition: A *schema mapping with key constraints* is a schema mapping, s.t., each relation in the source and target schema has a key.

One interesting question to raise is that given a schema mapping (with key constraints), if the source instance satisfies the key constraints (defined by the source schema), is it always the case that the chased target instance also satisfies key constraints (defined by the target schema)? This problem can indeed be reduce to the results in [33], which gives an algorithm to determine a complete and sound set of FDs given a relational algebra.

Definition: Given a full schema mapping $\mathtt{M}$ with key constraints, $\mathtt{M}$ is *safe* if for each source instance I that satisfies the key constraints, the target instance $chase_{\mathtt{M}}(I)$ satisfies the key constraints.

In essence, a schema mapping that is not safe should be consider as a "bad" design. In the remainder of this section, we only focus on safe mapping.

Chase Validity Problem with Keys ($\mathbf{CVP_k}$) Given a full and safe schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with key constraints and an instance J of $\mathbf{T}$ that satisfies the key constraints of $\mathbf{T}$, is there an instance I of S that satisfies the key constraints of $\mathbf{S}$, such that $J = chase_{\mathtt{M}}(I)$?

The following proposition states a negative results regarding the Chase Validity Problem with key constraints added.

Proposition 4.3.1 Given a safe and full schema mapping M with key constraints, CVP_K is NP-Complete if M is self-joined, unioned, or multi-viewed; otherwise, is in PTIME.

Proof: For the NP-Complete cases, it is straightforward to prove by reducing each (NP-Complete) case (shown in Fig. 4.1) to the corresponding case with key constraints and let each given relation symbol of some arity n to have a key $[1..n]$ (i.e., everything in the relation).

For the only PTIME case (i.e., a single tgd with no self-join), we use the same techniques in the proof of Lemma 4.2.9 to obtain the source instance. And then check if both the target and the source instances satisfy the key costraints. ∎

The following proposition shows a same result even when each relation symbol has a single-column key.

Proposition 4.3.2 Given a safe and full schema mapping M with key constraints, where each key has cardinality of 1, CVP_K is NP-Complete if M is self-joined, unioned, or multi-viewed; otherwise, is in PTIME.

Proof: For the only PTIME case (i.e., a single tgd with no self-join), the proof is similar to the one in Proposition 4.3.1, thus omitted.

If the given schema mapping is self-joined, unioned, or multi-viewed, we construct a new mapping by (1) adding a "dummy" key to each relation symbol, where these keys are not "useful" in the mapping, and (2) adding a "dummy" source predicate for each tgd to serve as a "container" for all the dummy keys. For example, consider equation (vii); the new mapping with "dummy" keys and predicates is as follows.

$$\begin{gathered} K_{\hat{C}}(\underline{k_{\hat{C}}}, k_{\hat{B}}, k_R) \wedge \hat{B}(\underline{k_{\hat{B}}}, s, b) \wedge R(\underline{k_R}, b, c) \rightarrow \hat{C}(\underline{k_{\hat{C}}}, s, c) \\ K_D(\underline{k_D}, k_{\hat{B}}) \wedge \hat{B}(\underline{k_{\hat{B}}}, s, b) \rightarrow D(\underline{k_D}, b) \end{gathered} \tag{ix}$$

where the keys (lower-case k's) are underlined and the upper-case K's are "container" predicates to hold all the "dummy" keys. Intuitively, each source "dummy" key does not "contribute" in the mapping; and the "container" predicates are to enforce the safety property of a mapping. It is trivial to have a reduction to show that the existence of a source instance is "equivalent" w.r.t. equations (vii) and (ix). ∎

Although enforcing key constraints will not reduce the complexity to validate the existence of the source instances, a simple yet general requirement on the form of the tgds provides the tractability to CVP_K.

Definition: A schema mapping with key constraints is called *key-preserving* if for each tgd, every variable that is on a key position in a source predicate occurs in a target predicate (in the same tgd).

Example 4.3.3 Consider the schema in Examples 4.2.3 and 4.2.4, together with the equations (iii), (iv), (v), and (vi). Suppose all the relations (C, D, E, F, G, and H) have key $\{1\}$. Then the mapping that contains some of all of equations (iii), (iv), and (v) is key-preserving; while a schema mapping that contains (vi) is not key-preserving because the target predicate H does not contain the key x of C. ∎

Theorem 4.3.4 Given `M` a key-preserving, safe, and full schema mapping, if `M` is unioned, CVP_K is NP-Complete even when `M` is not self-joined or multi-viewed; otherwise is in PTIME.

Similar to Fig. 4.1, the result of Theorem 4.3.4 is visualized in Fig. 4.2. The unioned cases are NP-Complete; while the others are in PTIME.

In the following, we provide several lemmas to prove Theorem 4.3.4.

Lemma 4.3.5 Given a key-preserving full schema mapping, CVP_K is in class NP.

To prove Lemma 4.3.5, two notions are needed: "homomorphism" and "disjunctive backchase", which is similar to the "disjunctive chase" in [28].

Homomorphism Given a schema $\mathbf{R}$ and two instances I_1, I_2 of $\mathbf{R}$, a *homomorphism* $h : I_1 \to I_2$ is a mapping from **Dom** to **Dom**, such that (1) for each constant c occurring in I_1, $h(c) = c$, and (2) for each tuple $(a_1, a_2, ..., a_n)$ of I_1, $(h(a_1), h(a_2), ..., h(a_n))$ is a tuple of I_2.

Disjunctive backchase Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and a relation symbol $R \in \mathbf{T}$, $\Sigma_R \subseteq \Sigma$ is a set of all tgds whose target relation symbols are R. Denote $P^+(\Sigma_R)$ to be the power set of Σ_R without empty set. Denote $\tilde{P}^+(\Sigma_R)$ to be a set of sets of tgds obtained by flipping all the arrow directions in $P^+(\Sigma_R)$.

Example 4.3.6 Consider the schema in Examples 4.2.3 and 4.2.4, together with the equations (iii), (iv), (v), and (vi). We have $\Sigma_F = \{(\text{iii}), (\text{v})\}$ and $P^+(\Sigma_F) = \{\{(\text{iii})\}, \{(\text{v})\}, \{(\text{iii}), (\text{v})\}\}$. Similarly, $\tilde{P}^+(\Sigma_F) = \{\{F(x,z) \to E(x,z) \wedge E(z,y)\}, \{F(x,y) \to D(x) \wedge E(x,y)\}, \{F(x,z) \to E(x,z) \wedge E(z,y), F(x,y) \to D(x) \wedge E(x,y)\}\}$. ∎

Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, an instance I of $\mathbf{S}$, and a tuple τ of a relation symbol R in $\mathbf{T}$, a *disjunctive backchase step* (w.r.t I, τ, and $\mathtt{M}$) is of form $I \xrightarrow{\tau,\ \mathtt{M}} \{I_1, ..., I_n\}$, where for each $i \in [1..n]$, $I_i = I \cup \Delta I_i$, where ΔI_i is a chase result of a set of tgd in the $\tilde{P}^+(\Sigma_R)$.

Example 4.3.7 Continue with Example 4.3.6. Suppose we have a tuple $\tau = (1, 2)$ of E and an instance $I = \varnothing$, then a disjunctive chase step w.r.t. I, τ, and the mapping $\mathtt{M}$

specified by equations (iii), (iv), (v), and (vi) is $\varnothing \xrightarrow{\tau,\ \mathtt{M}} \{(E = \{(1, \perp_1), (\perp_1, 2)\}), (E = \{(1, 2)\}, D = \{(1)\}), (E = \{(1, \perp_1), (\perp_1, 2), (1, 2)\}, D = \{(1)\})\}$. ∎

Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance J of $\mathbf{S}$, a *disjunctive backchase*, denoted by $chase_{\mathtt{M}}^{-1}(J)$, is for each tuple τ in J, to apply a disjunctive backchase step $I^i \xrightarrow{\tau,\ \mathtt{M}} \{I_1^{i+1}, ..., I_m^{i+1}\}$, where $I^0 = \varnothing$ and $I^i \in \{I_1^{i-1}, ..., I_n^{i-1}\}$ for each $i > 0$. The return result of $chase_{\mathtt{M}}^{-1}(J)$ is a set of all the "backchased" source instances (i.e., all the leaf nodes, if consider the disjunctive backchase as an execution tree, whose height is the same as the number of tuples in J).

Lemma 4.3.8 Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and a ground instance J of $\mathbf{T}$, there exists an instance I of $\mathbf{S}$, s.t., $J = chase_{\mathtt{M}}(I)$, if and only if there exists an instance $I' \in chase_{\tilde{\mathtt{M}}}^{-1}(J)$ and a homomorphism h from $chase_{\mathtt{M}}(I')$ to J, s.t., $J = chase_{\mathtt{M}}(h(I'))$.

Proof: Let J and $\mathtt{M}$ be as stated in the lemma. Indeed, the disjunctive backchase produces a set of source instances, in which, zero or more instances can be chased through $\mathtt{M}$ to obtained J. The reason we need to consider such a large number of possible source instances is because if a $\mathtt{M}$ is unioned, i.e., exist at least two tgds in $\mathtt{M}$, s.t., the target relation symbols are the same, then it is unknown which tgds (or both of them) are used to obtain a target tuple. Therefore, the disjunctive chase is to exhaustively enumerate all the possibilities.

Once all the candidate source instances are generated, we chase each of them through $\mathtt{M}$ to see if the chase result is the same as J. In general, null values will be carried over during the backchase and chase; therefore a homomorphism is needed to check if the new chase result can "match" J. ∎

Note that the ground instance assumption in Lemma 4.3.8 can be trivially extended to an arbitrary instance by treating the null values in the given target instance as constants.

Now we finalize the proof for Lemma 4.3.5.

Proof: (Lemma 4.3.5) To determine if a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and an instance J of $\mathbf{T}$, there exists an instance I of $\mathbf{S}$, s.t., $J = \mathit{chase}_{\mathtt{M}}(I)$, it is to guess an instance $I' \in \mathit{chase}_{\tilde{\mathtt{M}}}^{-1}(J)$ and a homomorphism from $\mathit{chase}_{\mathtt{M}}(I')$ to J; then check if $J = \mathit{chase}_{\mathtt{M}}(h(I'))$ and $h(I')$ satisfies the key constraints. ∎

In the following, we prove that CVP_{K} is also NP-hard if the given mapping is unioned.

Lemma 4.3.9 CVP_{K} is NP-hard if the given key-preserving, safe, and full schema mapping is unioined and multi-viewed, but not self-joined.

To prove the hardness of Lemma 4.3.5, we reduce it from the well-known Three Satisfiability Problem [32].

3SAT Problem Given a finite set U of variables and a collection C of clauses over U, s.t., for each $c \in C$, $|c| = 3$, is there a truth assignment for C?

To better illustrate the intuition of the reduction, we walk through a specific example.

Assume the variable set $U = \{a, b, c, d\}$ and $C = \{(a, \bar{b}, c), (\bar{a}, c, d), (b, \bar{c}, \bar{d})\}$. (Or equivalently, the 3SAT Problem is to check if formula $(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee c \vee d) \wedge (b \vee \bar{c} \vee \bar{d})$ has a truth assignment). Now consider the following tgds, where each relation symbol

has key that contains every column (thus safe and key-preserving):

$$
\begin{gathered}
T(x) \to V(x) \qquad F(x) \to V(x) \qquad T(x) \wedge F(x) \to E(x) \\
B_L(x_a, x_b, x_c, x_d) \to B_R(x_a, x_b, x_c, x_d \\
C_L^1(y) \to C_R^1(y) \qquad C_L^2(y) \to C_R^2(y) \qquad C_L^3(y) \to C_R^3(y) \\
T(x_a) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^1(y) \to S(y, x_a, x_b, x_c, x_d) \\
F(x_b) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^1(y) \to S(y, x_a, x_b, x_c, x_d) \\
T(x_c) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^1(y) \to S(y, x_a, x_b, x_c, x_d) \\
F(x_a) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^2(y) \to S(y, x_a, x_b, x_c, x_d) \\
T(x_c) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^2(y) \to S(y, x_a, x_b, x_c, x_d) \\
T(x_d) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^2(y) \to S(y, x_a, x_b, x_c, x_d) \\
T(x_b) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^3(y) \to S(y, x_a, x_b, x_c, x_d) \\
F(x_c) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^3(y) \to S(y, x_a, x_b, x_c, x_d) \\
F(x_d) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^3(y) \to S(y, x_a, x_b, x_c, x_d)
\end{gathered}
\tag{x}
$$

together with the following target instance:

$$
\begin{gathered}
V = \{(a), (b), (c), (d)\} \qquad E = \varnothing \qquad B_R = \{(a, b, c, d)\} \\
C_R^1 = \{(1)\} \qquad C_R^2 = \{(2)\} \qquad C_R^3 = \{(3)\} \\
S = \{(1, a, b, c, d), (2, a, b, c, d), (3, a, b, c, d), \}
\end{gathered}
$$

The above tgds and target instance "encode" the given 3SAT input. For example, $T(x) \to V(x)$, $F(x) \to V(x)$, and $T(x) \wedge F(x) \to E(x)$ denote that a variable should occur in T or F (since V contains all variables) but not both (since E is empty). B_L, B_R, C_L^i, and C_R^i $(i = 1, 2, 3)$ serve as "equality witness" for variables and clauses; for

example, $T(x_a) \wedge B_L(x_a, x_b, x_c, x_d) \wedge C_L^1(y) \rightarrow S(y, x_a, x_b, x_c, x_d)$ denotes that if x_a equals a (implied by $B_L(x_a, x_b, x_c, x_d)$, where B_L and B_R should all be (a, b, c, d)) and a occurs in the first clause $(a, \bar{b}, c)$ (implied by $C_L^1(y)$, where C_L^1 and C_R^1 both should contain tuple (1)) without a negation (i.e., assigned to be true, which is indicated by $T(x_a)$), then the first clause should be true (implied by $S(y, x_a, x_b, x_c, x_d)$, where S contains $(1, a, b, c, d)$).

Proof: (Lemma 4.3.9) Given a set U of variables $\{v_1, v_2, ..., v_m\}$ and a collection C of clauses $\{c_1, c_2, ..., c_n\}$, we construct a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with key constraints, such that (1) $\mathbf{S}$ contains $n+2$ unary relation symbols T, F, C_L^1, C_L^2, ..., C_L^n and a m-ary relation symbol B_L, (2) $\mathbf{T}$ contains $n+2$ unary relation symbols V, E, C_R^1, C_R^2, ..., R_L^n, a m-ary relation symbol B_R, and a $(m+1)$-ary relation symbol S, (3) for each relation symbol R in $\mathbf{S} \cup \mathbf{T}$ with arity of k (where k is some positive integer), R has key $\{1, 2, ..., k\}$, and (4) Σ contains the following set of tgds:

$$\begin{aligned}
&\{T(x) \rightarrow V(x), F(x) \rightarrow V(x), T(x) \wedge F(x) \rightarrow E(x)\} \ \cup \\
&\{B_L(x_1, x_2, ..., x_m) \rightarrow B_R(x_1, x_2, ..., x_m)\} \ \cup \ \{C_L^i(y) \rightarrow C_R^i(y) \mid c_i \in C\} \ \cup \\
&\{T(x_j) \wedge B_L(x_1, .., x_m) \wedge C_L^i(y) \rightarrow S(y, x_1, .., x_m) \mid x_j \text{ is in } c_i \text{ without negation}\} \ \cup \\
&\{F(x_j) \wedge B_L(x_1, .., x_m) \wedge C_L^i(y) \rightarrow S(y, x_1, .., x_m) \mid x_j \text{ is in } c_i \text{ with negation}\}
\end{aligned} \tag{xi}$$

Consider the following instance J of $\mathbf{T}$:

$$V = \{(v_1), (v_2), ..., (v_m)\} \quad E = \varnothing \quad B_R = \{(v_1, ..., v_m)\}$$
$$C_R^i = \{(i)\} \text{ for each } i \in [1..n] \quad S = \textstyle\bigcup_{i=1}^{n} \{(i, v_1, ..., v_m)\}$$

Note that the mapping specified by equation (xi) is safe and key-preserving. Now we prove that 3SAT Problem with input U and C has a truth assignment if and only if there

exists an instance I of $\mathbf{S}$, s.t., $J = chase_{\mathtt{M}}(I)$.

($\Leftarrow$): Suppose there exists an instance I of $\mathbf{S}$, s.t., $J = chase_{\mathtt{M}}(I)$. According to the tgds in Σ, the values in the instances of T and F should be disjoint (because E is empty) and the union of T and F should cover all variables (in the given U of the 3SAT problem). Moreover, based on tgds $B_L(x_1, x_2, ..., x_m) \rightarrow B_R(x_1, x_2, ..., x_m)$ and $C_L^i(y) \rightarrow C_R^i(y)$ ($i = [1..n]$), the instances of B_L and C_L^i in I can only be $\{(v_1, ..., v_m)\}$ and $\{(i)\}$. Therefore, each rule $T(x_j)$ (or $F(x_j)$) $\wedge B_L(x_1, ..., x_m) \wedge C_L^i(y) \rightarrow S(y, x_1, ..., x_m)$ is essentially to encode the truth assignment that if x_j equals variables v_j and x_j occurs in clause c_j without (resp. with) negation, an instance of S should be "validated" as true (i.e., having tuple $(i, v_1, ..., v_m)$). Hence, the 3SAT problem has an assignment by assigning each variable in T to true, and those in F to false.

($\Rightarrow$): This direction is similar to ($\Leftarrow$). Suppose there is a truth assignment to U, s.t., each clause in C is true. We can construct I by assigning all variables in U with true assignment to the instance of T and all others (i.e., with false assignment) to F. For the instances of B_L and C_L^i ($i = [1..n]$), they should be exactly the same as B_R and C_R^i. It is easy to show that $J = chase_{\mathtt{M}}(I)$. ∎

The following Lemma 4.3.10 considers a more restrictive case comparing with Lemma 4.3.9

Lemma 4.3.10 CVP_{K} is NP-hard if the given key-preserving, safe, and full schema mapping is unioined, but not self-joined or multi-viewed.

Proof: The proof technique of Lemma 4.3.10 is similar to the one of Lemma 4.2.8. The technique in Lemma 4.2.8 is to "expand" equation (vii) into equation (viii) and show these two equations are "equivalent". Similarly, we can "expand" equation (xi) into another equation with a single "big" target relation symbol and show the "equivalence".

An example "big" target relation symbol could be $R_{\text{big}}(x_V, x_E, x_1^B, x_2^B, ..., x_m^B, y, x_1^S, x_2^S, ..., x_m^S)$, where columns x_V and x_E represent V and E, $x_1^B, x_2^B, ..., x_m^B$ represent B_R, and $y, x_1^S, x_2^S, ..., x_m^S$ represent S.

The only problem with equation (xi) is that the given instance of E is empty; however, if we assign a column x_E in R_{big} for E, it is impossible to assign values to x_E to represent emptiness. Therefore, we revise T, F, V, and E in equation (xi) to be arity of 2 with keys on the first columns and the corresponding tgds in equation (xi) to be $T(x, y) \rightarrow V(x, y)$, $F(x, y) \rightarrow V(x, y)$, and $T(x, y) \wedge F(x, y) \rightarrow E(x, y)$. The given target instances are $V = \{(v_1, 1), ..., (v_m, 1)\}$, where v_1, ..., v_m are the variables given in the 3SAT Problem and $E = \{(0, 0)\}$. Essentially, the 0 value in E represents "emptiness" as each tuple in the source instances of T and F will never have 0 value on the second column. (Note the other tgds in equation (xi) that involve T and F should be revised correspondingly). ∎

Till this point, Lemmas 4.3.5 and 4.3.10 indicate "half" of Theorem 4.3.4 is correct; i.e., given a key-preserving schema mapping, if the mapping is unioned, CVP_{K} is NP-complete even when the mapping is not self-joined or multi-viewed. Regarding Fig. 4.2, this "half" result essentially is denoted by the four "NP-C" cases on the "is unioned" dimension. In the remainder of this section, we will show that the rest four cases are in PTIME.

Lemma 4.3.11 Given a key-preserving, safe, and full schema mapping, if the mapping is not unioned, CVP_{K} is in PTIME.

To prove Lemma 4.3.11, we first go through an example to see how the case in Lemma 4.3.11 can be solved in PTIME.

Example 4.3.12 Consider the following full schema mapping, where each relation sym-

bol has key {1} (i.e., the first column; hightlighted with underline).

$$E(\underline{x}, y, z) \wedge E(\underline{y}, u, w) \rightarrow S(\underline{x}, y, w) \qquad F(\underline{x}, y) \wedge E(\underline{y}, z, w) \rightarrow T(\underline{x}, y, z)$$

It is straightforward to show that the above mapping is key-preserving and safe.

Suppose the given target instance J is $S^J = \{(1,2,3),(2,3,5)\}$ and $T^J = \{(4,3,6)\}$, which both satisfy the key constraints.

Now we obtain the original source instance I of E and F by chasing J through the reversed mapping (i.e., $S(x,y,w) \rightarrow E(x,y,z) \wedge E(y,u,w)$ and $T(x,y,z) \rightarrow F(x,y) \wedge E(y,z,w)$). The chased results are $E^I = \{(1,2,\perp_1),(2,\perp_2,3),(2,3,\perp_3),(3,\perp_4,5),(3,6,\perp_5)\}$ and $F^I = \{(4,3)\}$, where the first 4 tuples in E^I are from the first tgd above and the last tuple in E^I together with the only one in F^I are from the second.

With the chased source instance I, we again apply chase to obtain the new chased target instance J' through the original schema mapping and have $S^{J'} = \{(1,2,3),(1,2,\perp_3),(2,3,5),(2,3,\perp_5)\}$ and $T^{J'} = \{(4,3,6),(4,3,\perp_4)\}$. Note that in order to make $S^{J'}$ and $T^{J'}$ coincide with S^J and T^J, $\perp_3$ has no choice but 3 since the corresponding key in the same tuple has a constant value 1 which functionally determines the other values in the same tuple. Similar observation can be made for $\perp_5 = 5$ and $\perp_4 = 6$.

With the above homomorphism, the source instance I becomes $E^I = \{(1,2,\perp_1),(2,\perp_2,3),(2,3,3),(3,6,5)\}$ and $F^I = \{(4,3)\}$. Similarly, to enforce E^I to satisfy the key constraints, $\perp_2 = 3$ and $E^I = \{(1,2,\perp_1),(2,3,3),(3,6,5)\}$.

Finally, it can be shown that by replacing $\perp_1$ by an arbitrary value, we have J can be chased from I through the mapping given. ∎

In the following, several claims are given to prove Lemma 4.3.11.

Claim 4.3.13 Given a key-preserving, safe, and full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ that is not unioned, its reversed mapping $\tilde{\mathtt{M}}$, and a ground instance J of $\mathbf{T}$, instance $I = chase_{\tilde{\mathtt{M}}}(J)$ has no null values on key columns for each relation in I.

Proof: Let I, J, $\mathtt{M}$, and $\tilde{\mathtt{M}}$ be as stated in the claim. As a corollary from [33], if a tgd is key-preserving, safe, and full, then the variables occurring on the key positions on the source relation symbols essentially form a super key for the target relation symbol; i.e., each key column of the target relation symbol has a variable that occurs on the source relation symbols. Otherwise, the tgd will not be safe. Hence, each variable that occurs on the key column in some target relation symbol in a tgd in in $\tilde{\mathtt{M}}$ occurs in the source relation symbol for the same tgd. As a result, $I = chase_{\tilde{\mathtt{M}}}(J)$, does not contain a null value on a key column. ∎

The following Claim 4.3.14 is self-explanatory; thus proof is omitted.

Claim 4.3.14 Given a key-preserving, safe, and full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ that is not unioned, and an instance I of $\mathbf{S}$, where nulll values do not occur on key columns, instance $J' = chase_{\mathtt{M}}(I)$ does not have null values on key columns.

Claim 4.3.15 Given a key-preserving, safe, and full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ that is not unioned, its reversed mapping $\tilde{\mathtt{M}}$, and a ground instance J of $\mathbf{T}$ that satisfies the key constraints of $\mathbf{T}$, there is at most one homomorphism h from instance $J' = chase_{\mathtt{M}}(chase_{\tilde{\mathtt{M}}}(J)))$ to J, s.t., $h(J') = J$.

Proof: Let J, J', $\mathtt{M}$, and $\tilde{\mathtt{M}}$ be as stated in the claim. According to Claims 4.3.13 and 4.3.14, J' does not have null values on key columns. Therefore, if there is a homomorphism from J' to J, there can only be at most one since each null value can only be mapped to a constant that is functionally determined by the key. ∎

The following Claim 4.3.16 is a special case of Lemma 4.3.8; thus the proof is omitted.

Claim 4.3.16 Given a key-preserving, safe, and full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ that is not unioned, its reversed mapping $\tilde{\mathtt{M}}$, and a ground instance J of $\mathbf{T}$ that satisfies the key constraints of $\mathbf{T}$, there exists an instance I of $\mathbf{S}$ that satisfies the key constraints of $\mathbf{S}$, s.t., $J = \mathit{chase}_{\mathtt{M}}(I)$ if and only if for instance $I' = \mathit{chase}_{\tilde{\mathtt{M}}}(J)$, there exists a homomorphism h from $\mathit{chase}_{\mathtt{M}}(I')$ to J, where $h(I')$ satisfies the key constraints of $\mathbf{S}$ and $\mathit{chase}_{\mathtt{M}}(h(I')) = J$.

Claim 4.3.16 can be easily generalized to allow arbitrary target instance by treating null values in the given target instance as constants. Note that the if and only if condition in Claim 4.3.16 can be check in polynomial time, therefore Lemma 4.3.11, together with Theorem 4.3.4, is proved.

4.4 Missing Source Relations

The previous two sections assume that the entire source database is unknown. However in practice, when a target database is updated, only a few source tables may be affected and other tables remain unchanged. Therefore, in this section, we relax CVP by assuming that some source tables are known and the goal is to identify the unknown source tables given a tgd mapping and a target database.

Chase Validity Problem with k Missing Source Tables ($\text{CVP}^{\mathbf{M}-k}$) Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, a schema $\mathbf{R} \subseteq \mathbf{S}$ with size of $k > 0$, a database instance $I_{\mathbf{S}\setminus\mathbf{R}}$ of $\mathbf{S} - \mathbf{R}$, and a database instance J of T, does there exist an instance $I_{\mathbf{R}}$ of $\mathbf{R}$ such that $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\mathbf{R}} \cup I_{\mathbf{R}})$?

Note that $\text{CVP}^{\text{M}-k}$ does not require key constraints. When k equals to the size of the source schema, $\text{CVP}^{\text{M}-k}$ becomes CVP.

Theorem 4.4.1 CVP^{M-k} is NP-Complete for $k \geqslant 2$, no matter whether the given full schema mapping is self-joined or not, unioned or not, or/and multi-viewed or not.

Theorem 4.4.1 denotes that all 8 combinations of self-joined, unioned, and multi-viewed will lead to an intractable result (shown in Fig. 4.3).

To prove Theorem 4.4.1, two lemmas (4.4.2 and 4.4.3) are needed.

Lemma 4.4.2 If a full schema mapping is not self-joined, unioned, or multi-viewed, CVP^{M-2} is NP-Complete.

Proof: The lemma can be directly reduced from Set Basis Problem (cp. the proof for Lemma 4.2.7) by rewriting the equation (vii) to be $\hat{B}(s,b) \wedge R(b,c) \wedge D(b) \rightarrow \hat{C}(s,c)$ with the same given instances as the ones in the proof for Lemma 4.2.7. ∎

Lemma 4.4.3 If a full schema mapping is self-joined only, CVP^{M-1} is NP-Complete.

Proof: The lemma can be directly reduced from Boolean Matrix Root Problem [34]: given a $n \times n$ boolean matrix B, does there exist another $n \times n$ boolean matrix A, s.t., $A^2 = AA^T = B$. According to this problem, we can construct a single mapping $D(y) \wedge A(x,y) \wedge A(y,z) \rightarrow B(x,z)$, with the instance of B representing the given matrix B and $D = \{(1),(2),...,(n)\}$. ∎

Lemma 4.4.3 can also be proved by reducing from the Graph 3-Colorable Problem [32].

Proof: (Theorem 4.4.1) (Sketch) We first prove that CVP^{M-k} is NP-hard for $k \geqslant 2$.

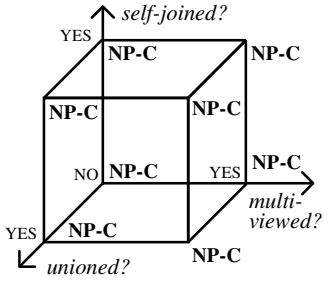

Figure 4.3: $\geqslant 2$ tables missing

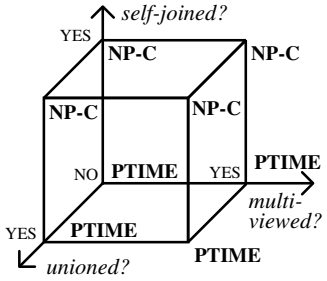

Figure 4.4: One table missing

Suppose the given full schema mapping is unioned only or multi-viewed only, it can be shown that $\text{CVP}^{\text{M}-2}$ is NP-hard by applying the same reduction as the one for CVP from the Set Basis Problem. The reduction essentially only requires two source tables unknown.

Suppose $\text{CVP}^{\text{M}-(k-1)}$ is NP-hard, it is trivial to show that $\text{CVP}^{\text{M}-k}$ is NP-hard by reducing from $\text{CVP}^{\text{M}-(k-1)}$. The technique is to introduce a dummy source and target relation given the input of $\text{CVP}^{\text{M}-(k-1)}$. Together with Lemmas 4.4.2 and 4.4.3, the hardness is proved.

The proof of the membership of NP is similar to the one of Lemma 4.3.5 by conducting a disjunctive backchase, guessing a homomorphism, and verify if the homomorphism is valid. ∎

The following Theorem 4.4.4 indicates that if there is only one relation missing, some intractable cases in Theorem 4.4.1 can be solved in PTIME.

Theorem 4.4.4 $\text{CVP}^{\text{M}-1}$ is NP-Complete if a full mapping is self-joined; otherwise, is in PTIME.

Fig. 4.4 is a visualization of Theorem 4.4.4. Half of Theorem 4.4.4 (i.e., the NP-Complete case) has already been proved in Lemma 4.4.3. In the remainder of this section,

we prove the PTIME case.

To prove Theorem 4.4.1, the notion "transposed mapping" is needed and will be used in the remainder of this section.

Definition: Given a full schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ and a schema $\mathbf{R} \subseteq \mathbf{S}$, a *transposed mapping* of $\mathtt{M}$ wrt $\mathbf{R}$, denoted as $\hat{\mathtt{M}}_{\mathbf{R}}$ is a (not necessarily full) schema mapping $(\mathbf{T} \cup \mathbf{S} - \mathbf{R}, \mathbf{R}, \Sigma')$ where for each rule $R_1(\bar{x}_1) \wedge ... \wedge R_n(\bar{x}_n) \rightarrow S(\bar{y})$ in Σ, there is a corresponding rule $S(\bar{y}) \wedge R_1(\bar{x}_1) \wedge ... \wedge R_m(\bar{x}_m) \rightarrow R_{m+1}(\bar{x}_{m+1}) \wedge ... \wedge R_n(\bar{x}_n)$ in Σ' (where $m < n$), s.t., $R_1, ..., R_m \in \mathbf{S} - \mathbf{R}$ and $R_{m+1}, ..., R_n \in \mathbf{R}$.

The intuition of introducing the transposed mapping from $\mathbf{T} \cup \mathbf{S} - \mathbf{R}$ to $\mathbf{R}$ is to define a set of new rules to generate the unknown source tables $\mathbf{R}$, where the new source is the original target tables $\mathbf{T}$ together with the original source tables $\mathbf{S} - \mathbf{R}$ that are known and the new target is the original source tables $\mathbf{R}$ that are not known.

Example 4.4.5 Consider the following set of mapping:

$$\begin{aligned} A(a,d,c) \wedge B(b,e) \wedge D(d,e) &\rightarrow S(c,e) & A(a,b,c) \wedge E(c,e) &\rightarrow T(b,c,e) \\ A(a,b,c) \wedge F(a,g) &\rightarrow G(a,c,g) & A(a,b,c) \wedge C(c,f) &\rightarrow H(a,c,f) \end{aligned} \tag{xii}$$

Suppose relation A is missing, then the transposed mapping wrt A is:

$$\begin{aligned} S(c,e) \wedge B(b,e) \wedge D(d,e) &\rightarrow A(a,d,c) & T(b,c,e) \wedge E(c,e) &\rightarrow A(a,b,c) \\ G(a,c,g) \wedge F(a,g) &\rightarrow A(a,b,c) & H(a,c,f) \wedge C(c,f) &\rightarrow A(a,b,c) \end{aligned} \tag{xiii}$$

Essentially, equation (xiii) defines a mapping to generate the missing table A. ∎

The following Example 4.4.6 illustrates the idea of how a PTIME algorithm is designed to check if the single missing source relation exists.

Example 4.4.6 Continue with Example 4.4.5. Suppose the missing relation is A; consider the schema mapping and its transposed mapping defined by equations (xii) and (xiii), together with the following instances:

$$B = \{(4,2),(4,3)\} \quad C = \{(2,8)\} \quad D = \{(5,3),(6,3)\} \quad E = \{(2,7)\} \quad F = \{(2,8)\}$$
$$S = \{(2,3)\} \quad T = (6,2,7) \quad G = \{(2,9,8)\} \quad H = \{(2,2,8),(1,2,8)\}$$

With the above instance, we are able to recover a "template" of A instance by chasing through transposed mapping specified in equation (xiii):

$$A^{\perp} = \{(\perp_1,5,2),(\perp_2,6,2),(\perp_3,6,2),(2,\perp_4,9),(2,\perp_5,2),(1,\perp_6,2)\}$$

Each tuple in $A^{\perp}$ serves as a "witness" to justify each tuple in the original target instance. For example, tuple $(\perp_1,5,2)$ in $A^{\perp}$, together with $(4,3)$ in B and $(5,3)$ in D justifies tuple $(2,3)$ in S according to the first tgd rule in equation (xii).

Then for each tuple in $A^{\perp}$, together with the original known relations (i.e., B, C, D, E, and F), we chase through the original mapping (i.e., equation (xii)) to find a homomorphism to the original given target instance. For example, chasing tuple $(\perp_1,5,2)$ for A with B, C, D, E, and F, we have a chased target instance $S^{\perp} = \{(2,3)\}$, $T^{\perp} = \{(5,2,7)\}$, $G^{\perp} = \varnothing$, and $H^{\perp} = \{(\perp_1,2,3)\}$, where there is no homomophism from the newly chased target to the original target, therefore tuple $(\perp_1,5,2)$ is not a "valid" witness. However for tuple $(\perp_2,6,2)$ in A, we will end up with an instance $S^{\perp} = \{(2,3)\}$, $T^{\perp} = \{(6,2,7)\}$, $G^{\perp} = \varnothing$, and $H^{\perp} = \{(\perp_2,2,3)\}$, where by picking $\perp_2$ to be either 1 or 2, there is a homomorphism to the original target.

Once all the homomorphisms are found for each chased result, we need to verify if

each homomorphism is "valid" by replacing the null values in the corresponding tuple by the mapped constants. For example, if the homomorphism defines $\perp_2 = 2$, the $A-$tuple becomes $(\perp_2 = 2, 6, 2)$; if a chase is performed based on $A = \{(\perp_2 = 2, 6, 2)\}$, together with B, C, D, E, and F through equation (xii)), we have $G^{\perp} = \{(2, 2, 8)\} \neq \{(2, 9, 8)\}$, which is not a "valid" witness. On the contrary, when $\perp_2 = 1$, we can have $S^{\perp} = \{(2, 3)\}$, $T^{\perp} = \{(6, 2, 7)\}$, $G^{\perp} = \varnothing$, and $H^{\perp} = \{(1, 2, 3)\}$ that is a subset of the original target.

After iterate through all the tuples in $A^{\perp}$ and verify all the candidate homomorphisms, only two tuples are "valid" in this case: $A^{\perp} = \{(\perp_2 = 1, 6, 2), (2, \perp_4, 9)\}$, where $\perp_4$ is allowed to map to an arbitrary constant that is not in the active domain of the given instance. Unfortunately, tuple $(2, 2, 8)$ in H cannot be justified by either of the tuples; therefore, the instance of A does ont exist. ∎

Let I_1 and I_2 be two database instances of the same schema, and h_1 and h_2 both be the homomorphisms from I_1 to I_2, h_1 and h_2 are *equivalent* (wrt pair (I_1, I_2)) if $h_1(I_1) = h_1(I_2)$. A *equivalent homomorphic class* $[h]$ from I_1 to I_2 is a set of homomorphisms s.t., for each $h_1, h_2 \in [h]$, h_1 and h_2 are equivalent (wrt pair (I_1, I_2)).

Lemma 4.4.7 Let I_1 and I_2 be two database instances of the same schema, the number of the equivalent homomorphic classes from I_1 to I_2 is polynomially many wrt the size of I_1 and I_2 if the number of null values in I_1 is bounded by a constant.

Lemma 4.4.7 is straightforward; thus the proof is omitted.

The detailed steps to check whether the missing source table exists is described by Algorithm 1, where line 3 is to construct the "witness table" with null values, line 5 is to re-construct the target from each witness tuple, and lines 7 - 10 are to check if the re-constructed target has a "valid" homomorphism to the original target. The complexity of Algorithm 1 is polynomial wrt to the size of the given instances. (Note that for line

Algorithm 1 Deciding $\text{CVP}^{\text{M}-1}$ given a non-self-joined full mapping

Input: full mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, a relation symbol $R \in \mathbf{S}$, instance $I_{\mathbf{S}\setminus\{R\}}$ of $\mathbf{S} - \{R\}$, and instance J of $\mathbf{T}$

Output: true or false (i.e., if exists an instance $I_{\{R\}}$ or $\{R\}$, s.t., $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$)

1: Let $\hat{\mathtt{M}}_{\{R\}}$ be the transposed mapping of $\mathtt{M}$
2: Set $I_{\{R\}} := \varnothing$
3: $I^{\perp}_{\{R\}} := \mathit{chase}_{\hat{\mathtt{M}}_{\{R\}}}(I_{\mathbf{S}\setminus\{R\}} \cup J)$
4: **for each** tuple $\tau^{\perp}_{\{R\}} \in I^{\perp}_{\{R\}}$ **do**
5: $\quad J^{\perp} := \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{\tau^{\perp}_{\{R\}}\})$
6: $\quad$ **for each** equivalent homomorphic class $[h]$ from $J^{\perp}$ to J **do**
7: $\quad\quad J_0 := \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{h(\tau^{\perp}_{\{R\}})\})$, where $h \in [h]$
8: $\quad\quad$ **if** $J_0 \subseteq J$ **then**
9: $\quad\quad\quad I_{\{R\}} := I_{\{R\}} \cup \{h(\tau^{\perp}_{\{R\}})\}$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end for**
13: **return** true iff $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$;

6, since the number of null values in a tuple is bounded, only polynomially many times of iterations will occur according to Lemma 4.4.7).

Lemma 4.4.8 Given a full mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, a relation symbol $R \in \mathbf{S}$, instance $I_{\mathbf{S}\setminus\{R\}}$ of $\mathbf{S} - \{R\}$, and instance J of $\mathbf{T}$, Algorithm 1 is sound and complete determining whether there exists an instance $I_{\{R\}}$ of R, s.t., $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$

Proof: Let $\mathtt{M}$, $\mathbf{S}$, $\mathbf{T}$, Σ, R, $I_{\mathbf{S}\setminus\{R\}}$, and J be as stated in the lemma.

(Soundness) Straightforward based on line 13 of Algorithm 1.

(Completeness) Suppose there exists an instance $I^{\text{c}}_{\{R\}}$ of $\{R\}$ s.t., $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I^{\text{c}}_{\{R\}})$; while Algorithm 1 returns false given the same input. Let $I_{\{R\}}$ be the instance of $\{R\}$ constructed at the end of Algorithm 1.

Let $\tau^{\text{c}}_{\{R\}}$ be a tuple in $I^{\text{c}}_{\{R\}}$ and J^{c} be $\mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{\tau^{\text{c}}_{\{R\}}\})$. W.l.o.g, suppose J^{c} is not empty. For each tuple τ^{c}_{J} in J^{c}, suppose τ^{c}_{J} is generated by rule $R, R_1, ..., R_n \rightarrow T$, where

$R_1, .., R_n \in \mathbf{S}$ and $T \in \mathbf{T}$ (note that the variables are ignored as the context is clear) with tuples $\tau^{\mathrm{c}}_{\{R\}}$ of R, $\tau^{\mathrm{c}}_{\{R_1\}}$ of R_1, $...\tau^{\mathrm{c}}_{\{R_n\}}$ of R_n. Therefore, two facts can be concluded: (1) $T, R_1, ..., R_n \rightarrow R$ is a rule in the transposed mapping $\hat{\mathtt{M}}_{\{R\}}$ of $\mathtt{M}$ (line 1 of Algorithm 1) and (2) $\tau^{\mathrm{c}}_{\{R_1\}}, ...\tau^{\mathrm{c}}_{\{R_n\}}, \tau^{\mathrm{c}}_{J} \in I_{\mathbf{S}\setminus\{R\}} \cup J$. Let $\tau^{\perp}_{\{R\}}$ be the tuple in $I^{\perp}_{\{R\}} := \mathit{chase}_{\hat{\mathtt{M}}_{\{R\}}}(I_{\mathbf{S}\setminus\{R\}} \cup J)$ (line 3 of Algorithm 1) generated based on tuples $\tau^{\mathrm{c}}_{\{R_1\}}, ...\tau^{\mathrm{c}}_{\{R_n\}}, \tau^{\mathrm{c}}_{J}$ and rule $T, R_1, ..., R_n \rightarrow R$. It is easy to show that there is a homomorphism h from $\tau^{\perp}_{\{R\}}$ to $\tau^{\mathrm{c}}_{\{R\}}$, s.t., $h(\tau^{\perp}_{\{R\}}) = \tau^{\mathrm{c}}_{\{R\}}$. As a result, $\mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{h(\tau^{\perp}_{\{R\}})\}) = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{\tau^{\mathrm{c}}_{\{R\}}\}) = J^{\mathrm{c}} \subseteq J$; therefore, $h(\tau^{\perp}_{\{R\}}) \in I_{\{R\}}$ (cp. lines 7 - 10).

Accordingly, for each tuple $\tau^{\mathrm{c}}_{\{R\}}$ in $I^{\mathrm{c}}_{\{R\}}$, if $J^{\mathrm{c}} = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup \{\tau^{\mathrm{c}}_{\{R\}}\})$ is not empty, there exists a tuple $\tau^{\perp}_{\{R\}}$ in $I^{\perp}_{\{R\}}$ and a homomorphism h from $\tau^{\perp}_{\{R\}}$ to $\tau^{\mathrm{c}}_{\{R\}}$, s.t.,$h(\tau^{\perp}_{\{R\}}) = \tau^{\mathrm{c}}_{\{R\}}$. Hence, $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I^{\mathrm{c}}_{\{R\}}) \subseteq \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$. Moreover, based on lines 7 - 10, $J \supseteq \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$. Therefore, we have $J = \mathit{chase}_{\mathtt{M}}(I_{\mathbf{S}\setminus\{R\}} \cup I_{\{R\}})$, a contradiction. ∎

4.5 Missing Source Relations with Key Constraints

Recall that Proposition 4.3.1 shows a trivial results that with or without key constraints, CVP and $\mathrm{CVP}_{\mathrm{K}}$ make no difference in terms of complexity results (both can be visualized by Fig. 4.1). However, it is not the same case for $\mathrm{CVP}^{\mathrm{M}-k}$. Essentially with key constraints, all the PTIME cases in $\mathrm{CVP}^{\mathrm{M}-1}$ will interestingly become NP-Complete.

Chase Validity Problem with k Missing Source Tables and Keys ($\mathbf{CVP}^{\mathbf{M}-k}_{\mathbf{k}}$)
Given a full and safe schema mapping $\mathtt{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with key constraints, a schema $\mathbf{R} \subseteq \mathbf{S}$ with size of $k > 0$, a database instance $I_{\mathbf{S}\setminus\mathbf{R}}$ of $\mathbf{S} - \mathbf{R}$ that satisfies the key constraints, and a database instance J of T that satisfies the key constraints, does there exist an instance $I_{\mathbf{R}}$ of $\mathbf{R}$ that satisfies the key constraints such that

$J = chase_{\mathtt{M}}(I_{\mathbf{S}\setminus\mathbf{R}} \cup I_{\mathbf{R}})$?

Theorem 4.5.1 $\text{CVP}_{\text{K}}^{\text{M}-k}$ is NP-Complete for $k > 0$, no matter whether the given full schema mapping with key constraints is self-joined or not, unioned or not, or/and multi-viewed or not.

Theorem 4.5.1 denotes that if key constraint is incorporated, all 8 combinations of self-joined, unioned, and multi-viewed will lead to an intractable result (shown in Fig. 4.3), even when there is only one source table missing.

Let k be as stated in Theorem 4.5.1. When $k \geqslant 2$, Theorem 4.5.1 is a corollary of Theorem 4.4.1 and its proof resembles the one for Proposition 4.3.1; thus omitted. Therefore, in the remainder of this section, we only consider the case when $k = 1$. In details, we consider the following Lemma 4.5.2

Lemma 4.5.2 Given a full schema mapping with key constraints that is not self-joined, unioned, or multi-viewed, $\text{CVP}_{\text{K}}^{\text{M}-1}$ is NP-Complete if the number of the non-prime attributes (i.e., the attributes that are not part of a key) in the schema of the only missing source relation is no less than 1.

The upper bound of Lemma 4.5.2 can be proved with the same idea of applying "disjunctive backchase" and "chase", together with guessing a homomorphism and verify if the guess is correct. Therefore, we omit the proof for the upper bound and only discuss the proof of hardness. We reduce from the Domatic Number Problem [32].

Given a graph $G = (V, E)$, a *dominating set* of G is a set of vertices $V' \subseteq V$, where for each $u \in V - V'$, there exists $v \in V'$, s.t., $(u, v) \in E$.

Domatic Number Problem Given a graph $G = (V, E)$ and a positive integer $K \leqslant |V|$, is there a partition of V into K disjoint set V_1, V_2, ..., V_K, s.t., for each $i \in [1..K]$, V_i is a dominating set?

Specifically, The Domatic Number Problem is NP-Complete for a fixed given integer $K \geqslant 3$ and in PTIME when $K < 3$ [32].

The following Lemma 4.5.3 is introduced for a cleaner representation of the reduction.

Lemma 4.5.3 The Domatic Number Problem has a solution (i.e., a partition) with input (V, E) and K iff it has a solution with input $(V, E \cup \bigcup_{v \in V}\{(v, v)\})$ and K.

Proof: Let V, E, and K be as stated in the lemma.

($\Rightarrow$) Suppose there exists a partition for $V = V_1 \cup V_2 \cup ... \cup V_K$ given inputs (V, E) and K. According to the definition of dominating set, for each $i \in [1..K]$ and each $x \in V - V_i$, there exists $y \in V_i$, s.t., $(x, y) \in E \subseteq E \cup \bigcup_{v \in V}\{(v, v)\}$. By nature, this direction holds.

($\Leftarrow$) Suppose there exists a partition for $V = V_1 \cup V_2 \cup ... \cup V_K$ given inputs $(V, E \cup \bigcup_{v \in V}\{(v, v)\})$ and K. Then, for each $i \in [1..K]$ and each $x \in V - V_i$, there exists $y \in V_i$, s.t., $(x, y) \in E \cup \bigcup_{v \in V}\{(v, v)\}$. Since $(V - V_i) \cap V_i = \varnothing$, we have $x \neq y$. Therefore, $(x, y) \in E$. This direction also holds. ∎

Proof: (NP-hardness of Lemma 4.5.2) Let V, E, and K be as stated in the Domatic Number Problem. The goal is to create a single tgd rule without self-join to encode the Domatic Number Problem. Now consider (1) a binary source relation symbols $\hat{E}(x, y)$ with key $\{1, 2\}$ to denote the edge set E together with all the self loops (i.e., $(\bigcup_{v \in V}\{(v, v)\})$), a unary source relation symbol $D(d)$ with key $\{1\}$ to encode the "IDs" of the K dominating sets, and a binary source relation symbol $R(x, d)$ with key $\{1\}$ to denote that a vertex x is in which dominating set d, (2) a binary target relation symbol

$C(y, d)$ with key $\{1, 2\}$ to denote that each vertex y should be "covered" by a dominating set d, and (3) the following single tgd:

$$R(x, d) \wedge \hat{E}(x, y) \wedge D(d) \rightarrow C(y, d) \tag{xiv}$$

Suppose V contains n vertices: v_1, v_2, ..., and v_n. The instances of $\hat{E}$, D, and C are defined as follows:

$$\hat{E} = E \cup \bigcup_{v \in V} \{(v, v)\} \qquad D = \{(1), (2), ..., (K)\}$$

$$V = \{(v_1, 1), (v_1, 2), ..., (v_1, K), \quad (v_2, 1), (v_2, 2), ..., (v_2, K), \quad ..., \quad (v_n, 1), (v_n, 2), ..., (v_n, K)\}$$

It can be shown that the Domatic Number Problem has a solution if and only if the mapping specified by equation (xiv) with the instances $\hat{E}$, D, and C given above has a source relation R. And the relation R itself forms a partition. The reason is that (1) each tuple in D serves as an ID for a unique dominating set and there are exactly K different dominating sets, (2) due to the key constraint in R, each vertex in V can only occur in a single dominating set (identified by d in R), and (3) for $\hat{E}(x, y)$, if vertex y is adjacent to vertex x through an edge in E or a self-loop in $\bigcup_{v \in V} \{(v, v)\}$, then y, according to Lemma 4.5.3, should be "justified" by the corresponding dominating set, which is represented by C. Note that each vertex in V should be "justified" by each dominating set; since there are K dominating sets and n vertices in V, the size of relation V is $K \times n$.

Since the Domatic Number Problem is NP-Complete, together with that equation (xiv) is not multi-viewed, unioned, or self-joined, Lemma 4.5.2 holds. ∎

4.6 Summary

Tuple generating dependency (tgd) is a widely used mapping language in data exchange area. This chapter studies the problem of deciding the existence of source database given a full tgd mapping and a target database, which is essentially equivalent to deciding the updatability. An arbitrary mapping specification will unfortunately lead to intractable results in most cases (Fig. 4.1), even with key constraint. However, with the property of key-perserving, a mapping that is not unioned can then be resolved in polynomial time (Fig. 4.2). Moreover, we also study the cases where not the entire source database (with or without key constraints) is known. Surprisingly, if two source tables or one source table with key constraints are missing, the problem can be intractable even the given rule is not self-joined, unioned, or multi-viewed (Fig. 4.3). However, if there is only one table missing and key constraint is not enforced, the problem also becomes tractable when the mapping is not self-joined (Fig. 4.4).

Chapter 5

Universal Artifacts

In most BPM systems, the data for process execution is scattered across databases for enterprise, auxiliary local data stores within the BPM systems, and even file systems (e.g., specification of process models). The interleaving nature of data management and BP execution and the lack of a coherent conceptual data model for all data needed for execution make it hard for (1) providing BPaaS (2) effectively support collaboration between business processes, due to an enormous effort required on maintaining both the engines as well as the data for the client applications. In particular, different modeling languages and different BPM systems make process interoperation one of the toughest challenges. In this chapter we formulate a concept of a "universal artifact", which extends artifact-centric models by capturing *all needed data* for a process instance throughout its execution. A framework called SeGA based on universal artifacts is developed to support separation of data and BP execution, a key principle for BPM systems. We demonstrate that SeGA is versatile enough to fully facilitate not only executions of individual processes (to support BPaaS), but also all four collaboration models discussed in the chapter. Moreover, SeGA reduces the complexity in runtime management including runtime querying, constraints enforcement, and dynamic modification upon collaboration

across possibly different BPM systems.

This chapter is organized as follows. Section 5.1 introduces and motivates the need for separating data and execution in order to support BPaaS and collaboration. Section 5.2 defines universal artifacts, and the mappings to/from GSM/EZ-Flow. Section 5.3 outlines the SeGA framework, support for BPaaS, a new conceptual architecture for BPM systems, and reports technical details of the SeGA prototype. Section 5.4 presents a classification of collaboration models and illustrates SeGA support for these models. Section 5.5 gives details on runtime queries, constraints enforcement, and dynamic modification through SeGA, and finally Section 5.6 summarizes the chapter.

5.1 Independence of Data and Execution

A typical BPM system [35] manages process definitions, BP executions, tasks and worklists, resources, etc. through keeping track of the data about them in one or more databases within the BPM system (Fig. 5.4 of [35]), while the actual application data is typically managed in enterprise databases (Fig. 5.5 of [35]). This architecture has been used in many BPM systems such as YAWL [36], jBPM, and JTang [37]. In this chapter, we argue that this traditional architecture must be revised to meet two new challenges in BPM, and develop a new approach based on a technique to free BPM systems from managing local data.

The development of BPM systems typically requires application knowledge and software development experience. The development team does not only formulate concrete BP models, identify data and other resources including human, but also decides on computing hardware and software. During operation, in addition to routine maintenance, every BPM system is required to change in order to adapt to the changes in the environment, regulations and policies, market competitions, etc. Changes are hard technically

and cost wise to many organizations. For example, soon after installing its BPM system, the Housing Management Bureau in city of Hangzhou, China decided to design another system due to the changed policies, environment, and requirements [16]. Such incidents prompted the State Council of China[1] to urge provincial and lower governments to use/purchase more services available in the market to streamline administration, an essential aspect of this call is to shift towards the "Business-Process-as-a-Service" (BPaaS) paradigm.

Cost effective BPaaS is challenging to achieve. Multi-tenancy for BPM systems is an obvious option for effective BPaaS, but is technically hard to realize. A primary reason is that the existing BP design methodologies lack coherent plans for data design. BP execution needs at least the following five types of data: (i) BP model specification, (ii) business data for the process logic, (iii) execution states (and histories), (iv) correlations among BP instances, and (v) resources and their states (e.g, room reserved). Without coherent data design, current BPM systems handle and manage data in ad hoc manners, data for BP execution is scattered across databases, auxiliary data stores managed by the BPM systems [35], and even in files (e.g., BP schemas). It is important to note that artifact-centric BPM systems are similar since their BP models [12, 15, 16] only focus on data of type (ii) but are agnostic of types (iii) to (v).

According to Gartner, BP improvement is the top business strategy of CIOs in enterprises nowadays [38]. Being able to query execution status, gather all traces of tasks, and find correlations of instances is a key element for BP improvement. Consider a permit approval process in a housing management department. During the process execution, staff in the department may want to know: the number of applications that have been lodged since the beginning of the year, the peak time of the application lodging, the applications that have passed "Preliminary Decision", and the applications that did not

[1] `http://www.gov.cn/zwgk/2012-07/20/content_2187242.htm`

follow the defined process. Such information can be used in key performance indicators (KPIs) for BP improvement. Currently the data required to answer these queries are scattered in process logs, data stores, process models, and even execution engines. Data/process warehousing techniques can be applied to extract, transform, and load up (ETL) the data, OLAP tools are then used for process analysis.

Process data warehousing presents some interesting challenges: (1) since the data is tightly coupled with the process model and its execution engine, it is hard to provide generic solutions for warehousing process data for different BPs; (2) data warehousing approach is not efficient for runtime execution monitoring and analysis; (3) process warehousing gets the data but misses the process information. As a result, when process models change, ETL mechanisms for the warehouse often need change as well.

In today's economic market, different BPs need to engage with each other to achieve competitiveness. Enabling collaboration between different BPs continues to pose a fundamental challenge, i.e., ensuring partners' BPs to collaborate with each other under the guidance of business rules and policies to achieve an agreed business goal under any circumstances. A BPM system is typically used for internal BP management in a business unit. Such systems are inadequate for business collaboration that involves independently executing processes with different BP models. Therefore, interoperation between BPM systems is in a huge demanded, and an extremely hard problem.

BP interoperation needs to address two fundamental issues: (1) different model transformation, and (2) runtime BP status and behavior analysis. The former can smooth the communication, while the latter is critical for execution analysis, monitoring and management. Web service standards such as WSDL, BPEL, WSCDL have provided basic interoperation support specified in terms of flow of activities, messages to be exchanged, roles and relationships. But they do not provide a satisfactory support in runtime analysis, monitoring and process change.

A key observation arising from these challenges is that, in spite of significant recent progress in process modeling and enactment, there is a lack of integrated conceptual models and support tools that can capture a sufficient semantics of BPs for runtime execution queries, business analysis, and process improvement.

A fundamental principle needed to support BPaaS and BP collaboration is the *independence of data management and execution management*. The principle entails that a BP execution engine should be free of managing *any* data while the manager of data needed for BP executions should not interfere with decisions on BP execution. A technical challenge here is to develop a new generation of BPM systems that adhere to this principle. In [39], the authors studied how data auditing can be done for BPaaS, where data and execution management are interleaved. We observe that this data auditing problem [39] can be easily solved if data and execution are independently managed.

To address these challenges, in this chapter we introduce a new concept of a "universal artifact" to conceptualize BP execution instances. Intuitively, a universal artifact extends the information model and lifecycle model in a business artifact [12] with (i) the process model specification conformed by the instance to serve as its private copy of the "prescription" for its execution, and (ii) runtime status and dependency information to serve as the context. A novelty here is that a universal artifact captures sufficiently detailed semantics for a BP throughout its execution to support runtime monitoring, analysis, and management. Universal artifacts have two folds. First, they provide a uniform conceptual framework for describing BP schema as well as instance level information for execution. Second, they standardize execution mechanisms and facilitate runtime execution management functions (monitoring, querying, etc.)

We believe a new architecture for BPM systems is needed to fully embrace the independence principle. As a first step, in this chapter we develop a Self-Guided Artifacts

(SeGA) framework to show that existing systems can be "wrapped" around and "mediated" to achieve execution independence. We focus on two representative artifact BP modeling languages: GSM [15] of a declarative flavor, and EZ-Flow [16] as a procedural variant, and show how we use universal artifacts to package and elevate BPs to the conceptual level and "strip" the two engines completely to their "bare bones" (i.e., dataless). By "bare bone" we mean an engine and its system maintains *no* persistent data of each of its running process instance concerning its model, current status, data needed, and status. At runtime, the engine will be *supplied* with *all* necessary data when it needs to take an action, and upon completion the engine again is stripped of all data about the instance. We show that universal artifacts indeed make BP engines and BPM systems free of data management.

This chapter makes the following technical contributions:

1. We formulate the notion of a *universal artifact*, and define mappings between universal artifacts and GSM/EZ-Flow artifact instants (snapshots), i.e., translations between universal artifacts and Barcelona/EZ-Flow.

2. A framework SeGA based on universal artifacts is developed. This framework supports separation of data and BP execution for the two BPM systems for GSM/EZ-Flow. A prototype for SeGA is designed that works with the two systems as expected. An immediate advantage is that SeGA supports BPaaS.

3. We provide a new classification of collaborative BP models based on control and data dimensions. We further demonstrate that the SeGA framework is capable to fully facilitate all BP collaboration models.

4. Finally, we discuss some technical details of how SeGA aids in runtime management including runtime querying, constraints enforcement, and dynamic modification for

(collaborative) BP executions possibly across different BPM systems.

In the remainder of this section, we provide specific examples to unveil two important deficiencies of current BPM systems.

Cloud computing has undoubtedly fuelled the desire to provide BP execution as service or BPaaS. Consider real estate property management in China. There are roughly 10 to 50 Housing Management Bureaus (HMBs) in each of 30 provinces for managing titles, permits, licenses etc. Every HMB currently runs and maintains its own BPM system. For example, the BPM system for the HMB in a large city Hangzhou handles about 300,000 cases annually (with about 500 BP models). BPaaS could potentially bring huge savings to HMBs in managing and maintaining BPM systems and is a great business opportunity in the software market in general.

Virtualization (i.e., VMs) is a key technology for cloud computing that frees clients from owning and maintaining computing hardware and operating systems. In Fig. 5.1, a service provider uses VMs to run BPM systems as services for many HMBs. In Hangzhou, its HMB manages its business data in the enterprise database; the service provider can then run and manage the BPM system, including the data store "Local 1" containing data specific to Hangzhou HMB's BP execution. Current BPM systems store and manage data related to running processes locally in one or more databases as shown in Figure 5.4 of [35]. For a small city Yiwu, the situation is similar except that the provider also manages Yiwu's enterprise data Enterprise Data Store 2 besides its BP execution specific data in Local 2. BPM systems are semantically rich, each BP engine only suits in its local context, its local data store is a main part of the context. As a result, one BP engine cannot be used to serve multiple HMBs. Thus each HMB's BP engine needs to be managed individually, the total effort of maintenance of all BPM systems for HMB clients is not reduced by much but merely shifted to the service provider. For example,

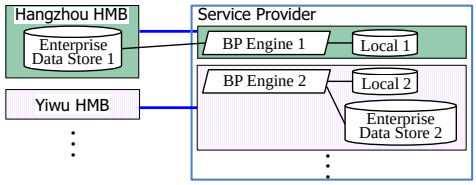

Figure 5.1: Running Clients' BP Engines

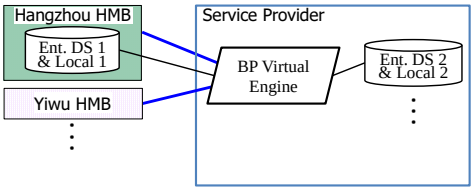

Figure 5.2: Shared BP Virtual Engine

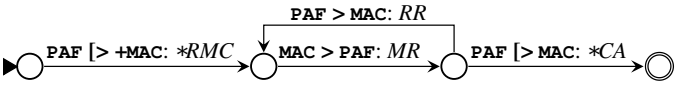

Figure 5.3: Interactions between EAF and MSC

when the core execution engine is to be upgraded, each installation must be upgraded individually in a seemingly repetitive manner.

Fig. 5.2 shows a desirable situation. In this case, only one BP virtual engine is running, each HMB's enterprise business data and engine-specific local data are packaged and stored in an extended data store and maintained either by the client (e.g., in Hangzhou's case) or by the service provider (e.g., in Yiwu's case). Both the data and process definition are provided to the virtual engine when it needs to schedule tasks; upon completion, all data is again packaged and stored accordingly for the client. This is far more efficient and scalable as the number of clients grows.

Achieving Fig. 5.2 turns out to be technically challenging. A primary reason is that most BPM software systems today interleave process execution and data management [35], moreover, some data are collected, stored, and managed without a conceptual data model. In order to understand how to separate data management from BP execution, we present a concrete example below, which is also used to illustrate some difficulties in runtime execution management and behavior analysis of collaborative BPs.

Example 5.1.1 Consider a BP model in Hangzhou HMB (HHMB) concerning approval

for "Early-sell permits" submitted by developers to allow some apartments in the buildings under construction to be put on the market. Permit approval involves at least two collaborating BPs carried out by different departments (may use different BP engines). The primary BP "Early-sell Approval Flow" (`EAF`) accepts applications from developers, performs reviews in several aspects, processes fee payment, and issues approval certificates. One aspect of the review concerns reserved space for building maintenance functions (total area, accessibility, etc.) and is done by the other BP "Maintenance Space Check" (`MSC`). An `EAF` instance launches a `MSC` instance for all apartments in the `EAF` instance and located in the same building. If multiple buildings are involved in the `EAF` instance, one `MSC` instance for each building will be launched.

Fig. 5.3 shows interactions between `EAF` and `MSC` instances. An `EAF` instance initiates new `MSC` instances for maintenance check on apartments through sending multiple *requests for maintenance check* (*RMC*), `MSC` instances may send *maintenance reports* (*MR*) back to `EAF`, and `EAF` may seek an additional *revised request* (*RR*) or decide that it has enough information for a decision and terminate all `MSC` instances with *complete and archive* (*CA*) events to all correlated `MSC` instances. In Fig. 5.3, edge labels specify event details: "$>$" indicates sender and receiver(s), "$*$" stands for multiple events, "$+$" for creation of new `MSC` instances, and "[" for *all* correlated `MSC` instances. ∎

Clearly, `EAF` and `MSC` BPs must collaborate in successful execution. They may run in difference BPM systems, and even use different modeling languages. Providing effective runtime support for such collaboration is difficult. For example, to find all `MSC` instances with at least one apartment failing the check, a naive approach is to hand-code the query directly against the local data stores of the BPM system for `MSC`. Unfortunately one can't do much better. As another example, one may wish to find all `EAF` instances that are not finished but at least one correlated `MSC` instances already completed. This query needs to

develop ad hoc code at both BPM systems, run them and then join the results together. Again, a conceptual data model could easily permit general purpose query evaluation and avoid such ad hoc development. A similar functionality is to monitor at runtime the executions in order to detect violations of choreography constraints (that usually reflects policies and regulations).

During the execution of an `EAF` instance, there are at least five kinds of data involved: (1) the specification of `EAF` model, (2) the business data about the applicant, the apartments, etc., (3) the current execution status, e.g., the initial review of the applicant is completed and two `MSC` instances are initiated, (4) correlation information of the `EAF` and two `MSC` instances, and (5) the building records (owned by Hangzhou's Land Management Bureau) have been checked out for possible update by the `EAF` instance (an approved apartment will be marked on the building records). Among the above types of data, only business data (the 2nd kind) is managed in the HHMB enterprise database, while all others are stored within the HHMB's BPM system. If this BPM system is also to manage executions of BPs from other HMBs, problems will rise since these data (1st, 3rd-5th kinds) from all HMBs are mixed together in the BPM system. HHMB uses a proprietary BPM software but the situation is similar for YAWL and jBPM; the conclusion easily applies to YAWL and jBPM.

An overhaul of storage and management of data of kinds (1), (3) through (5) seems necessary in order to support multi-tenancy and collaboration. In this chapter, we formulate a concept "universal artifact" to cleanly separate all types of data from the execution management of a BPM system. Based on universal artifacts, a framework called "SeGA" (Self-Guided Artifacts) was developed, SeGA allows a single BPM system to serve BP executions from multiple clients and querying over execution at runtime.

5.2 Universal Artifacts

Our goal is to develop techniques for separating data from execution in order to enable business processes as a service and to support collaboration. To this end, we formulate a key notion of a "universal artifact", which is an data object that packages everything needed for a BP engine to perform individual steps. A universal artifact provides a uniform structure to record all necessary data needed for execution, including (***i***) BP schemas, (***ii***) business data, and (***iii***) runtime status and dependencies. By elevating and "wrapping" (i)-(iii) into universal artifacts and detaching them from the underlying BP engines, no local data will be maintained by these BP engines, contrasting to the traditional architecture [35]. This technique of making the engines "data-less" (and thus stateless) is crucial to support BPaaS and collaboration.

In principle, the elevation idea can be applied to all workflow engines. Traditional control-flow-centric BP models lack conceptual modeling for data of types (ii) and (iii), and would require more efforts in finding out how the underlying engines store these data. Based on the data organization, data of types (ii) and (iii) can be extracted. Artifact-centric models [2] conceptually model data of types (i) and (ii). For these models, data elevation can focus on type (iii).

GSM [15] and EZ-Flow [16] incorporate business data/documents and processing "instructions" or lifecycle specification into (business) artifacts. A common feature in them is that type (ii) data is represented as a set of data attributes. However, EZ-Flow specifies lifecycle using graphs, while GSM uses a set of rules and conditions on data to declaratively define the processing sequences and resolves execution ordering at runtime. In this section we briefly review the two models based on Example 5.1.1, and then illustrate how to wrap them into universal artifacts.

Key definitions of GSM and EZ-Flow are reviewed in Sections 5.2.1 and 5.2.2, resp.

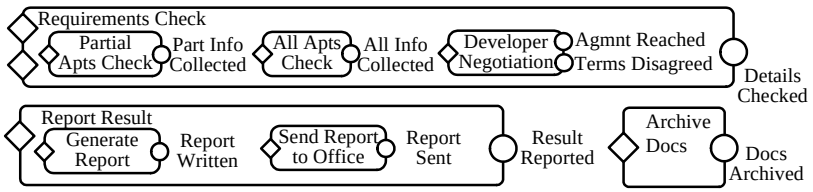

Figure 5.4: A Guard-Stage-Milestone (GSM) Artifact Lifecycle Model of `MSC`

Universal artifacts and their mappings from/to artifacts in GSM and EZ-Flow are provided in Section 5.2.3.

5.2.1 GSM and Barcelona

An artifact stores all business data related to the BP using *attribute*-value pairs. An *event type* is of form $E_{\text{name}}(\sigma_1, ..., \sigma_n)$ where E_{name} is the name for the type, $\sigma_1, ..., \sigma_n$ is a sequence of distinct attributes, and $\sigma_1 =$ "ID", the special attribute to hold an artifact identifier (that uniquely identifies each running artifact instance). An *event* of an event type $E_{\text{name}}(\sigma_1, ..., \sigma_n)$ is of form $E_{\text{name}}(\sigma_1{:}\, c_1, ..., \sigma_n{:}\, c_n)$ where for each $i \in [1..n]$, c_i is a value for attribute σ_i. An event can be *incoming* (received) or *outgoing* (sent).

We now briefly review GSM [40] with the following example.

Example 5.2.1 Continue with Example 5.1.1; Fig. 5.4 shows the *lifecycle* of a GSM process for `MSC` that prescribes how the process should be executed. The lifecycle starts from *stage* "Requirements Check". It is opened once the condition in the diamond-shaped *guard* is satisfied. The guard tests if a "Request Maintenance Check" event arrives. Once the stage is activated, some *sub-stages* can open. For example, if HHMB decides to revise the maintenance apartments plan, sub-stage "Partial Apts Check" can be activated. During the execution, outgoing events can be sent out to request execution of actual tasks outside environment (e.g., human-performed). Once the requirement is checked,

the circle-shaped *milestone* "Details Checked" will automatically close the associated stage. The instance finishes when milestone "Docs Archived" is achieved. ∎

Definition: A *GSM artifact schema* is a tuple $\Gamma = (R, Stg, Mst, Substg, Owns, Att, \mu, EType, Lcyc)$, where

- R is a (unique) name of Γ,
- Stg is a set of *stage names* (or simply *stages*),
- Mst is a set of *milestone names* (or simply *milestones*),
- $Substg \subseteq Stg \times Stg$ defines a forest that represents sub-stage relationships,
- $Owns$ maps each stage in Stg to a non-empty subset of Mst such that one milestone can be associated to only one stage,
- Att is a set of *data attributes* containing "ID" (to store all data used in the process),
- μ is a one-to-one mapping from $Stg \cup Mst$ to a set of special, *status attributes* (the domain of status attributes is Boolean to denote if a stage is open/closed or a milestone is achieved/invalidated),
- $EType$ is a set of event types such that for each $E(\sigma_1, ..., \sigma_n) \in EType$, $\sigma_i \in Att$ for all $i \in [1..n]$, and
- $Lcyc$ is the lifecycle model, which defines conditions to open/close stage and achieve/ invalidate milestones. Furthermore, it binds an outside task to an atomic stage.

In a GSM artifact schema $(R, Stg, Mst, Substg, Owns, Att, \mu, EType, Lcyc)$, sets Att, Stg and Mst are pairwise disjoint. More details of the formal model of GSM artifact schemas and lifecycle models can be found in [15].

Definition: Given a GSM artifact schema $\Gamma = (R, Stg, Mst, Substg, Owns, Att, \mu, EType, Lcyc)$, a *GSM artifact instance* of Γ is a triple $\Sigma = (id, \mathcal{V}_\mathrm{d}, \mathcal{V}_\mathrm{s})$, where id is a unique identifier (ID), $\mathcal{V}_\mathrm{d}$ and $\mathcal{V}_\mathrm{s}$ are two sets of attributes and values pairs such that for each

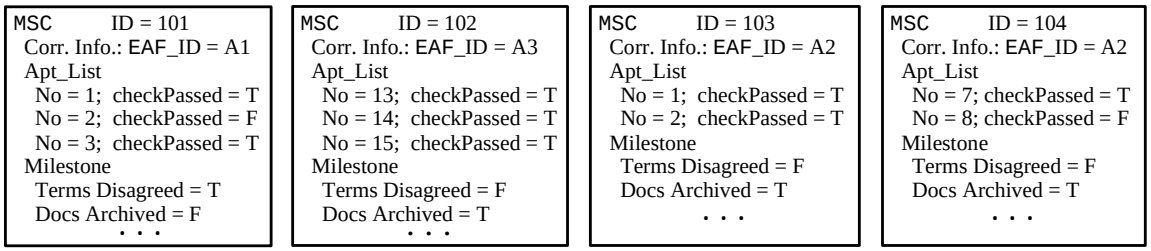

Figure 5.5: GSM Artifact Instances

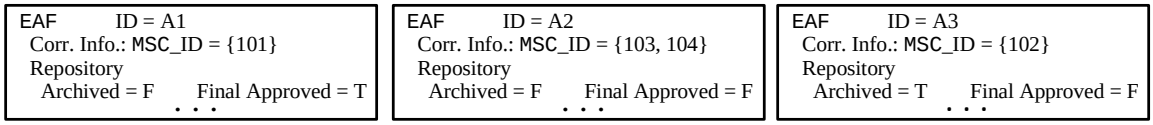

Figure 5.6: EZ-Flow Artifact Instances

$\sigma \in Att$, there is a pair $(\sigma, c) \in \mathcal{V}_{\mathrm{d}}$, where c is the value for σ, and for each $s \in Stg \cup Mst$, there is a pair $(\mu(s), c) \in \mathcal{V}_{\mathrm{s}}$, where c is *true* (T) or *false* (F).

Example 5.2.2 Fig. 5.5 shows four `MSC` artifact instances for the BP described in Example 5.2.1. The instance with ID = 101 has three maintenance apartments, where the one labeled "No. 2" failed the maintenance check. The milestone "Term Disagreed" is achieved to denote that the negotiation with the developer fails at the current moment. The attribute "`EAF_ID`" in `MSC` holds the correlated `EAF` business processes mentioned in Example 5.1.1. ∎

An artifact instance represents a running BP instance (with all data values). Artifact instances may depend on each other through their IDs stored as attribute values among themselves. If some attributes of an instance change during execution, other instances referencing this instance should possibly change as well. The BP engine must keep track of all dependency relationships.

The set of *foreign* IDs of an artifact instance Σ, denote by $\mathrm{FID}(\Sigma)$, is the set of all IDs

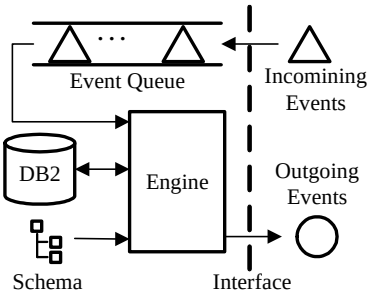

Figure 5.7: Barcelona prototype

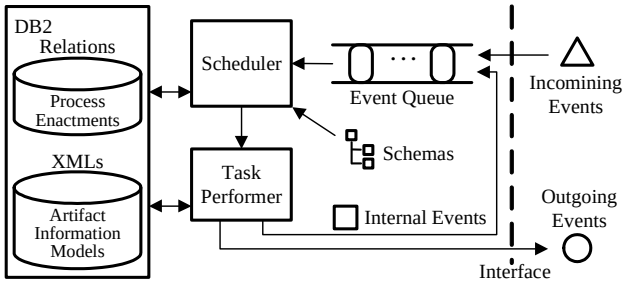

Figure 5.8: EZ-Flow Engine

appeared as attribute values in Σ except for the ID of Σ itself. Σ *depends on* an instance with ID value id if $id \in \mathrm{FID}(\Sigma)$.

Definition: A *GSM system* $\boldsymbol{G}$ is a finite set of GSM artifact schemas. A *GSM (system) snapshot* $\boldsymbol{S}$ is a finite set of GSM artifact instances of artifact schemas in $\boldsymbol{G}$ such that for each instance Γ in $\boldsymbol{S}$ and each $id \in \mathrm{FID}(\Sigma)$, there is an instance in $\boldsymbol{S}$ with ID id.

The semantics of GSM is defined in [41]. It is based on handling incoming events sequentially, and for each event, a "*B-step*" is performed atomically. Intuitively a B-step modifies the values of attributes in an artifact (instance) according to the schema, and once the instance is changed, the depending instances may also need to be changed as they may test the values in the depending instances.

Based on GSM semantics [15], an engine "Barcelona" [40] was developed. Fig. 5.7 shows the architecture. The communication between the environment and Barcelona is accomplished through events. The incoming events (sent by a task or a user) are handled sequentially. For each event, a B-step is performed to update the correlated artifact instance stored in a DB2 database according to the schema. Some depending artifacts may also change during this B-step. Once it is done, the engine proceeds to handling of the next event.

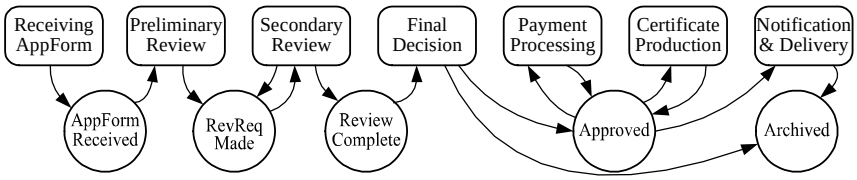

Figure 5.9: An EZ-Flow Model of `EAF` artifact

5.2.2 EZ-Flow

We briefly review the artifact-centric model EZ-Flow [16] through examples.

Example 5.2.3 Continue with Example 5.1.1; Fig. 5.9 shows a EZ-Flow process for `EAF`. An `EAF` instance is created when a developer submits a request for a pre-sell permit; and the instance will be stored in the *repository* "AppForm Received". When *task* "Preliminary Review" completes, the process will send multiple "Request Maintenance Check" events to create one or more `MSC` instances to request checking on maintenance. The remainder of the process is self-explanatory and ends once the instance is archived.

Fig. 5.6 shows 3 instances for the `EAF` process. The instance with ID $= A3$ has one correlated `MSC` instance with ID$=102$. The repository status "Archived" is true, indicating that this `EAF` instance is in the "Archived" repository and the process execution has completed. ∎

In EZ-Flow, artifact classes model the key artifacts associated with BPs. Each *EZ artifact class* has a distinct *name* and a set of associated *attributes*. (The notions of attributes and events are given in Section 5.2.1.) Each EZ-Flow process has a unique *core* artifact class. Other involved artifacts are *auxiliary*.

In EZ-Flow, artifacts are manipulated by *tasks* in their lifecycles. A task is triggered by an event and can produce one or more events when it completes. A lifecycle of an EZ artifact may consists of a sequence of tasks manipulating it. In-between tasks, artifacts

must be stored in "repositories". A *repository* has a unique name and an associated artifact class and contains a set of artifact instances of the class at runtime.

Definition: An *EZ artifact schema* is a tuple (A, X, E, T, F, R, L), where

- A is (the name of) the *core* artifact class, X a set of *auxiliary* artifact classes not containing A such that the set $X \cup \{A\}$ is closed under cross references,
- E is a set of event types,
- T is a set of tasks,
- F associates each task in T to a set of *triggering* events, and a set of *produced* events,
- R is a set of repositories, and
- L contains a set of directed edges between tasks and repositories with *guards* (conditions) as edge labels.

Definition: Given an EZ artifact schema (A, X, E, T, F, R, L), an *EZ artifact instance* is a quadruple (id, o, l, ρ) where id is a unique identifier (ID), o assigns attributes values in their domains, $l \in R \cup T$ indicates the current location (processing state), and ρ is a set of ids of auxiliary artifacts needed in the current state.

Example 5.2.4 Fig. 5.6 shows 3 instances for the `EAF` process described in Example 5.2.3. The `EAF` instance with ID = A1 has one correlated `MSC` artifact (instance) with ID = 101 (which is the one described in Example 5.2.3). The repository status "Archived" is false, indicating that this `EAF` instance still has process execution underway. ∎

Definition: An *EZ-Flow system* is a closed set of EZ artifact schemas (under references), an *EZ-Flow (system) snapshot* is a closed set of EZ artifact instances.

An operational semantics is described in [16]. A step of execution in EZ-Flow moves from one snapshot to another by performing a "transition" on one artifact instance. A transition is associated with execution of task and triggered by an event. When an event arrives, the task execution is initiated. The core and auxiliary artifact instances are fetched before the task is performed. When the task completes, all artifacts are stored back to repositories. Note that fetch and store actions are atomic: one snapshot may indicate that artifacts are in repositories, the next one would show the event consumed and all relevant artifact instances moved from repositories to the task, and in the third snapshot, all artifact instances could be in repositories again.

Fig. 5.8 shows the EZ-Flow engine [16], which consists of (1) a *scheduler* that responds to events and decides to launch tasks according to the EZ-Flow artifact schemas, and (2) many *task performers*, each managing one task execution.

5.2.3 Execution Independence and Universal Artifacts

In this section, a new notion of "universal artifacts" is introduced. The model abstracts key ingredients of artifact BP models so that the conceptual model is independent from the execution. By mapping heterogeneous artifacts to the same model, it is possible to monitor and query collaborative BPs (Section 5.5), even though the artifacts are running in different engines. Moreover, a universal artifact incorporates both the notion of artifact and the process model that this universal artifact will follow.

Current BP/workflow management systems are facing enormous difficulties arising from the need for (more) automation, process analytics (e.g., BI analysis), run time execution monitoring, process improvements, etc. A key cause for these difficulties is the lack of capturing *adequate* semantics of running BPs at the *conceptual level*. Conventional BP modeling languages allow specification of tasks/activities and their sequencing con-

straints (BPMN, Activity Diagrams, YAWL, etc.), leaving data modeling to some later stage at a lower conceptual level. BPEL on the other hand lacks necessary abstraction for modeling data involved in BPs that are essential for runtime monitoring and management. Artifact-centric models [12, 15] make a big step forward by integrating logical data models and task/activity models. However, current systems include only partial runtime context for BPs. Barcelona [40] stores artifact dependency and the execution state information directly in its local database and the dependency information is not visible in the conceptual level. In EZ-Flow [42], the states of obtaining auxiliary data and executing a task are also hidden in the conceptual level. This was recently elevated to the conceptual level to support on-the-fly changes [16].

We believe and advocate a fundamental principle for BPM systems:

Execution independence *refers to the freedom of making changes to the process execution engine while leaving conceptual BP models unchanged.*

A necessary ingredient to support execution independence is the ability to capture adequate semantics in conceptual BP models, specifically through logical modeling of all three types of data (schemas, business data, and states and context) as discussed at the beginning of this section.

In data management systems, "physical data independence" was a key enabler for the development of transaction models (concurrency, crash recovery) independently from query optimization. The independence principle could allow BP execution issues (scheduling, isolation, etc.) and BP modeling issues to be dealt with separately.

Essentially, a universal artifact (instance) is a GSM/EZ artifact augmented with (a) state and runtime dependency information, and (b) the artifact schema.

Definition: A *universal (artifact) schema* is a tuple (A, ID, Att, Sta) where A is a (unique) name, ID is the ID attribute, Att is a set of *data* attributes, and Sta is a set of

state attributes. Given a universal schema (A, ID, Att, Sta), a *universal artifact (instance)* of A is a tuple $(\nu, \mathcal{L}, \mathcal{M}, Dep)$ where ν assigns values to attributes in $\{ID\} \cup Att \cup Sta$ such that $\nu(ID)$ is a unique ID, $\mathcal{L}$ is either "GSM" or "EZ" representing a modeling language (GSM or EZ-Flow), $\mathcal{M}$ is an artifact schema in $\mathcal{L}$, and Dep is a set of *dependencies* whose representation depends on $\mathcal{L}$.

A universal artifact is an abstraction of *running instances* of both GSM and EZ-Flow artifacts. Each universal artifact captures data attribute values, status and dependencies, and (its own) BP schema. The inclusion of the schema removes "sharing" of the schema among artifact instances, and allows changes to be made at runtime without affecting other running instances, valuable for change support (Section 5.5).

To achieve execution independence for GSM (EZ) artifacts, all data concerning execution are extracted from Barcelona (EZ-Flow) and stored as universal artifacts. When Barcelona performs a B-step (EZ-Flow performs a step), it updates a GSM (EZ-Flow) system snapshot. Thus, it is necessary to establish a 1-1 mapping from GSM (EZ) instances to universal artifacts so that the fact of universal artifacts storing the system snapshot is transparent to Barcelona (EZ-Flow). In this section, we discuss a few technical notions for the mappings for GSM and EZ-Flow separately.

To begin with, we say that a GSM artifact schema $\Gamma = (R, Stg, Mst, Substg, Owns, Att, \mu, EType, Lcyc)$ is *compatible* with a universal schema (A, ID, Att', Sta), if A is the name R, $Att \subseteq Att'$ (data attributes of Γ are also that for A), and $Mst \cup Substg \subseteq Sta$ (stages and milestones of Γ are used as state attributes for A).

We fix $\boldsymbol{S}$ to be some GSM snapshot.

Definition: For each GSM artifact instance $\Sigma = (id, \mathcal{V}_{\mathrm{d}}, \mathcal{V}_{\mathrm{s}})$ in $\boldsymbol{S}$, the *dependency closure of* Σ denoted as $\Delta_{\Sigma}^{\mathrm{GSM}}$, is a set of IDs where (i) id is in $\Delta_{\Sigma}^{\mathrm{GSM}}$; (ii) For each $id' \in \Delta_{\Sigma}^{\mathrm{GSM}}$, $\mathrm{FID}(\Sigma') \subseteq \Delta_{\Sigma}^{\mathrm{GSM}}$ (all IDs referenced in id' are in $\Delta_{\Sigma}^{\mathrm{GSM}}$).

In Barcelona, once an event comes, it will first affect one GSM instance; during the same B-step, the effect may also ripple to the other depending instances. For each GSM instance Σ, $\Delta_{\Sigma}^{\text{GSM}}$ limits range of the snapshot that may be affected once Σ is changed.

The key notion relating GSM instances and universal artifacts is given below.

Definition: Let $\Sigma = (id, \mathcal{V}_{\text{d}}, \mathcal{V}_{\text{s}})$ be a GSM artifact instance of schema Γ with data attributes *Att* in a GSM snapshot $\boldsymbol{S}$, and $\Sigma' = (\nu, \mathcal{L}, \mathcal{M}, Dep)$ a universal artifact of a univesal schema compatible with Γ. Σ' is a *conceptualization of* Σ *in the context of* $\boldsymbol{S}$ if (i) $id = \nu(\text{ID})$ (IDs are identical), (ii) $\mathcal{V}_{\text{d}}$ and ν coincide on all attributes in *Att*, (iii) $\mathcal{V}_{\text{s}}$ and ν coincide on all stage and milestone attributes, (iv) $\mathcal{L}$ = "GSM", (v) $\mathcal{M} = \Gamma$, and (vi) $Dep = \Delta_{\Sigma}^{\text{GSM}}$.

Given a GSM artifact schema Γ and a GSM instance Σ of Γ, it is straightforward to create a universal artifact Σ' by simply mapping each attribute together with its value (if any) from Σ' to Σ. The mapping not only keeps the original ID, data and status attributes, but also includes the execution language and the schema. For the dependency set, though it can be derived from the data attribute values, it is necessary to raise it as the first-class citizen in order to explicitly denote the relationship with other instances as well as make it convenient for computing the GSM system snapshot.

We now consider the mapping for EZ-Flow. Let Γ be an EZ artifact schema with a set T of tasks, a set R of repositories, and the core artifact class A containing a set *Att* of attributes, and let $\Gamma' = (A', ID, Att, Sta)$ be a universal schema. Γ and Γ' are *compatible* if A', A has the name, $Att \subseteq Att'$ (data attributes of Γ are also that for A), and $(T \cup R) \subseteq Sta$ (tasks and repositories of Γ are used as state attributes for A).

Definition: Let $\boldsymbol{S}$ be an EZ-Flow snapshot. For each EZ-Flow artifact instances $\Sigma = (id, o, l, \rho)$ in $\boldsymbol{S}$, the (EZ) *dependency closure of* Σ, denoted as $\Delta_{\Sigma}^{\text{EZ}}$, is a mapping on ρ such that for each auxiliary artifact ID $id' \in \rho$, $\Delta_{\Sigma}^{\text{EZ}}(id') = r$, where r is a repository in the

schema of the artifact with ID id'.

We also view $\Delta_{\Sigma}^{\text{EZ}}$ as a set of ID-repository pairs. We now extend the notion of conceptualization to EZ-Flow artifacts.

Definition: Let $\Sigma = (id, o, l, \rho)$ be an EZ artifact instance of schema Γ with attributes Att in an EZ snapshot $\boldsymbol{S}$, and $\Sigma' = (\nu, \mathcal{L}, \mathcal{M}, Dep)$ a universal artifact of a universal schema compatible with Γ. Σ' is a *conceptualization of* Σ *in the context of* $\boldsymbol{S}$ if (i) $id = \nu(\text{ID})$ (IDs are identical), (ii) o and ν coincide on all attributes in Att, (iii) for each $x \in T \cup R$, $\nu(x) = true$ iff $x = l$ (the state attributes reflect the current execution state), (iv) $\mathcal{L} =$ "EZ", (v) $\mathcal{M} = \Gamma$, and (vi) $Dep = \Delta_{\Sigma}^{\text{EZ}}$.

Similar to GSM, it is also straightforward to map EZ-Flow artifact instances to universal artifacts. The logical description of the mapping is omitted.

5.3 The SeGA Framework and Support for BPaaS

A universal artifact clearly captures all necessary data for execution and allows for a BP engine to process without any data outside of the universal artifact. In this section, a framework called "SeGA" (Self-Guided Artifacts) is presented (Section 5.3.1) to show how to wrap (existing) BP engines into "stateless" services to support BPaaS (Section 5.3.2) and collaboration (Sections 5.4 and 5.5) and how universal artifacts can interact with the provided services. SeGA serves as a broker between BP engine and the environment. Inspired by the SeGA framework, we envision how a new architecture for BPM systems that can support BPaaS and collaborations (Section 5.3.3). Finally, a SeGA prototype was developed and briefly described in [24], the design details are discussed in Section 5.3.4.

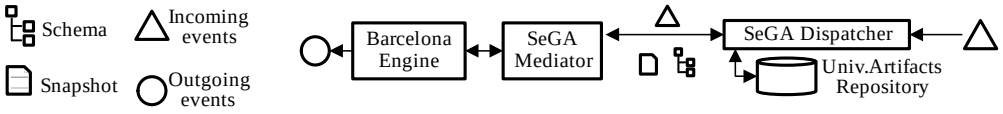

Figure 5.10: The SeGA Framework

5.3.1 The SeGA Framework

Fig. 5.10 shows the architecture of the SeGA framework (or simply SeGA), which consists of a *SeGA dispatcher* and a *SeGA mediator*. When an external event arrives, the dispatcher fetches the relevant universal artifact from a universal artifact repository, extracts the schema from the universal artifact and maps it back to the original form (GSM or EZ-Flow) restores the original artifact instance (GSM or EZ-Flow), and sends the external event, schema, and the original artifact instance to the mediator. When the mediator receives the event, schema, and the instance, it deposits the artifact schema in the appropriate location where the Barcelona/EZ-Flow engine will access, and passes the control over to the Barcelona/EZ-Flow engine by forwarding the event. When the Barcelona/EZ-Flow engine receives the incoming event, it executes the next step and updates the artifact instances according to the schema deposited by the mediator; and outgoing events may also be sent directly from the engine if there exists task invocation during the execution. Once it completes, the mediator fetches the updated artifact instances, together with their schemas and states, and sends them back to the dispatcher. The dispatcher then maps the instances and schemas back to universal artifacts and stores into the corresponding repository.

SeGA requires a universal artifact repository so that the dispatcher can fetch universal artifacts. In general, an enterprise stores the data in a enterprise persistent data store (e.g., a relational database) rather than storing data for individual BP models. A general approach of a data mapping to bridge the relationships between artifacts and databases

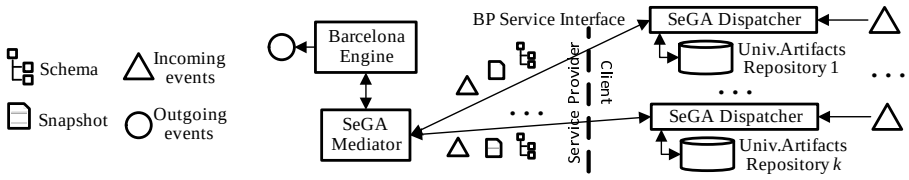

Figure 5.11: SeGA to Support BPaaS

was developed in [26]. As an advantage, one can design artifacts and map the artifacts (data) into an existing database. The mapping in [26] allows to propagate updates on artifact instances to the database and vice versa.

5.3.2 Supporting BPaaS

Based on Fig. 5.10, SeGA can be used to support BPaaS, as shown in Fig. 5.11. The dispatcher would reside at the service *consumer* (or *client*), where a repository of universal artifacts is maintained. The mediator is located at the service *provider* who runs a BP engine (or multiple engines to balance workload). The dispatcher and mediator communicate through service invocations such as WSDL or REST, and work in pairs so that the service provider can use its BP engines to execute BP received from the service consumer in the form of data.

The SeGA framework takes the advantage of the execution independence that separate data and execution management. From the engine's perspective, it provides business-process-as-a-service but does not maintain any data. This allows the provider to serve a large number of consumers. From the consumer's view, all BP data are maintained at its site; beyond that, there is no need to manage BP execution.

BPaaS can be achieved by separating data management from the execution engine and let the execution engine simply provide stateless services with zero knowledge of what data should be processed and the context. Notice that the engines in Fig. 5.11 have no data repository to store the information of BPs. This makes it easier for enter-

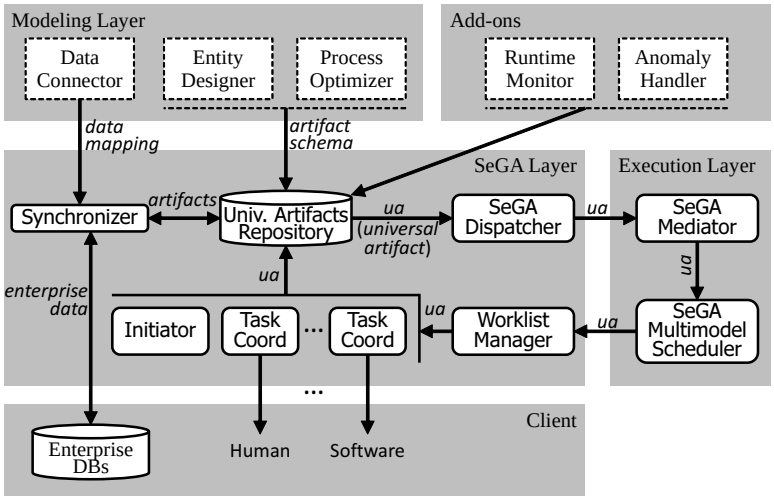

Figure 5.12: A Conceptual Architecture of BPM Systems

prise to purchase BPaaS services to manage their business processes instead of in-house maintenance of the BPM system. Meanwhile the enterprise has a full control over the management of the BP data as well as enterprise data.

5.3.3 A Design Methodology to BPM Systems

Our study on SeGA leads to two specific suggestions for future BPM system development. First, existing BPM systems can be augmented so that data in the process manager is extracted and packaged with the business data into universal artifacts. Although we only explored two systems, the same method is applicable to other systems including jBPM and possibly YAWL. Section 5.3.4 provides a general methodology for this. Second, in general it is most desirable to develop future BPM systems that support the independence principle. In this regard, we envision that a BPM system consists of three layers, a *modeling layer* to accept/analyze the data and BP design, and map to universal artifacts; a *SeGA layer* to manage universal artifacts and interact with the engine at runtime; an *execution layer* to manage executions with no local data. Such new style BPM systems will provide a tremendous support for BPaaS and process collaboration.

Fig. 5.12 presents a conceptual architecture for future BPM systems. The three key layers: modeling, SeGA, and execution layers are explicitly shown. For modeling layer, "entity designer" provides tools for people to design artifact/BP schemas and the designed schemas will be optimized by the "process optimizer" and deposited into the repository in the SeGA layer. The "data connector" is a tool that helps people to define mappings between artifact data and enterprise data [26]. The mapping will be maintained by the "synchronizer" at runtime in the SeGA layer. For execution layer, it contains only a SeGA mediator and a multimodel scheduler (which may be a collection of several BP engines, one or more for each BP language) that have been shown in Fig. 5.10 and Fig. 5.11. In addition to the synchronizer, SeGA repository, and the SeGA dispatcher in the SeGA layer, this layer also contains a "worklist manager" that passively receives the execution result from the multimodel scheduler, so that the manager can decide what task is to perform next. Each task performance is controlled by either (1) a "task coordinator" that informs human or software to execute the task, or (2) a "initiator" whose only job is to initialize the instance of an artifact.

In addition to the three layers in Fig. 5.12, some add-ons can be built upon this architecture. Some example add-ons could be runtime monitor, constraint enforcement component, or anomaly handler (to support runtime changes). In general, these add-ons only need to connect with the repository as it is the only place that all process data is stored. Some add-ons and their advantages will be discussed in Section 5.5.

5.3.4 A SeGA Prototype

We now describe the technical details of a prototype for SeGA. We first give the details steps of how Barcelona and EZ-Flow are configured in SeGA and how SeGA can interact with them. We then briefly outline how a new BP engine can be plugged into SeGA.

Based on the SeGA framework in Fig. 5.10, the prototype consists of a dispatcher and a mediator. Both dispatcher and mediator are written in JAVA. In addition to the dispatcher and mediator, the prototype also implements a universal artifact repository using MySQL. A RESTful interface is used for accepting incoming events to SeGA dispatcher, while SeGA dispatcher, SeGA mediator, and the Barcelona/EZ-Flow engines interact with each other through their RESTful interfaces. In addition, SeGA mediator also interacts with the two DBMSs used by the engines (resp.) for storing and fetching artifacts. Finally, SeGA mediator needs to deposit schema definition files in the appropriate locations of the two engines. In the current implementation, SeGA mediator and the two engines run on the same machine and SeGA mediator simply overwrites the schema files in the appropriate locations. Once the remote file copy ability is provided, SeGA mediator and the engine(s) can run across a network.

We now show how in detail Barcelona and EZ-Flow is configured and executed in SeGA. A new Barcelona or EZ-Flow BP engine is registered into SeGA through specifying a configuration file including IP address, database address, and schema location. SeGA will then automatically fetch, transform, and deposit the artifact instances from/into the corresponding engine.

Interacting with Barcelona

Barcelona [40] was treated as a black box since the source code is not available to us. Based on the mapping discussed in Section 5.2.3, ideally, the following steps can be applied for each incoming event that arrives at SeGA.

1. Fetch a universal artifact matching the correlation ID in the incoming event.

2. For each ID in the dependency set of the universal artifact, generate a GSM artifact and store it into the Barcelona engine.

3. Forward the original event to Barcelona to trigger one B-step. (After the B-step,

Barcelona should update the GSM instances.)

4. If there are outgoing events sent from Barcelona, then capture these events and forward them to the original receivers.

5. Fetch the updated GSM artifacts, map them back to universal artifacts and store them in the repository.

In Step 2, when an incoming event arrives, SeGA dispatcher should map universal artifacts to GSM instances with a schema. However, our implementation encountered a couple of technical problems concerning interaction with Barcelona.

Barcelona uses auto-ID feature in DB2 for generation of IDs in an auto-increased manner. Therefore, every time SeGA restores a GSM instance (from a universal artifact) and attempts to insert it into DB2, artifact IDs will change automatically. Given that the GSM structure in DB2 cannot be changed, SeGA simply keeps a duplicate of each instance in DB2. Thus, when an event arrives, SeGA dispatcher does not need to map universal artifacts back to GSM instances. However, a mechanism is provided in SeGA to make updates directly on universal artifacts in SeGA, and the changes will be reflected in the corresponding GSM instances in DB2 via SQL updates.

Once a B-step has completed (Step 3), Barcelona should notify SeGA mediator to fetch the updated GSM instances. However, Barcelona is passive and a change in the source code is needed to send a completion signal. To avoid this problem, a special artifact is deployed whose job is to consume *auxiliary control* events. When such an event is consumed, this artifact will invoke a task to send an outgoing message to SeGA mediator to inform a completion of the previous B-step. Since Barcelona consumes one external event to perform a B-step, by alternating external events and auxiliary control events, SeGA keeps track of B-step completions and fetches GSM artifact instances appropriately.

Step 4 allows to implement, e.g., a constraint checking function (Section 5.5). To

capture an outgoing event, SeGA (1) replaces the addresses of all the outgoing events specified in the schema files by its own addresses; (2) maintains a table to record the mapping between the new addresses and the original addresses; and (3) when receive an outgoing event, forwards it to the original receiver according to the mapping.

Interacting with EZ-Flow

SeGA interaction with EZ-Flow [16] follows 4 similar steps, here we briefly explain key differences. Due to the availability of the source code, slight modifications to EZ-Flow are done to simplify the interface.

SeGA stores universal artifacts in MySQL. However, EZ-Flow manages execution state data in DB2 while keeping data attributes in XML documents. Therefore, the instance mapping between SeGA and EZ-Flow requires XML-relation transformation.

In Step 2, SeGA is to restore the core and auxiliary artifact instances in the EZ-Flow engine and its corresponding repositories using the dependencies in the universal artifact. A notable difference is that when the auxiliary artifacts change their repositories (state) in their process execution, the dependencies in universal artifacts in SeGA must be refreshed (explained in the next paragraph).

Once a task completes in EZ-Flow, we modified the EZ-Flow engine to notify SeGA mediator proactively. This avoids having auxiliary control events used for interaction with Barcelona. Furthermore, the completion event to SeGA reports both core artifact modifications and dependent artifacts (e.g., new artifact created, new location). Upon receiving the completion event, SeGA directly fetches the artifact instances and transforms to universal artifacts. SeGA will also trace all dependencies in all universal artifacts in its repository and update the location information corresponding to the newly fetched EZ-Flow instances.

Upon completion, a task performer may produce event(s) to the EZ-Flow engine to

activate the next task. Again, the engine is slightly modified to send this event to SeGA that forwards it to the event queue of EZ-Flow.

Registering and Interacting with Other BP Engines

The SeGA prototype is able to support the installation of new BP engines. To register a new engine, two wrappers (one each for SeGA mediator and dispatcher), are needed. The wrappers are in form of plug-ins, where SeGA is able to invoke their interfaces. The mediator wrappers should be responsible for interacting with the new BP engine, receiving/sending artifact instances, events, and schemas from/to the dispatcher wrapper, who is responsible for interacting with universal artifacts in the repository. Note that the SeGA dispatcher or mediator will not interact with the BP engine nor the repository directly, as the format of the new artifacts or the semantics/interfaces of the new engines are unknown to SeGA. Therefore, SeGA dispatcher or mediator needs to invoke its corresponding wrapper to accomplish the job.

In general, the wrappers for mediator and dispatcher should be designed to meet the following requirements.

1. The dispatcher wrapper should be aware of how to pack and unpack artifacts together with other information into universal artifacts, and

2. Mediator wrapper, should know (1) how to compose and decompose universal artifacts, and (2) where to store/fetch the decomposed data into the BP engine.

The specific interaction steps are similar to that given for Barcelona and EZ-Flow but will be specific for the BP engine.

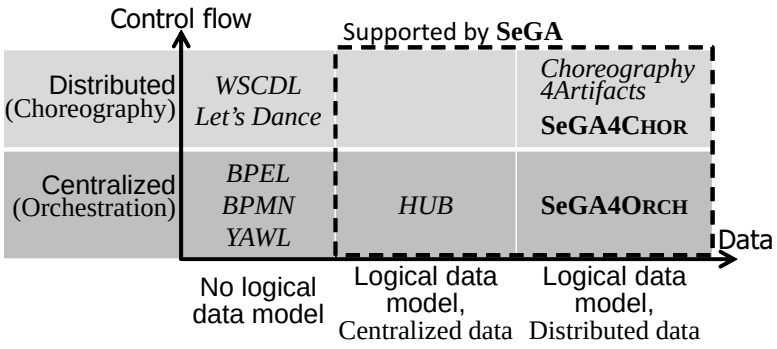

Figure 5.13: Classification of Collaboration Models

5.4 A Classification of Collaborative Process Models

In this section we provide a new classification of collaborative BPs based on two dimensions: whether control flows are centralized/distributed and availability of logical data models in conjunction with centralized/distributed data management. We then discusses how SeGA supports all four models that incorporate data.

Consider the collaborative BPs `EAF` and `MSC` in Example 5.1.1. To implement the BPs, typically an orchestration or choreography approach [8] can be employed. For orchestration, a BP engine serves as an orchestrator to coordinate all participants during the execution. Several orchestration languages, such as BPEL, BPMN, or YAWL can be used to specify the orchestrator process. Alternatively, the collaboration can be specified in a choreography language such as WSCDL or Let's Dance [43].

However, none of the above BP modeling languages support logical data modeling. Even though variables can be used, the lack of the global view of data involved hinders the ability to reason about collaborations, e.g., consistency with underlying databases, or the execution logic. For example, the staff of HHMB frequently want to know for a specific `EAF` instance, how many buildings have passed the maintenance apartment check, or which correlated `MSC` instances are currently under negotiation with developers; these cannot be easily answered, unless with ad hoc software modules.

We classify collaborative BP models along data and control flow dimensions as shown in Fig. 5.13. Clearly, orchestration means centralized control, while choreography represents distributed control. Along the data dimension, the languages discussed above lack logical data modeling capability. Among the modeling approaches that support logical data modeling are HUB [44] and Choreography for Artifacts [45].

In the HUB framework [44], data (such as customer name, ordered items, prices, payment, etc. in the above example) is explicitly specified and stored in a centralized repository. All participants interact with a hub engine (an artifact-centric process engine with view authorization [46]) to engage in collaboration. The hub serves as a scheduler as well as a data manager to maintain the process and the data. Note that all data involved in the collaboration are managed centrally at the engine.

Choreography for Artifacts [45], on the other hand, models the data in collaboration globally but requires each participant to manage its relevant global data locally. The relevant global data is modeled as an "artifact interface". Thus, the choreography language is classified as distributed control and logical data model with distributed data management in Fig. 5.13.

The SeGA framework is capable of supporting all four collaboration models that have logical data models. In the remainder of the section, we illustrate the support for two collaboration models, SeGA4Orch and SeGA4Chor (Fig. 5.13).

SeGA4Orch. Consider a SeGA framework that involves participants 1, 2, and 3, and BP engine 1 shown in Fig. 5.14 (ignore the other participants and BP engines 2 and 3 for now), where participants 1 and 2 maintain their own universal artifacts storage; while participant 3 stores its universal artifacts in a cloud storage. BP engine 1 is an orchestrator to coordinate the collaboration among the three participants. Then, the orchestration under SeGA framework proceeds as follows: A single universal artifact is

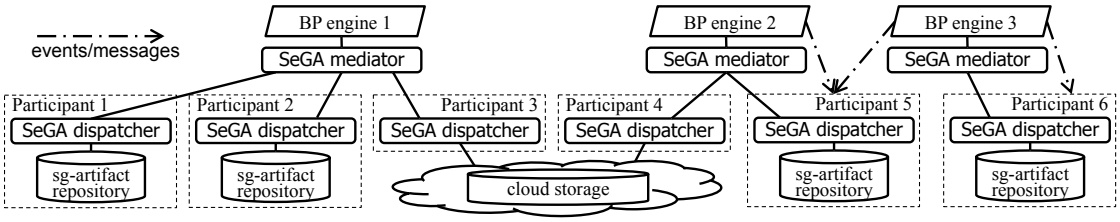

Figure 5.14: Collaboration Models Supported by SeGA

designed to serve as the schema for the orchestration. When the orchestration starts, the corresponding artifact instance (or equivalently, the orchestration instance) is generated by the first task performer (one of participants 1, 2, and 3). Once the first task of the orchestration is done, the task performer will pass the decomposed universal artifact (i.e., the original artifact instances together with their schemas) to BP engine 1, where the instance can be executed according to the schema. After the execution, the mediator extracts the execution results and passes them back to the dispatcher of a participant, who is responsible to perform the next task. The above routine repeats until the orchestration completes.

Compared with the HUB framework, the main difference between SeGA4Orch and HUB is that artifact instances are maintained by each participant (SeGA4Orch) or by the orchestration hub (HUB).

SeGA4Chor. Consider a SeGA framework that involves only participants 4, 5, and 6, and BP engines 2 and 3 shown in Fig. 5.14. Suppose that participants 4 and 5 run their own BPs by using the service provided by BP engine 2 and participant 6 uses the service from BP engine 3. When a choreography proceeds, the only means for participants to communicate is by sending and receiving messages/events. Under the traditional choreography setting, BP engines are responsible for message sending and receiving. However, in SeGA4Chor as shown in Fig. 5.11, for each participant, the messages are only received by its corresponding dispatcher, and sent by the engine that

provides service to it. Therefore, in SeGA4Chor engines send messages to dispatchers, and participants communicate with engines through the SeGA paradigm discussed in Section 5.3.1.

In SeGA4Orch and SeGA4chor, data management is done by individual participants. However, SeGA can be easily adapted to the case when a centralized cloud data storage is used to maintain all needed data for the collaboration (e.g., as particularly illustrated in cloud storage shared by participants 3 and 4 in Fig. 5.14).

5.5 Runtime Support

In addition to SeGA's capability to support a variety of collaboration models (discussed in Section 5.4) additional runtime support can also be easily provided for SeGA collaboration including, in particular, runtime queries, constraint enforcement, and dynamic modifications.

Querying and monitoring

Universal artifacts provide uniform structures to record the business data, schema, and status data. Such structures facilitate querying (both current and completed) execution of collaborative BPs, even when different participants use different BP engines.

We develop a query language "aQL" for artifacts that incorporates artifact and BP concepts into an OQL-like syntax (Object Query Language [47]). aQL supports the notions based on universal artifacts including "instances", "IDs", and "states".

In the following we present two queries to illustrate aQL. The queries are formulated against the two BPs `EAF` and `MSC` discussed earlier. Although some of the artifact instances are shown in Fig. 5.5 and Fig. 5.6 for GSM and EZ-Flow, aQL acts on their universal artifact renditions (that are not shown).

Example 5.5.1 Continuing with Example 5.2.1, consider the query to find all IDs of `MSC` artifacts with at least one apartment that failed the maintenance apartment check. This query is expressed as:

```
SELECT M.ID FROM MSC M WHERE EXISTS (SELECT *
FROM M.Apt_List A WHERE NOT A.checkPassed)
```

In the above expression, `MSC` is the name of a universal artifact schema, `M` is a variable representing an `MSC` artifact instance, `M.ID` and `M.Apt_List` denote the values of attributes "ID" and "Apt_List" (resp.) where `Apt_List` is a set valued attribute consists of a set of apartment groups, and `A` is a variable holding the data for a group including `checkPassed` (Boolean). This query on the universal artifact version of the four `MSC` artifacts in Fig. 5.5 would return the answer $\{101, 104\}$. ∎

aQL uses path expressions to access nested structural values in the same way as OQL. Nested queries are easily incorporated. In the query from the above example, if some apartment `A` fails the check, the nested query returns a nonempty set and the `MSC` ID will be returned for the outer query.

Example 5.5.2 Consider both BPs `MSC` and `EAF` in Fig. 5.5 and Fig. 5.6. The following query lists all IDs of `EAF` artifacts that have not finished but have at least one correlated `MSC` artifact archived in the maintenance apartment check process.

```
SELECT DISTINCT P.ID FROM MSC M, EAF P
WHERE M.ID IN P.MAC_ID AND M.Docs_Archived AND NOT P.Archived
```

Similar to cross product in SQL, the above query selects combinations of `MSC` and `EAF` instances satisfying all three conditions in the WHERE clause. For our example, `MSC` instances with ID 102, 103, and 104 are archived; and `EAF` instances with ID "A1" and "A2" have not completed. However, only 103 and 104 are in `MAC_ID` of "A2"; therefore the query returns "A2". ∎

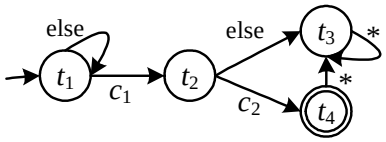

Figure 5.15: Constraints for `MSC`

Note that the concepts of "milestone", "stage", "task", and "repository" are all mapped to *states* in aQL.

Enforcing choreography constraints

In collaborative BPs, choreography constraints are used to restrict how one BP should execute in a collaboration with other BPs to prevent the process from behaving undesirably. Generally, if the participant BPs use different workflow engines, then for each engine, the constraints should be specified in a way that works for the engine. Furthermore, some engines may not have a clear model of data used by a BP, which gives rise to the difficulty in checking data-related constraints. In order to overcome these problems, the SeGA framework provides a uniformed approach for specifying constraints and maintain them at runtime.

SeGA uses a state machine to model constraints for a BP and monitor running instances. In particular, when an instance is created, a state machine then is associated. SeGA maintains a *state table* that records universal artifact IDs and the current states of the corresponding state machines. The constraints are based on ECA (event-condition-action) rules, i.e., when an event is received (or sent) by a running instance, if the corresponding condition is satisfied, a transition of the associated state machine is made, and this change is recorded in the *state table*. If at the end of the lifecycle of the running instance, the state machine reaches a final state, the constraints are satisfied.

An *event formula* ξ is of form "**in** $E(\phi_1, \phi_2, ..., \phi_n)$" or "**out** $E(\phi_1, \phi_2, ..., \phi_n)$", where each ϕ_i is an atomic condition on the event content. **in** $E(\phi_1, \phi_2, ..., \phi_n)$ (or **out** $E(\phi_1,$

$\phi_2, ..., \phi_n)$) is true if and only if an incoming (or outgoing) event arrives (or sent) and each ϕ_i is true on the contents of the current event.

Example 5.5.3 Continue with Example 5.1.1; if a `MSC` instance sends out a maintenance report (MR) whose attribute "report_result" has value "approved", the event formula "**out** E(event_name='MR', report_result='approved')" is true. ∎

With the testing of an event only, a condition is not expressive enough to capture constraints. A query is further needed to test against the values inside a universal artifact. A *condition* is a formula of form "Q **when** ξ" or "ξ", where ξ is an event formula and Q is a aQL query. Given a universal artifact σ, a constraint is satisfied based on σ if (1) an event for σ arrives (or sent by σ), (2) ξ is evaluated to true, and if Q is present, (3) the result set of Q queried on σ is empty.

With a condition defined, a state machine can be constructed to test if a set of conditions can be satisfied during the execution.

Definition: A *(choreography) constraint* is a tuple (T, s, F, C, δ), where (1) T is a set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states, (2) C is a set of conditions; (3) $\delta \subseteq T \times C \times T$ is a set of transitions.

A constraint is essentially a state machine with conditions on edges. The semantics of a constraint follows a traditional manner.

For each universal artifact σ, a constraint c should be associated. σ satisfies c if c can reach a final state by the time when the lifecycle of σ ends.

Example 5.5.4 Continuing with Example 5.1.1, when an `MSC` instance is created, it will send maintenance reports back to the correlated `EAF` instance. If the report states that

the maintenance apartment check is passed, then in the future, the `MSC` instance is only expecting an archive message. Fig. 5.15 shows the constraint in form of a state machine for `MSC` artifact. t_1 is the initial state and t_4 is the only final state. Two conditions c_1 (to specify a passed report) and c_2 (to specify an archive message) are specified:

- c_1 is of form "Q **when** ξ", where ξ is an "event formula" of form "**out** E(event_name = 'MR', report_result = 'approved')" to denote that a `MSC` instance is expected to send out a maintenance report (MR) event whose attribute "report_result" has value "approved" after its initialization; and Q is the query in Example 5.5.1. c_1 is evaluated to be true if the result set of Q queried on the current `MSC` instance is empty and ξ is true. Hence, c_1 denotes that an `MSC` instance should send an "approved" report and all the apartments should pass the check (notice that the query Q is only applied to the current `MSC` instance instead of all the `MSC` instances).

- c_2 is of form "ξ", where ξ is event formula "**in** E(event_name = 'CA')" to denote that an archive message is expected to be received.

An edge labeled with "else" stands for a collection of transitions other than the specified one(s) leaving the same state. An edge labeled with "*" represents all possible transitions leaving the state. ∎

Constraints checking requires the knowledge of when events are received or sent. For SeGA, the incoming events are handled by dispatchers; while the outgoing events are sent directly by engines. Therefore, in order to enable SeGA to maintain the state machine correctly, a modification is needed to let the engine inform SeGA when an outgoing event is sent.

Dynamic modification

BP models change often. In current BPM systems, the specifications of a BP model is shared by *all* running instances of the model. Modification of the model specification

becomes difficult: should a running instance follow the old model or the new model, which is called the instance migration problem. Also, during the execution of a BP, ad hoc changes may happen to some running instances. Those changes are temporal, more likely "one-off", but diversiform. In the traditional approach, it is non-trivial to implement such behavior due to the following points.

1. The lack of a conceptual model with complete semantics of workflow execution. That means making a change in traditional workflow not only depends on the workflow model (schema) and the instance but also on the specific execution mechanism in the execution engine, the latter is hard to acquire and understand.

2. The difficulty of restricting a change within a scope. In other words, sometimes a change of an instance do not intend to affect other instances.

A novel approach was presented in [16] to allow the intended changes specified using rules while the original EZ-Flow model stay unchanged. Both rules and model specification are shared among all running instances. At runtime, each instance will have a chance to check if a rule requires the execution to be altered on the spot. Four types of process changes were supported there: *skip*, *replace*, *add*, and *retract*.

The SeGA framework does not provide any methods to make changes to BPs. However, the framework naturally provides a mechanism that can facilitate runtime changes such as the BP change operations described in [16]. By inclusion of the schema in each universal artifact, SeGA elevates runtime process modifications to the conceptual level and simplifies complex implementation details. SeGA can handle all four types of modifications defined in [16] and more.

The following Example 5.5.5 shows how to **skip** execution of a task for a running artifact instance. And the **retracting** of a task can be done in a similar approach.

Example 5.5.5 Continue with Example 5.2.3; consider the `EAF` instance with ID = A2 and the value of "AppForm Received" being "T" (the process instance is waiting for executing the task "Preliminary Review" shown in Fig. 5.9). Suppose that this BP instance is recognized as an urgency case. HHMB wants to skip the "Preliminary Review" and the "Secondary Review" tasks to speed up only this instance. With the help of universal artifacts this skip change is easy to be done. First the `EAF` artifact snapshot can be mapped into a universal artifact snapshot carrying the state of process execution. Then change the current state in the snapshot from the state right before the task execution to the state right after the execution, i.e., set the current state of "Review Complete" to be true. At last, when the universal artifact maps back to an EZ-Flow artifact, it will lead to the EZ-Flow engine continue the instance from the new state, effectively skipping "Preliminary Review" and "Secondary Review" tasks. for this running instance, and this change does not affect other instances. ∎

The following Example 5.5.6 shows how to **add** a task in a running artifact instance. **Replacing** of a task for a running instance can be done in a similar way.

Example 5.5.6 Consider a new government policy that requires HHMB to check background of each developer who never had any early-sell permits. HHMB decides that this check should be done before the final decision in `EAF` (see Fig 5.9). Consequently, a new task "Background Check" is added in `EAF` before executing "Final Decision" effective immediately. Since each universal artifact for `EAF` contains the `EAF` model, this change can be done by replacing the current `EAF` model with a slightly modified model in which the background check task is added (conditionally). ∎

5.6 Summary

The demand for BPaaS is emerging while collaborative BPs remains a challenge. We have seen various vertical BPaaSs in for example HR and procurement. Clearly BPaaS is not just about providing APIs and interfaces for configuration and graphical analysis. The challenges lie in the capability to handle massive scaling, the service must be able to support multiple languages and execution environments, as well as massive customers and processes. We argue that the separation of the data from the execution engine is a good way to meet this demand. We demonstrate in the chapter that the SeGA framework provides a holistic approach in supporting this separation and result in a uniform way of facilitating different BP collaboration frameworks and supporting runtime analysis.

Chapter 6

Declarative Collaboration for Artifacts

A choreography models interoperation among multiple participants in a distributed environment. Existing choreography specification languages focus mostly on message sequences and are weak in modeling data shared by participants and used in sequence constraints. They also assume a fixed number of participants and make no distinction between participant types and participant instances. Artifact-centric business process models give equal considerations on modeling data and on control flow of activities. These models provide a solid foundation for choreography specification. This chapter of the thesis makes two contributions. First, we develop a choreography language for artifacts with four new features: (1) Each participant type is an artifact schema with (a part of) its information model accessible by choreography specification. (2) Instance level correlations are supported and cardinality constraints on correlation of participant instances are explicitly defined. (3) Messages have data models, both message data and artifact data can be used in specifying choreography constraints. (4) The language is declarative based on a mixture of first order logic and a set of binary operators from

DecSerFlow. Second, we develop a realization mechanism and show that a subclass of the choreography specified in our language can always be realized. The mechanism consists of a coordinator running with each artifact instance and a message protocol among participants.

The remainder of this chapter is organized as follows: Section 6.1 introduces and motivates the need for a instance-level choreography language with data, Section 6.2 defines each component of the choreography language, a realization protocol for a subcase of the language is proposed in Section 6.3, and Section 6.4 summarizes the chapter.

6.1 Instance-Level Collaboration with Data

Enterprises nowadays rely on business process systems to support their business, information flows, and data analytics [48]. Interoperation among business processes (in a distributed environment) continue to be a fundamental challenge. In general, two approaches [8, 49], namely *orchestration* and *choreography*, are used to model interoperation. An orchestration requires a designated "mediator" to communicate and coordinate with all participating business processes. One well-known orchestration language called BPEL [50] has been widely used in practice. However, orchestration reduces the autonomy of participating business processes and does not scale well due to the mediator. The choreography approach specifies desirable global behaviors among participating business processes but otherwise leaves the business processes to operate autonomously and communicate in peer-to-peer fashion. One difficulty for this approach is to coordinate among participating business processes in absence of a central control point.

A choreography models interoperations among multiple participants in a distributed environment. A choreography may be specified as a state machine representing message exchanges between two parties [51] or permissible messages sequences among two or more

parties with FIFO queues [52]. It may be specified in individual pieces using patterns [43], or implicitly through participants behaviors [53].

Data has been playing a more essential role in business process modeling [2]. The interoperation of business processes also needs data to precisely specify the global behavior among participants. Existing choreography languages focus mostly on specifying message sequences and are weak in modeling data shared by participants and used in choreography constraints. A tightly integrated data model with message sequence constraints would allow a choreography to accurately constrain execution. Also, these languages assume a fixed number of participants and makes no distinction between participant *types* and participant *instances*. For example, an *Order* business process instance may communicate with many *Vendor* business process instances. Therefore, a choreography language should be able to model correlations between business process instances.

Artifact-centric business process models (introduced in Section 2.2), which contain a complete specification of business data (i.e., business entity), provide a solid foundation for choreography specification.

This chapter focuses on choreography specification, execution semantics, and realization. And makes the following TWO technical contributions. **First**, we develop a choreography language with four distinct and new features: (1) Each participant type is an artifact model with a specified part of its *information model* accessible by choreography specification. (2) Correlations between participant types and *instances* are explicitly specified, along with *cardinality constraints* on correlated instances. (3) Messages can include data; both message data and artifact data can be used in specifying choreography constraints. (4) Our language is declarative and uses logic rules based on a mix of first-order logic and a set of binary operators from DecSerFlow [54]. **Second**, we formulate a distributed algorithm that realizes a subclass of the choreography in our language.

In the remainder of this section, we illustrate some important concepts through exam-

ples including artifact-centric business processes [2] as well as their choreographies, and motivates the need for specification of correlations among process instances and choreographies with data contents. Further, we provide the key skeleton for the proposed language and explain the reason to make it declarative.

Consider an online *store* that provides various items for customers, where all the items are available at *vendor*s. A vendor may use several *warehouses* to store and manage its inventory. Once the customer completes shopping, she initiates a payment process in her *bank* that will send a check to the store on her behalf. Meanwhile, the store groups (1) the items in her cart by warehouses and sends to each warehouse fulfillment, and (2) the items by vendors and requests each vendor to complete the purchase. The vendors inform warehouses upon completion of purchase. After the store receives the payment and vendors' completion of purchases, the store asks warehouses to proceed with shipments.

In this example, four types of participants (store, vendor, warehouses, and bank) are involved and each type has its own business process. Although store and bank have only one process instance each, there may be multiple instances for vendor and for warehouse. In artifact-centric modeling, an artifact instance encapsulates a running process. For example, the store may initiate an "`Order`" (artifact) instance. Fig. 6.1 shows a part of the structured data (i.e., business entity or information model) in an `Order` instance. (Note that in this chapter, we do not require a business entity to have a key or a local key). The structure contains attributes "ID", "(shopping) Cart", etc. Moreover, the "Cart" is a set-typed attribute (denoted by "$*$") that may include 0 or more tuples with four nested attributes: "Inv(entory)_ID", "(item) Name", "Quan(tity)", and "Price". Similarly, other participant business processes are also artifact instances: `Purchase` instances represent order processing at vendors; `Fulfillment` instances are packing and delivery processes at warehouses; and a `Payment` instance is initiated upon a customer request to make a payment to the online store.

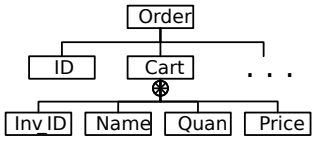

Figure 6.1: Biz entity

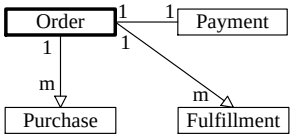

Figure 6.2: Corr. digram

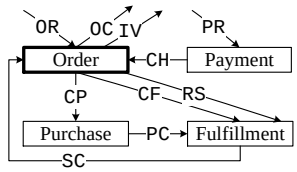

Figure 6.3: Message diagram

Consider the design of a choreography for the collaborative business process in this example, there are two major difficulties. First, existing languages do not support multiple participant instances, and thus the fact that multiple vendor/warehouse instances cannot be easily represented and included in specifying behaviors. Some process algebra based languages allow creation of new instances from sub-expressions in a choreography [55, 56], but it is not clear how it is related to multiple participant instances. Second, behaviors often depend on data contents. For example, when an order request is received with total amount $\geqslant$10, the order processing should proceed as described in the above; for orders with amount $<$10, the processing may be optional. Such conditions on data cannot be easily expressed in most languages. WS-CDL [57] may express this through copying messages to variables, but copying introduces unnecessary data manipulations.

In this chapter we develop a new choreography language which can deal with data contents in messages and from business process, and allows multiple participant instances.

In general, a choreography language needs to specify "a sender sending a message to a receiver at a specific time". To model this, two aspects are essential: (1) correlation between senders and receivers, and (2) temporal constraints upon message sending. The first point aims to establish the "channels" for senders and receivers for communication and the second point is to specify when to communicate.

With the above observation, in our approach, to design a choreography language, a correlation should be defined. Fig. 6.2 shows a correlation diagram among the four types of artifacts (shown in boxes). Exactly one primary artifact (shown in bold box) is

required in a diagram; it represents the lifespan for a collaborative process. Naturally, the starting of a primary artifact denotes the starting of a collaborative process. The edges (with labeled cardinality) denote the correlation relationships among artifacts. For example, an `Order` instance may create (with an arrow) multiple `Purchase` instances, as several vendors may be involved in one shopping cart. While the relationship between `Order` and `Payment` (edge without arrows, denoting the correlation relationship between them is set up by some mechanism other than creation) is one-to-one, since the online store may receive exactly one check from the bank for one `Order` instance.

In addition to the correlations that are explicitly shown in a correlation diagram, two artifacts without an edge in between could also be correlated through "derivation". For example, if a customer places an order with three items, say item 1, 2, and 3, which are provided by three different vendors, then three corresponding `Purchase` instances will be created, say `Purchase` 1, 2, and 3 respectively. Meanwhile, suppose that item 1 and 3 are stored in warehouse A and item 2 is stored in warehouse B; then two `Fulfillment` instances will be created, say `Fulfillment` A and B. In this case, naturally, `Purchase` 1 and 3 will be correlated with `Fulfillment` A and `Purchase` 2 will be correlated with `Fulfillment` B. This kind of correlations is implicitly "derived". Thus the proposed choreography language should also be able to capture the "derived" correlations.

Once the correlations are defined, a choreography is needed to specify how and when an artifact can communicate with its correlated artifacts by sending messages. Continue with the above example; Fig. 6.3 shows the message diagram that represents sender-receiver relationships. Commonly, the messages cannot be sent in arbitrary orders, some temporal constraints are needed. The remainder of this paragraph illustrates one possible message sending sequence: The primary artifact `Order` instance is initiated when receiving the "Order Request" Message (`OR`) from a customer. If the order is placed, an invoice (`IV`) is sent to her. With the invoice, the customer may initiate a `Payment` instance by sending

a "Payment Request" message (`PR`) to the bank. Once verified, the `Payment` instance will send a check (`CH`) to the correlated `Order` instance. Meanwhile, the `Order` instance will create all the correlated `Purchase` instances by sending "Create Purchase" messages (`CP`) and all the correlated `Fulfillment` instances by sending "Create Fulfillment" messages (`CF`). A package will be delivered from a warehouse, if a `Fulfillment` instance receives the "Purchase Complete" messages (`PC`) from the correlated `Purchase` instances and a "Ready to Ship" message (`RS`) from the `Order` instance. Once a package is delivered to the customer, a shipping confirmation (`CS`) will be sent to the `Order` instance by a `Fulfillment` instance. If the `Order` instance receives all the correlated confirmations, it will finalize the process by sending the customer a "Order Complete" message (`OC`).

Traditionally, the temporal constraints that work on the type level (i.e. only one instance for each participant type during the execution) can be captured by graphs (e.g. Petri-nets [43] or automata [58]). However, in practice, as some business processes may have multiple instances involved and usually the number of instances is unknown during the design time, graphs are unable to capture the instance-level information. Thus in our proposed choreography language, temporal constraints are expressed by rules in a declarative way. The rules are based on DecSerFlow [54] and first order linear temporal logic.

6.2 A Choreography Language

This section introduces a declarative language for defining choreographies. In this language, a choreography assumes participant business processes are modeled as artifacts [2] and consists of correlations between artifacts and instances, messages, and a set of choreography constraints (i.e. temporal constraints).

The language has five main components, namely, "artifact declaration", "correlation

declaration", "derived correlation declaration", "message declaration", and "choreography constraints". All the components will be introduced in the following subsections.

6.2.1 Artifact and correlation declaration

Artifacts represent participant business processes, the notion of an "artifact" captures the "visible" data contents for choreography specification.

To formally define the data contents in an artifact, the concept of "data types" is needed. For technical development, we define "*primitive* (*data*) *types*" as scalars that includes boolean, numeric, and character types. Some common primitive types in most programming languages include strings, integers, boolean values, and float numbers.

Comparing with the primitive types, some data types could be hierarchically structured. Thus we need the notion of "complex attributes" that has already been introduced in Section 2.2, where each "leaf" attribute should be of a primitive type.

The following definition introduces "artifact interfaces", which is a key- or local-key-free version of "business entities" in Section 2.2. The reason we do not restrict a business entity to have keys in this chapter is to allow a more general space of business entity enactments.

Definition: An *artifact* (*interface*) `A` is a tuple (ν, π), where ν is the *name* of `A` and π is a complex attribute called the "(*visible*) *business entity*".

In general, an artifact (which represents a business process) needs to expose some but not necessary all of its data content (i.e. only the "visible" data contents) for global referencing. In some literatures [2, 15, 16], not only the business entity of an artifact is specified but also the "lifecycle model", which describes the "execution" schema of an artifact. However, as the internal states are not visible to the global choreography, it is not necessary to include a lifecycle model for an artifact in the current proposed

choreography language. Thus, the language is rather general as the participants can choose to use different lifecycle models.

Without loss of generality, each artifact should always contain a top-level and non-set-typed attribute "ID" (in its information model) to hold a unique identifier for each "artifact enactment" (i.e., an instance for an artifact at a specific time stamp). The reason we use term "enactment" instead of "instance" is explained in Remark 2.2.2.

Example 6.2.1 Fig. 6.1 shows the graphic representation for the `Order` artifact described in Section 6.1. The name "Order" is shown in the rectangle. Within the artifact, the top-level attributes include "ID" and "Cart". For "Cart", it is of a set type, which includes "Inv_ID", "Name", "Quan", and "Price". All the four child attributes of "Cart" are of primitive types. ∎

Given an artifact `A`, an "enactment" of `A` is similar to the concept of a "value" to a complex attribute in Section 2.2.

In the technical discussion, we assume that for each artifact, there is a countably infinite set of artifact enactment IDs; furthermore, these ID sets are pairwise disjoint. Let $\mathbf{ID}_{\mathrm{A}}$ be the union of all artifact instance ID sets.

Definition: Given an artifact $\mathtt{A} = (\nu, \pi)$, an (*artifact*) *enactment* of `A` is a pair (id, μ), where $id \in \mathbf{ID}_{\mathrm{A}}$ and μ is a value of π with attribute ID taking value id.

Given an artifact enactment I, denote $\mathrm{ID}(I)$ to be the value of the ID attribute of I. Given a set of artifact enactment $\mathbb{T}$, denote $\mathrm{ID}(\mathbb{T})$ to be the set of the IDs of all the artifacts in $\mathbb{T}$.

We now define an important notion of a "correlation graph". Intuitively, such a graph specifies whether instances of two business processes (i.e. artifacts) are correlated and whether the correlation is one instance of a business process correlating to one or many

instances of the other business process. Similar to WS-CDL [57], only a pair of correlated instances may exchange messages in our model.

Given a binary relation C, denote $C^{\mathbf{r}}$ as a binary relation such that $(u, v) \in C^{\mathbf{r}}$ if and only if (v, u) is in C.

Definition: A *correlation graph* G is a tuple (V, ρ, E, C, λ), where

- V is a set of artifacts, whose cardinality is greater than 1. We may call artifacts in V "nodes" (of the graph),
- $\rho \in V$ is the *primary* artifact (the root),
- $E \subseteq V \times V$ is symmetric denoting correlations (undirected edges) among artifacts that contains no cycle,
- C is asymmetric denoting creation relationships among artifacts such that
  - $C \cap E = \varnothing$,
  - graph $(V, E \cup C \cup C^{\mathbf{r}})$ is acyclic,
  - there is no $v \in V$ such that $(v, \rho) \in C$ (primary instances can only be created by external messages), and
  - for each $v \in V - \{\rho\}$, there is a sequence of edges $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n) \in C \cup E$, such that $v_1 = \rho$ and $v_n = v$ (the graph is connected from the root),
- λ is a partial mapping from $(E \cup C) \times V$ to $\{1, \mathrm{m}\}$ (cardinality of correlations) such that
  - $\lambda((u, v), v') \in \{1, \mathrm{m}\}$ if v' is an end node for $(u, v) \in E$,
  - for each $(\rho, v) \in C \cup E$, $\lambda((\rho, v), \rho) = 1$ (single primary instance),
  - for each $(u, v) \in C$, $\lambda((u, v), u)$ is 1 (no multiple creation), and
  - for each $(u, v) \in E$, $\lambda((u, v), u) = \lambda((v, u), u)$ (consistency on undirected edges).

Intuitively, a correlation graph models correlations among artifacts and artifact enactments. More precisely, if two artifacts are correlated (connected by an edge), it indicates that some enactments of these two artifacts are correlated. Given a correlation graph (V, ρ, E, C, λ), the mapping λ indicates the type of cardinality of enactments (1-to-1, 1-to-$m$any, m-to-1, m-to-m). Notice that a correlation graph is essentially a tree-like structure rooted by the primary artifact. In the next subsection, the "derived correlation" will be introduced to allow more correlations among artifacts.

Example 6.2.2 Fig. 6.2 shows an correlation diagram, in which there are four artifacts: `Order` (the primary one, whose information model is given in Example 6.2.1), `Payment`, `Purchase`, and `Fulfillment`. The correlation between `Order` and `Payment` is an undirected edge denoting the correlation is set up externally. While the correlations between `Order` and `Purchase` as well as `Order` and `Fulfillment` are of creation relationship to denote that an `Order` can create multiple `Purchase` and `Fulfillment` enactments according to the cardinality. ∎

In some cases, the cardinality constraints might contradict with each other, which needs more restrictions to pretend an undesired design in a correlation graph.

Example 6.2.3 Continue with Example 6.2.2; suppose in Fig. 6.2, there is an extra artifact, named "`A`" and there is an undirected edge between `Payment` and `A`. Suppose the cardinality of this edge is m on the `Payment` end and 1 on the `A` end. Then, this design is allowed according to the definition of correlation graphs. However, Since there could exist at most 1 `Order` and 1 `Payment` instances in a running collaborative process, the cardinality on the `Payment` end cannot be m. ∎

Algorithm 2 Normalization of Correlation Graph

Input: correlation graph $G = (V, \rho, E, C, \lambda)$
Output: mapping η^{G}

```
1: η^G(ρ) := 1
2: Let Q be an empty queue
3: Push ρ into Q
4: while Q is not empty do
5:    Pop node v from Q
6:    for each (v, u) ∈ E ∪ C, where u ∈ V do
7:       if λ((v, u), v) = 1 and λ((v, u), u) = 1 and η^G(v) = 1 then
8:          η^G(u) := 1
9:       else
10:         η^G(u) := m
11:      end if
12:      Push u to Q
13:   end for
14: end while
15: return η^G
```

To prevent the inconsistency on cardinality, a breadth-first search can be applied on a correlation graph and propagate the cardinality from the root to each other artifact.

Given a correlation graph $G = (V, \rho, E, C, \lambda)$, denote η^{G} as a total mapping from V to $\{1, m\}$, such that for each $v \in V$, $\eta^{\mathrm{G}}(v)$ is assigned according to the *normalization procedure*, which is shown in Alg. 2.

In Alg. 2, the procedure initially marks the root with 1. During the bread-first search, if the current node v is marked with 1, and its outgoing edge (v, u) is "1-to-1", then the other end node u should be marked with 1 as well to denote that u can have at most one artifact enactment during the execution. Otherwise, u should be marked with m.

Definition: A *normalized correlation graph* is a correlation graph (V, ρ, E, C, λ) such that for each $v \in V$ if $\eta^{\mathrm{G}}(v) = 1$, then for each edge $e \in E \cup C$ that contains v as an end node, $\lambda(e, v) = 1$.

A normalized correlation graph restricts the consistency on cardinality. The graph in Fig. 6.2 is a normalized correlation graph.

Without loss of generality, in the remainder of this chapter, we assume that the given correlation graphs are normalized.

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$ a (*non-derived*) *collaboration instance* (of G) is a pair $(\mathbb{T}, \mathbf{Corr})$, where $\mathbb{T}$ is a set of artifact enactment, whose IDs are pairwise distinct and $\mathbf{Corr} \subset \text{ID}(\mathbb{T}) \times \text{ID}(\mathbb{T})$ that satisfies

- For each $I \in \mathbb{T}$, I is an enactment of some artifact in V,
- There is exactly one enactment of ρ in $\mathbb{T}$,
- For each $v \in V$, if $\eta^{\text{G}}(v) = 1$, then the number of enactments (in $\mathbb{T}$) of v is at most 1,
- Given two distinct enactments I_u and I_v (in $\mathbb{T}$) of artifact u and v (resp.), $(\text{ID}(I_u), \text{ID}(I_v))$ can be in **Corr** only if $(u, v) \in E \cup C \cup C^{\mathbf{r}}$,
- **Corr** is symmetric, and
- Graph $(\mathbb{T}, \mathbf{Corr})$ is connected.

Essentially, a collaboration instance contains all the running artifact enactments in a collaborative business process execution and their correlations.

Given $(\mathbb{T}, \mathbf{Corr})$ as a collaboration instance and two artifact enactments $I_1, I_2 \in \mathbb{T}$, I_1 has a *direct correlation* with I_2, if $(\text{ID}(I_1), \text{ID}(I_2)) \in \mathbf{Corr}$.

6.2.2 Derived correlation

In addition to correlations specified in a correlation graph, there may be correlations that are "derived" from existing correlations. (One possible scenario is shown in Section 6.1).

Intuitively, two artifact enactments that have no direct correlation can have a "derived correlation" if some rules are satisfied. For example, a rule could be that a `Purchase` and a `Fulfillment` enactments are correlated with each other if they have one item in common.

In the proposed choreography language, rules are built upon the existing correlations (either direct or derived). Thus, whenever a correlation is established, the following derived correlations can use the previous existing correlations as "bridges" to define new rules.

To specify such rules and derived correlations, some concepts upon an extended version of correlation graphs are needed, which assume that some derived correlations have been established, so that the new derived correlations can be built upon them.

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$, an *extended correlation graph* is a pair (G, D), where $D \subseteq V \times V$ is a symmetric relation and $D \cap (E \cup C \cup C^{\mathbf{r}}) = \varnothing$.

Given an extended correlation graph (G, D), in addition to the correlations specified in G, an extra edge set D is introduced to capture the "derived correlations". The detailed definition of "derived correlations" will be introduced later in this subsection.

Definition: Given an extended correlation graph G, an *extended collaboration instance* is a tuple $(\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$, where pair $(\mathbb{T}, \mathbf{Corr})$ forms a collaboration instance of G and $\mathbf{DCorr} \subseteq \text{ID}(\mathbb{T}) \times \text{ID}(\mathbb{T})$, such that

- **DCorr** is symmetric, and
- For each pair of artifacts $\mathtt{A}_1, \mathtt{A}_2$ in G, suppose I_1 and I_2 are the enactments of $\mathtt{A}_1$ and $\mathtt{A}_2$ (resp.); $(\text{ID}(I_1), \text{ID}(I_2))$ can be in **DCorr**, only if $(\mathtt{A}_1, \mathtt{A}_2) \in D$.

An extended correlation graph assumes that some "derived correlations" among instances have been established. The detailed semantics of how two instances can have a "derived correlation" will be defined later in this subsection.

We now introduce the important notions of "correlation references", "dot expressions", "path expressions", "atomic conditions", and "correlation rules" that are used to define the "derived correlations".

Definition: Given an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$, a *correlation reference* (with respect to G) is of form "$\mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n$", where $n \in \mathbb{N}^+$, for each $i \in [1..n]$, $\mathtt{A}_i \in V$, and for each $i \in [1..(n-1)]$, $(\mathtt{A}_i, \mathtt{A}_{i+1}) \in E \cup C \cup C^{\mathbf{r}} \cup D$.

Given an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$, a correlation reference is used to access one artifact from another through a "path" of referencing. Intuitively, this "path" should be built upon a chain of edges, where the correlations have been established.

Given a correlation reference $\mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n$, an extended collaboration instance $\mathbb{I}$, and an artifact enactment I of $\mathtt{A}_1$ in $\mathbb{I}$, function $\textsc{CorrInst}(\mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n, I, \mathbb{I})$ returns all the correlated artifact enactments of $\mathtt{A}_n$ from I through reference $\mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n$. The formal definition of $\textsc{CorrInst}(\mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n, I, \mathbb{I})$ is given below.

$$\textsc{CorrInst}(\mathtt{A}_1 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n, I, \mathbb{I} = (\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})) = \begin{cases} \{I\} & (n = 1) \\ \{I' \mid \exists I'' \in \textsc{CorrInst}(\mathtt{A}_1 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_{n-1}, I, \mathbb{I}) & \\ \quad \wedge (\textsc{id}(I''), \textsc{id}(I')) \in \mathbf{Corr} \cup \mathbf{DCorr}\} & (n \geqslant 2) \end{cases}$$

Example 6.2.4 Continuing with Example 6.2.2, suppose the current extended correlation graph is Fig. 6.2 and is without a derived correlation specified. In addition, suppose a collaboration instance $\mathbb{I}$ is with two `Purchase` enactments I_1^{p} and I_2^{p} correlated with an `Order` instance I^{o}; then $\textsc{CorrInst}(\mathtt{Purchase} \blacktriangleright \mathtt{Order}, I_1^{\mathrm{p}}, \mathbb{I})$ will return $\{I^{\mathrm{o}}\}$ and $\textsc{CorrInst}(\mathtt{Order} \blacktriangleright \mathtt{Purchase}, I^{\mathrm{o}}, \mathbb{I})$ will return $\{I_1^{\mathrm{p}}, I_2^{\mathrm{p}}\}$. ∎

Given a correlation reference $\mathit{ref} = \mathtt{A}_1 \blacktriangleright \mathtt{A}_2 \blacktriangleright \cdots \blacktriangleright \mathtt{A}_n$, where $n \in \mathbb{N}^+$, $\mathtt{A}_1$ is called the *source artifact* (denoted as $\textsc{Src}(\mathit{ref})$) and $\mathtt{A}_n$ is called the *target artifact* (denoted as $\textsc{Tar}(\mathit{ref})$).

Example 6.2.5 Continuing with Example 6.2.4, in terms of correlation reference `Order`► `Purchase`, the source artifact is `Order` and the target artifact is `Purchase`. ∎

Definition: Given a complex type τ, a *dot expression* (of τ) is of form "$a_1.a_2.\cdots.a_n$", where $n \in \mathbb{N}^+$, a_1 is a top-level attribute of τ, and each a_{i+1} ($i \in [1..(n-1)]$) is a child attribute of a_i.

A dot expression is used to access the hierarchical data for a given artifact enactment (which contains an element of a complex type). Given a complex attribute τ, a dot expression $a_1.a_2.\cdots.a_n$ of τ, and a value $\overline{v}$ of τ, function $\textsc{DotExp}(a_1.a_2.\cdots.a_n, \overline{v})$ returns the set of values of a_n based on $\overline{v}$. The formal definition of $\textsc{DotExp}(a_1.a_2.\cdots.a_n, \overline{v})$ is given below (in which, $\textsc{Val}(a, \overline{v})$ denotes the value of attribute a in v).

$$\textsc{DotExp}(a_1.\cdots.a_n, \overline{v}) = \begin{cases} \{\textsc{Val}(a_n, \overline{v})\} & (n = 1 \text{ and } a_n \text{ is not set-typed}) \\ \textsc{Val}(a_n, \overline{v}) & (n = 1 \text{ and } a_n \text{ is set-typed}) \\ \{\textsc{Val}(a_n, v) \mid v \in \textsc{DotExp}(a_1.\cdots.a_{n-1}, \overline{v})\} & (n \geqslant 2 \text{ and } a_n \text{ is not set-typed}) \\ \{v \mid \textsc{DotExp}(a_1.\cdots.a_{n-1}, \overline{v}) \; \exists v' \in \wedge v \in \textsc{Val}(a_n, v')\} & (n \geqslant 2 \text{ and } a_n \text{ is set-typed}) \end{cases}$$

Example 6.2.6 Continuing with Example 6.2.1, suppose I^{o} is an `Order` instance and the "Cart" attribute contains three values; then $\textsc{DotExp}(\text{Cart.Inv\_ID}, I^{\text{o}})$ will return all three inventory IDs in I^{o}. ∎

Definition: Given an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$ a *path expression* (with respect to G) is of form "*ref.dot*", where *ref* is a correlation reference (with respect to G) and *dot* is a dot expression of (the information model in) $\textsc{Tar}(ref)$.

Essentially, path expressions are used to access the hierarchical data in a set of correlated artifacts. Given a path expression *ref.dot*, an artifact enactment I of SRC(*ref*), and a collaboration instance $\mathbb{I}$, function PATHEXP(*ref.dot*, $I, \mathbb{I}$) returns the values of a_n in each correlated enactments from I through reference *ref*. The formal definition of PATHEXP(*ref.dot*, $I, \mathbb{I}$) is given below.

$$\text{PATHEXP}(\textit{ref.dot}, I, \mathbb{I}) = \{v \mid \exists \textit{id}, \exists \mu, (\textit{id}, \mu) \in \text{CORRINST}(\textit{ref}, I, \mathbb{I}) \wedge v \in \text{DOTEXP}(\textit{dot}, \mu)\}$$

Example 6.2.7 Continuing with Example 6.2.4 and 6.2.6, let $\mathbb{I}$ and I_1^{p} be as stated in Example 6.2.4. PATHEXP(`Purchase`►`Order`.Cart.Inv_ID, $I_1^{\text{p}}, \mathbb{I}$) will return all the inventory IDs in all the correlated `Order` instances of I_1^{p} in $\mathbb{I}$. ∎

In order to manipulate on the values (obtained from function "PATHEXP"), some operators and quantifiers are needed (which are shown in the list below).

- Operators: "=", "≠", ">", "<", "⩾", "⩽", and "⊓"
- Quantifiers: "SOME" and "ALL"

For operators, "=", "≠", ">", "<", "⩾", and "⩽" are used to compare numbers or strings (in alphabetical order) in a natural manner; while "⊓" is binary operator associated with two operands in form of sets. Given two sets A and B, $A \sqcap B$ is "valid" (or "true") if A and B have at least one element in common.

For quantifiers, "SOME" and "ALL" are all associated with a single set. Given a set A, SOME(A) denotes "there exists an element in A"; while ALL(A) means "for all elements in A".

Given a path expression $exp = ref.a_1.a_2.\cdots.a_n$, the *type* of exp is the data type of attribute a_n. Given a primitive type τ and a set $S = \{v_1, v_2, ..., v_n\}$, where for each $i \in [1..n]$, v_i is in the domain of τ, define the *type* of S to be τ.

Definition: Given an extended correlation graph G, an *atomic condition* (with respect to G) is of form "$t_1 \theta t_2$" where either

- Case 1:
  - for each $i \in \{1, 2\}$, t_i is a path expression (with respect to G) or a set of values of the same primitive type,
  - t_1 and t_2 agree on the same type, and
  - θ is "$\sqcap$"; or
- Case 2:
  - for each $i \in \{1, 2\}$, t_i is a value of a some primitive type or of form "$f(exp)$", where f ranges over $\{$ SOME, ALL $\}$
    and exp is a path expression (with respect to G),
  - t_1 and t_2 agree on the same type, and
  - θ ranges over $\{=, \neq, >, <, \geqslant, \leqslant\}$.

Given an atomic condition φ, the source artifacts occurring in the correlation references in φ are called the *candidate* artifacts of φ.

Example 6.2.8 Continue with Example 6.2.2; suppose the current extended correlation graph is Fig. 6.2 and without a derived correlation specified. And suppose both `Purchase` and `Fulfillment` artifacts have a top-level attribute called "Item" (which is of a set type) and both the "Items" in `Purchase` and `Fulfillment` have a child attribute called

"Inventory_ID". Then an atomic condition to specify that `Purchase` and `Fulfillment` should have at least one item in common could be

$$\texttt{Purchase}.\text{Item.Inventory\_ID} \sqcap \texttt{Fulfillment}.\text{Item.Inventory\_ID}$$

The candidate artifacts of this atomic condition are `Purchase` and `Fulfillment`. ∎

Given (1) an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$, (2) an extended collaboration instance $\mathbb{I} = (\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$ of G, (3) an atomic condition $\varphi = t_1 \theta t_2$ with respect to G, (4) two artifacts $\mathtt{A}_1, \mathtt{A}_2 \in \mathbb{V}$, such that the candidate artifacts of φ are in $\{\mathtt{A}_1, \mathtt{A}_2\}$, and (5) two artifact enactments I_1 and I_2 (in $\mathbb{T}$) of $\mathtt{A}_1$ and $\mathtt{A}_2$ respectively, φ is *valid* with respect to I_1, I_2, and $\mathbb{I}$ if for each path expression *exp*, whose source artifact is $\mathtt{A}_i$ ($i \in \{1, 2\}$), then φ is true after replacing each *exp* by PathExp($exp, I_i, \mathbb{I}$).

Example 6.2.9 Continuing with Example 6.2.8, Suppose the current collaboration instance $\mathbb{I}$ contains two `Purchase` instances I_1^{p}, I_2^{p} and two `Fulfillment` instances I_1^{f}, I_2^{f}, in which, I_1^{p} has inventory_ID 3, I_2^{p} has inventory_IDs 1 and 2, I_1^{f} has inventory_ID 2, and I_2^{f} has inventory_IDs 1 and 3. Then the atomic condition (in Example 6.2.8) "`Purchase`.Item.Inventory_ID ⊓ `Fulfillment`.Item.Inventory_ID" is valid with respect to I_1^{p}, I_2^{f}, and $\mathbb{I}$, because the result of PathExp(`Purchase`.Item.Inventory_ID, $I_{\mathrm{p}}^1, \mathbb{I}$) is $\{3\}$, the result of PathExp(`Fulfillment`.Item.Inventory_ID, $I_2^{\mathrm{f}}, \mathbb{I}$) is $\{1, 3\}$, and $\{3\} \sqcap \{1, 3\}$ is true. However, the same atomic condition is not valid with respect to I_{p}^1, I_{f}^1, and $\mathbb{I}$, as PathExp(`Purchase`.Item.Inventory_ID, $I_{\mathrm{p}}^1, \mathbb{I}$) will return $\{3\}$, PathExp(`Fulfillment`.Item.Inventory_ID, $F_1, \mathbb{I}$) will return $\{2\}$, and $\{3\} \sqcap \{2\}$ is false. ∎

Definition: Given an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$ and two artifacts $\mathtt{A}_1, \mathtt{A}_2 \in V$ where $(\mathtt{A}_1, \mathtt{A}_2) \notin E \cup C \cup C^{\mathbf{r}}$, a *correlation rule* of $\mathtt{A}_1$ and $\mathtt{A}_2$ (with respect to G) is of form "COR($\mathtt{A}_1, \mathtt{A}_2$): c", where c is a a set (conjunction) of atomic conditions with respect to G, such that for each $\varphi \in c$, each candidate artifact of φ is in $\{\mathtt{A}_1, \mathtt{A}_2\}$.

Example 6.2.10 Continuing with Example 6.2.8, a correlation rule of `Fulfillment` and `Purchase` can be as follows.

$$\textsc{cor}(\texttt{Purchase}, \texttt{Fulfillment}):$$
$$\texttt{Purchase}.\text{Item.Inventory\_ID} \sqcap \texttt{Fulfillment}.\text{Item.Inventory\_ID}$$

to denote that a `Purchase` enactment is correlated with a `Fulfillment` enactment if they share the same inventory. ∎

Given (1) an extended correlation graph $G = ((V, \rho, E, C, \lambda), D)$, (2) a collaboration instance $\mathbb{I} = (\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$ of G, (3) a correlation rule $r = \textsc{cor}(\mathtt{A}_1, \mathtt{A}_2)$: c of $\mathtt{A}_1$ and $\mathtt{A}_2$ in V with respect to G, and (4) two artifact enactments I_1 and I_2 (in $\mathbb{T}$) of $\mathtt{A}_1$ and $\mathtt{A}_2$ respectively, I_1 has a *derived correlation* with I_2 (with respect to $\mathbb{I}$ and r) if for each atomic condition $\varphi \in c$, φ is valid with respect to I_1, I_2, and $\mathbb{I}$.

Example 6.2.11 Continuing with Example 6.2.9 and 6.2.10, let $\mathbb{I}$, I_1^{p}, I_1^{f}, and I_2^{f} be as stated in Example 6.2.9 and r be the correlation rule stated in Example 6.2.10. Then I_1^{p} has a derived correlation with I_2^{f} with respect to $\mathbb{I}$ and r; but not for I_1^{p} and I_1^{f}. ∎

Notice that derived correlations do not have specified cardinality constraints.

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$, the *correlation rule set* Γ (of G) is a totally ordered set of correlation rules (where the order is called the "*dependency order*"): $(r_1 = \textsc{cor}(\mathtt{A}_1, \mathtt{B}_1)\text{: } c_1), (r_2 = \textsc{cor}(\mathtt{A}_2, \mathtt{B}_2)\text{: } c_2), ..., (r_{|\Gamma|} = \textsc{cor}(\mathtt{A}_{|\Gamma|}, \mathtt{B}_{|\Gamma|})\text{: } c_{|\Gamma|})$, such that

- for each $i \in [1..|\Gamma|]$, $\mathtt{A}_i \neq \mathtt{B}_i$ and $\mathtt{A}_i, \mathtt{B}_i \in V$,
- for each distinct $i, j \in [1..|\Gamma|]$, $\{(\mathtt{A}_i, \mathtt{B}_i), (\mathtt{B}_i, \mathtt{A}_i)\} \cap \{(\mathtt{A}_j, \mathtt{B}_j), (\mathtt{B}_j, \mathtt{A}_j)\} = \varnothing$,

Algorithm 3 Build Derived Correlation

Input: correlation graph G, correlation instance $(\mathbb{T}, \mathbf{Corr})$ of G, correlation rule set Γ of G
Output: a binary relation **DCorr**

```
1: set DCorr := ∅
2: for each r of artifacts A_1 and A_2 in Γ in dependency order do
3:    for each I_1, I_2 ∈ 𝕋 do
4:       if I_1 has a derived correlation with I_2 with respect to (𝕋, Corr, DCorr) and r
         then
5:          DCorr := DCorr ∪ {(ID(I_1), ID(I_2)), (ID(I_2), ID(I_1))}
6:       end if
7:    end for
8: end for
9: return result
```

- for each $i \in [1..|\Gamma|]$, $(E \cup C \cup C^{\mathbf{r}}) \cap \{(\mathtt{A}_i, \mathtt{B}_i), (\mathtt{B}_i, \mathtt{A}_i)\} = \varnothing$, and
- for each $i \in [1..|\Gamma|]$, r_i is with respect to $(G, \bigcup_{j=1}^{i-1}\{(A_j, B_j), (B_j, A_j)\})$, which is an extended correlation graph.

Essentially, the correlation rule set restricts that for each correlation rule r, the path expressions of r can only use the artifact references that have been established.

Given a correlation graph G, a correlation instance $\mathbb{I} = (\mathbb{T}, \mathbf{Corr})$ of G, and a correlation rule set Γ of G, all the derived correlations can be built based on Alg. 3, which is essentially to compute all the possible derived correlations in $\mathbb{I}$.

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$ and the correlation rule set Γ of G, a *collaboration instance* (*with derivation*) (with respect to G and Γ) is an extended correlation instance $(\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$, such that **DCorr** is built based on Alg. 3 with inputs G, $(\mathbb{T}, \mathbf{Corr})$, and Γ.

6.2.3 Message declaration

With the correlations defined, messages can be sent between two correlation artifact enactments. This subsection describes the message types and instances.

Without loss of generality, assume there always exists an artifact with name "ext" with empty business entity to denote the the external environment (as the sender or receiver); further, the "artifact enactment" of "ext" is with ID "ext".

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$ and a correlation rule set Γ (of G), a *message type* M (with respect to G and Γ) is a tuple $(\nu, \mathtt{A}_\mathrm{s}, \mathtt{A}_\mathrm{r}, \pi, \tau, min, max)$, where

- ν is the name of M,
- $\mathtt{A}_\mathrm{s}, \mathtt{A}_\mathrm{r} \in V \cup \{\text{"ext"}\}$ are distinct artifacts denoting the sender and receiver (resp.) such that at most one of them can be "ext", and if both are in V, they must be correlated (via an edge in G or by a correlation rule in Γ),
- π is a complex data type, called "*payload*";
- τ is "+" (creation, i.e. the sending enactment creates an enactment of the receiving artifact upon arrival of each message instance) or "−" (no creation); τ can be "+" only if
  - $(\mathtt{A}_\mathrm{s}, \mathtt{A}_\mathrm{r}) \in C$, or
  - $\mathtt{A}_\mathrm{s}$ is "ext" and there does not exist $\mathtt{A}'_\mathrm{s} \in V$, such that $(\mathtt{A}'_\mathrm{s}, \mathtt{A}_\mathrm{r}) \in C$,
- $min \in \mathbb{N}$ and $max \in \mathbb{N} \cup \{\infty\}$ (where "∞" denotes infinity) is the minimum and maximum number of message instances (resp.) that can be sent from an enactment of $\mathtt{A}_\mathrm{s}$; $max = 1$ if τ is "+" and $\eta^\mathrm{G}(\mathtt{A}_\mathrm{r}) = 1$.

Without loss of generality, each message type should always contain a top-level and non-set-typed attribute "ID" (in its payload) to hold a unique identifier for each message instance, which is defined later.

Fig. 6.3 shows a message diagram, each edge represents a message type with the edge direction indicates the message flow.

Example 6.2.12 Continuing with the example in Section 6.1, a "Create Purchase" message (`CP`) can be formalized as "(CP, `Order`, `Purchase`, (OrderID, Amount,...), +, 1, ∞)", which denotes that it is a message type from `Order` to `Purchase`. The "+" symbol indicates that a new receiving instance will be created by each arriving message. The message payload includes "OrderID", "Amount", etc. The minimum number of messages can be sent from an `Order` instance is 1. Similarly, an "Order Complete" message (`OC`) from `Order` to the external environment can be defined as "(OC, `Order`, ext, (OrderID,...), −, 1, 1)". ∎

In the technical discussion, we assume that for each message type, there is a countably infinite set of message instance IDs; furthermore, these ID sets are pairwise disjoint. Let $\mathbf{ID}_{\mathrm{M}}$ be the union of all message instance ID sets.

Definition: A *message instance* of a message type $\mathsf{M} = (\nu, \mathtt{A}_{\mathrm{s}}, \mathtt{A}_{\mathrm{r}}, \pi, \tau, min, max)$ is a tuple $(id, id_{\mathrm{s}}, id_{\mathrm{r}}, \mu)$, where $id_{\mathrm{s}}, id_{\mathrm{r}} \in \mathbf{ID}_{\mathrm{A}}$ are the ID values of enactments of $\mathtt{A}_{\mathrm{s}}$ and $\mathtt{A}_{\mathrm{r}}$ (resp.) such that if $\mathtt{A}_{\mathrm{s}}$ ($\mathtt{A}_{\mathrm{r}}$) is "ext", id_{s} (resp. id_{r}) is also "ext", μ is an element of π, and $id \in \mathbf{ID}_{\mathrm{M}}$ is the value for attribute ID in μ.

Given an message instance I, denote ID(I) to be the value of the ID attribute of I.

Definition: A *collaboration schema* is a tuple (G, Γ, Msg), where

- $G = (V, \rho, E, C, \lambda)$ is a correlation graph,
- Γ is the correlation rule set of G, and
- *Msg* is a set of message types with respect to G and Γ, such that
  - for each distinct artifacts $\mathtt{A}_1, \mathtt{A}_2 \in V$, if $(\mathtt{A}_1, \mathtt{A}_2) \in C$, then there should have a corresponding creation message in *Msg* from $\mathtt{A}_1$ to $\mathtt{A}_2$, and
  - for each artifact $\mathtt{A} \in V$, if there does not exist another artifact $\mathtt{A}' \in V$, such that $(\mathtt{A}', \mathtt{A}) \in C$, then there should have a corresponding creation message in *Msg* from "ext" to $\mathtt{A}$.

Roughly, a collaboration schema defines the correlations among artifacts (participant types) and instances (participants), and the message types.

6.2.4 Choreography constraints

In this subsection we define the notion of "choreography constraints", which state temporal properties on message occurrences and may also contain conditions on data in related artifact enactments and the messages.

As the "choreography constraints" are temporal, we need to define "choreography states" that represent snapshots at time instants.

Definition: Given a collaboration schema $C = (G, \Gamma, \mathit{Msg})$, a *choreography* (*c-*)*state* of C is a tuple $(\mathbb{I}, m, \overline{\mathrm{M}}, \mathbf{MA}, \mathbf{MM})$, where

- $\mathbb{I} = (\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$ is a collaboration instance with respect to G and Γ,
- $m = (\mathit{id}, \mathit{id}_{\mathrm{s}}, \mathit{id}_{\mathrm{r}}, \mu)$ is a message instance of message type $\mathsf{M} \in \mathit{Msg}$,
- $\overline{\mathrm{M}}$ a finite set of message IDs to denote the messages that have been sent so far,
- $\mathbf{MA} \subseteq \overline{\mathrm{M}} \times (\mathrm{ID}(\mathbb{T}) \cup \{\mathit{id}_{\mathrm{r}}\})$ is a *message-artifact dependency* set, and
- $\mathbf{MM} \subseteq (\overline{\mathrm{M}})^2$ is an irreflexive *message-message dependency* set.

such that

(1) $\mathit{id} \in \overline{\mathrm{M}}$,

(2) if id_{s} is not "ext", the instance of id_{s} is in $\mathbb{T}$,

(3) if M is creation, then the instance of id_{r} is not in $\mathbb{T}$ and $(\mathit{id}, \mathit{id}_{\mathrm{r}})$ is in $\mathbf{MA}$,

(4) if M is not creation, then either id_{r} is "ext" or the instance of id_{r} is in $\mathbb{T}$,

(5) if neither id_{s} or id_{r} is "ext", then $(\mathit{id}_{\mathrm{s}}, \mathit{id}_{\mathrm{r}}) \in \mathbf{Corr} \cup \mathbf{DCorr}$, and

(6) the graphs $(\overline{\mathrm{M}} \cup \mathrm{ID}(\mathbb{T}) \cup \{\mathit{id}_{\mathrm{r}}\}, \mathbf{MA})$ and $(\overline{\mathrm{M}}, \mathbf{MM})$ encode functions (i.e. each node has $\leqslant$1 outgoing edge).

The message-artifact dependency set **MA** holds dependencies of an arriving message ID that causes creation of an artifact ID. The message dependency set **MM** represents the relationships between messages, e.g., one message may depend another based on contents, or simply request-response. For example, an invoice message may respond to an order request.

An c-state is a snapshot of artifact enactments (together with their correlations), past message IDs, the current message sent, message-artifact and message-message dependencies that have been established. Conditions (2)(4) demand that the sender and receiver are existing artifact enactments if not external for non-creation message types. Conditions (3)(5) concern correlations and dependencies. Finally condition (6) ensures that each message creates at most one artifact and/or depends on at most one message.

An c-state is *initial* if $\mathbb{I} = (\varnothing, \varnothing, \varnothing)$, m is from "ext" to the primary artifact, $\overline{\mathrm{M}}$ and **MA** are singleton sets, and $\mathbf{MM} = \varnothing$.

Example 6.2.13 Continue with the example in Section 6.1; suppose the system now only has one primary `Order` enactment I^{o} and one correlated `Payment` enactment I^{p} (from I^{o}). If a message instance $m_{\mathtt{CH}}$ of type `CH` ("check") is sent from I^{p} to I^{o}, then the system state at this moment can have the 2 (correlated) artifact enactments (I^{o} and I^{p}), the message instance $m_{\mathtt{CH}}$, and all the messages (IDs) that have been sent and all the dependencies that have been established till this moment. ∎

We now introduce, given a c-state, how it can proceed into another c-state by sending a message instance.

Definition: Given a collaboration schema $C = (G, \Gamma, \mathit{Msg})$ and two c-state σ, σ' of C, $\sigma' = ((\mathbb{T}', \mathbf{Corr}', \mathbf{DCorr}'), m', \overline{\mathrm{M}}', \mathbf{MA}', \mathbf{MM}')$ is a *successor* of $\sigma = ((\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr}), m, \overline{\mathrm{M}}, \mathbf{MA}, \mathbf{MM})$ if all the following conditions hold.

- $\text{ID}(m) \neq \text{ID}(m')$
- $\overline{\text{M}}' = \overline{\text{M}} \cup \{\text{ID}(m')\}$,
- $\mathbf{MM} \subseteq \mathbf{MM}'$; if $\mathbf{MM} \subset \mathbf{MM}'$, then $|\mathbf{MM}'| = |\mathbf{MM}| + 1$, and
- one of the two following conditions should hold:
  - if m is not creation, then $\text{ID}(\mathbb{T}') = \text{ID}(\mathbb{T})$, $\mathbf{Corr}' = \mathbf{Corr}$, and $\mathbf{MA}' = \mathbf{MA}$;
  - if m is creation, then $\text{ID}(\mathbb{T}') = \text{ID}(\mathbb{T}) \cup \{id_{\text{r}}\}$, $\mathbf{MA}' = \mathbf{MA} \cup (\text{ID}(m), id_{\text{r}})$, and if $id_{\text{s}} \in \text{ID}(\mathbb{T})$, then $\mathbf{Corr}' = \mathbf{Corr} \cup \{(id_{\text{s}}, id_{\text{r}}), (id_{\text{r}}, id_{\text{s}})\}$; otherwise, ($id_{\text{s}}$ is "ext"), there should exist $I \in \mathbb{T}$ of some artifact $\texttt{A}$ in G, such that $\texttt{A}$ has a direct correlation with the artifact of id_{r}, and $\mathbf{Corr}' = \mathbf{Corr} \cup \{\text{ID}(I), id_{\text{r}}), (id_{\text{r}}, \text{ID}(I)\}$.

Example 6.2.14 Continue with Example 6.2.13; suppose right after the time point described in Example 6.2.13, I^{o} sends a `CP` ("create purchase") message instance $m_{\texttt{CP}}$ to create a new `Purchase` instance I^{c}. Then the current system state have 3 artifact enactments (I^{o}, I^{p}, and I^{c}), the message instance $m_{\texttt{CP}}$, and all the sent messages and all the established dependencies. Moreover, the current c-state is a successor of the one in Example 6.2.13. Comparing with the c-state in Example 6.2.13, the message-artifact dependency set in the current c-state includes one more pair $(\text{ID}(m_{\texttt{CP}}), \text{ID}(I^{\text{c}}))$. ∎

Definition: Given a collaboration schema $C = (G, \Gamma, Msg)$, a *choreography* (*c-*)*behavior* of C is a finite sequence $\sigma_1\sigma_2\cdots\sigma_n$ of c-states of C such that

- σ_1 is initial,
- for each $i \in [1..(n-1)]$, σ_{i+1} is a successor of σ_i,
- messages in σ_i's have distinct IDs, and
- for each $\mathsf{M} = (\nu, \texttt{A}_{\text{s}}, \texttt{A}_{\text{r}}, \pi, \tau, min, max) \in Msg$, one of the two following conditions should hold:

- if $\mathtt{A}_\mathrm{s}$ is "ext", then the total number of message instances of message type M sent from "ext" in all σ_i's ($i \in [1..n]$) is in range $[min, max]$, or
- otherwise, for each artifact enactment I (in σ_n) of $\mathtt{A}_\mathrm{s}$, the total number of message instances of message type M sent from $\textsc{id}(I)$ in all σ_i's ($i \in [1..n]$) is in range $[min, max]$.

Inside each system state, artifact enactments and dependencies are persistent that will be kept in the following states; while message instances are instantaneous that will be consumed by their receivers and will not be kept in the following states.

Intuitively, an c-state advances by consuming the current message (instance) and producing the next message. If the receiving ID does not correspond to an artifact enactment, a new enactment is created. The changes of data contents of artifact enactments are the responsibility of participant processes and thus not captured in c-state transitions. Also, message-message dependency is not required, creating such dependencies is also done by individual participant business processes.

We now focus on "choreography constraints". Roughly, we apply (non-temporal) "message formulas" to c-states which examines message type and contents as well as the contents of sending/receiving artifact enactments. Each constraint then uses a temporal operator to connect two message formulas. Individual LTL operators are not expressive enough, therefore we use binary operators from DecSerFlow [54], which is set of templates built from LTL operators.

For technical development we assume there is an countably infinite set of *artifact variables* $\mathbf{V}_\mathrm{A}$ an countably infinite set of *message variables* $\mathbf{V}_\mathrm{M}$. Without loss of generality, assume $\mathbf{V}_\mathrm{A}$ contains variable "ext".

A (*variable*) *assignment* v is a total mapping from $\mathbf{V}_\mathrm{A} \cup \mathbf{V}_\mathrm{M}$ to $\mathbf{ID}_\mathrm{A} \cup \{\text{"ext"}\} \cup \mathbf{ID}_\mathrm{M}$, such that for each $x \in \mathbf{V}_\mathrm{A} - \{\text{"ext"}\}$, $v(x) \in \mathbf{ID}_\mathrm{A} \cup \{\text{"ext"}\}$, $v(\text{"ext"}) = \text{"ext"}$, and for

each $x \in \mathbf{V}_{\mathrm{M}}$, $v(x) \in \mathbf{ID}_{\mathrm{M}}$.

Definition: Given a collaboration schema $C = (G, \Gamma, \mathit{Msg})$, a *message predicate* (of C) is of form "$\textsc{Msg}(\mathsf{M}, z, x, y)$", where $\mathsf{M} \in \mathit{Msg}$, $z \in \mathbf{V}_{\mathrm{M}}$ to denote the message instance of M, and $x, y \in \mathbf{V}_{\mathrm{A}}$ to denote the sender and the receiver artifact enactment respectively.

Given a collaboration schema C, a c-state $\sigma = (\mathbb{I}, m, \overline{\mathrm{M}}, \mathbf{MA}, \mathbf{MM})$ of C, where $m = (\mathit{id}, \mathit{id}_{\mathrm{s}}, \mathit{id}_{\mathrm{r}}, \mu)$, a message predicate $\Phi = \textsc{Msg}(\mathsf{M}, z, x, y)$ of C, and an assignment v, Φ is *valid* with respect to σ and v (denoted as $\sigma \models_v \Phi$), if m is a message instance of M, $v(z) = \textsc{id}(m)$, $v(x) = \mathit{id}_{\mathrm{s}}$, and $v(y) = \mathit{id}_{\mathrm{r}}$. Otherwise, Φ is *invalid* with respect to σ and v (denoted as $\sigma \not\models_v \Phi$).

Example 6.2.15 Continuing with Example 6.2.14, let I^{o}, I^{c}, and $m_{\mathtt{CP}}$ be stated as in Example 6.2.14. Given an assignment v, the message predicate $\textsc{Msg}(\mathtt{CP}, z, x, y)$ checks if $v(x)$ sends message of $\mathtt{CP}$ with $\textsc{id}$ $v(z)$ to $v(y)$. $\textsc{Msg}(\mathtt{CP}, z, x, y)$ is valid with respect to v and the c-state in Example 6.2.13, if $v(x) = \textsc{id}(I^{\mathrm{o}})$, $v(y) = \textsc{id}(I^{\mathrm{c}})$, and $v(z) = \textsc{id}(m_{\mathtt{CP}})$. $\textsc{Msg}(\mathtt{CP}, z, x, y)$ is invalid with respect to the c-state in Example 6.2.13 and each possible assignment, because the message instance sent in Example 6.2.13 is of message type $\mathtt{CH}$. ∎

Definition: Given a message predicate $\Phi = \textsc{Msg}(\mathsf{M}, z, x, y)$, a *responding message predicate* (to Φ) is of form "$\textsc{Msg}[z](\mathsf{M}', z', x', y')$", where "$\textsc{Msg}(\mathsf{M}', z', x', y')$" forms a message predicate.

Given a collaboration schema C, a c-state $\sigma = (\mathbb{I}, m, \overline{\mathrm{M}}, \mathbf{MA}, \mathbf{MM})$ of C, where $m = (\mathit{id}, \mathit{id}_{\mathrm{s}}, \mathit{id}_{\mathrm{r}}, \mu)$, a responding message predicate $\varphi = \textsc{Msg}[z_2](\mathsf{M}', z_1, x, y)$ of C, and an assignment v, Φ is *valid* with respect to σ and v (denoted as $\sigma \models_v \Phi$), if $\sigma \models_v \textsc{Msg}(\mathsf{M}', z_1, x, y)$ and $(z_1, z_2) \in \mathbf{MM}$. Otherwise, Φ is *invalid* with respect to σ and v (denoted as $\sigma \not\models_v \Phi$).

Similar to a message predicate that is to check if a specific message instance has been sent in a c-state, a responding message predicate can perform the same check and also test if the current message instance is responding to same previous sent message instance.

Example 6.2.16 Continue with the example in Section 6.1; consider a restriction that if an `OR` ("order request") message instance $m_{\texttt{OR}}$ is sent (from "ext" to create a new `Order` instance I^o), then in the future, there should be an `OC` ("order complete") message instance (from I^o to "ext") responding to $m_{\texttt{OR}}$. This resection implies that, given an assignment v that maps artifact variable x to $\text{ID}(I^o)$, suppose message predicate $\text{MSG}(\texttt{OR}, z_{\texttt{OR}}, \text{ext}, x)$ is valid at the current c-state, then in the future, there should have another c-state that makes "$\text{MSG}[z_{\texttt{OR}}](\texttt{OC}, z_{\texttt{OC}}, x, \text{ext})$" valid. ∎

Definition: Given a collaboration schema $C = (G, \Gamma, Msg)$, a *variable path expression* (of C) is of form "$x.a_1.a_2.\cdots.a_n$", where either $x \in \mathbf{V}_{\text{A}} - \{\text{"ext"}\}$ and a_1 is a top-level attribute in an artifact in G, or $x \in \mathbf{V}_{\text{M}}$ and a_1 is a top-level attribute in a message type in Msg; in both cases, for each $i \in [1..(n-1)]$, a_{i+1} is a child attribute of a_i.

The concept of "variable path expressions" is similar to the one of "dot expressions" defined in Section 6.2.2. Essentially, both of them are used to access the hierarchical data structure. Thus, we may use function "DOTEXP" once more to define the semantics for "variable path expressions".

Given a collaboration schema C, a c-state $\sigma = ((\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr}), m, \overline{\text{M}}, \mathbf{MA}, \mathbf{MM})$ of C, where $m = (id_{\text{M}}, id_{\text{s}}, id_{\text{r}}, \mu_{\text{M}})$, and an assignment v, the *value* of a variable path expression $x.a_1.a_2.\cdots.a_n$ (with respect to σ and v) is either

- $\text{DOTEXP}(a_1.a_2.\cdots.a_n, \mu_{\text{M}})$, if $x \in \mathbf{V}_{\text{M}}$, $v(x) = \text{ID}(m)$, and a_1 is a top-level attribute in the message type of m,

- DotExp($a_1.a_2.\cdots.a_n, \mu_A$) (where μ_A is the element of the information model of some artifact enactment $I \in \mathbb{T}$), if $x \in \mathbf{V}_A - \{$"ext"$\}$, and a_1 is a top-level attribute in the artifact of I, or
- no value, otherwise.

Given a variable path expression $exp = x.a_1.a_2.\cdots.a_n$, the *type* of *exp* is the type of attribute a_n.

Definition: Given a collaboration schema $C = (G, \Gamma, Msg)$, a *data condition* (of C) is of form "$t_1 \theta t_2$" where either

- Case 1:
  - for each $i \in \{1, 2\}$, t_i is a variable path expression (of C) or a set of values of the same primitive type,
  - t_1 and t_2 agree on the same type, and
  - θ is "$\sqcap$"; or
- Case 2:
  - for each $i \in \{1, 2\}$, t_i is a value of a some primitive type or of form "$f(exp)$", where f ranges over {some, all} and *exp* is a variable path expression (of C),
  - t_1 and t_2 agree on the same type, and
  - θ ranges over $\{=, \neq, >, <, \geqslant, \leqslant\}$.

Given a collaboration schema C, a c-state σ of C, a data condition $\varphi = t_1 \theta t_2$ of C, and an assignment v, φ is *valid* (with respect to σ and v) (denoted as $\sigma \models_v \varphi$), if (1) the variable path expressions exp_1 and exp_2 of t_1 and t_2 have values v_1 and v_2 (resp.) with respect to σ and v, and (2) for each $i \in \{1, 2\}$, φ is true after placing each exp_i with v_i. Otherwise, φ is *invalid* with respect to σ and v (denoted as $\sigma \not\models_v \varphi$).

The concept and semantics of "data conditions" are similar to the ones of "atomic conditions" defined in Section 6.2.2.

Example 6.2.17 Continuing with Example 6.2.14, suppose the message type `CP` ("create purchase") has a top-level and non-set-typed attribute "cart"; inside "cart", there is a set-typed child attribute "item" and "item" has a non-set-typed attribute "price" of a primitive type (say "unsigned float"). Then given z as a message variable, data condition $\textsc{some}(z.\text{cart.item.price}) > 100$ is to check if there exists an item in the shopping cart in the current message instance that has a price greater than 100. ∎

Definition: Given a collaboration schema C, a *message formula* (of C) is of form "$\Phi \wedge (\bigwedge_{i=1}^{k} \varphi_i)$", where Φ is a message predicate $\textsc{Msg}(\mathsf{M}, z, x, y)$ of C, where $\mathsf{M} = (\nu, \mathtt{A}_\mathrm{s}, \mathtt{A}_\mathrm{r}, \pi, \tau, \mathit{min}, \mathit{max})$, and for each $i \in [1..k]$, φ_i is a data condition of C, such that for each variable path expression $\mathit{exp}_i = w.a_1.a_2.\cdots.a_n$ occurring in φ_i, exp_i should satisfy one of the following conditions.

- $w = z$ and a_1 is a top-level attribute in M,
- $w = x$ and a_1 is a top-level attribute in $\mathtt{A}_\mathrm{s}$, or
- $w = y$ and a_1 is a top-level attribute in $\mathtt{A}_\mathrm{r}$.

Given a collaboration schema C, a c-state σ of C, a message formula $\Psi = \Phi \wedge (\bigwedge_{i=1}^{k} \varphi_i)$ of C, and an assignment v, Ψ is *valid* (with respect to σ and v) (denoted as $\sigma \models_v \Psi$), if $\sigma \models_v \Phi$ and for each $i \in [1..k]$, $\sigma \models_v \varphi_i$. Otherwise, Ψ is *invalid* with respect to σ and v (denoted as $\sigma \not\models_v \Psi$).

Example 6.2.18 Continuing with Example 6.2.15 and 6.2.17, given an assignment v the message formula "$\textsc{Msg}(\mathtt{CP}, z, x, y) \wedge \textsc{some}(z.\text{cart.item.price}) > 100$" checks if (1) the

message instance of ID $v(z)$ from `Order` instance $v(x)$ to `Purchase` instance $v(y)$ is sent in the current c-state, and (2) $v(z)$ has an item with price greater than 100. ∎

Definition: Given $\Psi = \text{MSG}(\mathsf{M}, z, x, y) \wedge (\bigwedge_{i=1}^{k} \psi_i)$ as a message formula, a *responding message formula* (to Ψ) is of form "$\text{MSG}[z](\mathsf{M}', z', x', y') \wedge (\bigwedge_{i=1}^{n} \varphi_i)$", where"$\text{MSG}(\mathsf{M}', z', x', y') \wedge (\bigwedge_{i=1}^{k} \varphi_i)$" forms a message formula and $\text{MSG}[z](\mathsf{M}', z', x', y')$ is a responding message predicate to $\text{MSG}(\mathsf{M}, z, x, y)$.

Given a collaboration schema C, a c-state σ of C, a responding message formula $\Psi = \text{MSG}[z](\mathsf{M}, z', x, y) \wedge (\bigwedge_{i=1}^{k} \varphi_i)$, and an assignment v, Ψ is *valid* (with respect to σ and v) (denoted as $\sigma \models_v \Psi$), if $\sigma \models_v \text{MSG}(\mathsf{M}, z', x, y) \wedge (\bigwedge_{i=1}^{k} \varphi_i)$ and $(\text{ID}(z), \text{ID}(z'))$ is in the message-message dependency set of σ. Otherwise, Ψ is *invalid* with respect to σ and v (denoted as $\sigma \not\models_v \Psi$).

To specify the temporal constraints, some temporal operators are needed. The following lists all the temporal operators (adopted from DecSerFlow [54]) in the language and an intuitive explanation is also given.

Definition: Given a collaboration schema C, a *message constraint* (of C) is of form "$\Psi_1 \Theta \Psi_2$, where Ψ_1 is a message formula of C and Ψ_2 is a message formula of C or a responding message formula to Ψ_1 of C, and Θ ranges over

$$\{-(\textit{exist})-, -(\textit{co-exist})-, -(\textit{resp})\rightarrow, -(\textit{prec})\rightarrow, -(\textit{succ})\rightarrow,$$
$$-(\textit{al-resp})\rightarrow, -(\textit{al-prec})\rightarrow, -(\textit{al-succ})\rightarrow, -(\textit{im-resp})\rightarrow,$$
$$-(\textit{im-prec})\rightarrow, -(\textit{im-succ})\rightarrow\}.$$

Given a collaboration schema C, a c-behavior $\mathsf{B} = \sigma_1\sigma_2...\sigma_n$ of C, and an assignment v, a message constraint $\xi = \Psi_1 \Theta \Psi_2$ (where for each $i \in \{1, 2\}$, the (responding) message predicate in Ψ_i has message type M_i, message variable z_i, sender artifact variable x_i, and receiver artifact variable y_i) is *valid* (with respect to B and v) (denoted as $\mathsf{B} \models_v \xi$) if when ξ is of form

- $\Psi_1-(exist)-\Psi_2$: for each $i, j \in [1..n]$, if $\sigma_i \models_v \Psi_1$, then $\sigma_j \models_v \Psi_2$ (if Ψ_1 is valid sometime, then Ψ_2 is valid sometime),
- $\Psi_1-(co\text{-}exist)-\Psi_2$: Both $\mathsf{B} \models_v \Psi_1-(co\text{-}exist)-\Psi_2$ and $\mathsf{B} \models_v \Psi_2-(co\text{-}exist)-\Psi_1$,
- $\Psi_1-(resp)\rightarrow\Psi_2$: for each $i \in [1..n]$ and each $j \in [i..n]$, $\sigma_i \models_v \Psi_1$ and $\sigma_j \models_v \Psi_2$ (if Ψ_1 is valid sometime, then sometime in the future Ψ_2 is valid),
- $\Psi_1-(prec)\rightarrow\Psi_2$: for each $j \in [1..n]$ and each $i \in [1..j]$, $\sigma_i \models_v \Psi_1$ and $\sigma_j \models_v \Psi_2$ (if Ψ_2 is valid, then sometime in the past Ψ_1 is valid),
- $\Psi_1-(succ)\rightarrow\Psi_2$: Both $\mathsf{B} \models_v \Psi_1-(resp)\rightarrow\Psi_2$ and $\mathsf{B} \models_v \Psi_2-(prec)\rightarrow\Psi_1$,
- $\Psi_1-(al\text{-}resp)\rightarrow\Psi_2$: for each $i \in [1..n]$ and each $j \in [i..n]$, if $\sigma_i \models_v \Psi_1$ and $\sigma_j \models_v \Psi_2$, then for each $k \in [(i+1)..j]$, there does not exist a message formula Ψ whose message predicate is $\textsc{Msg}(\mathsf{M}_1, z, x_1, y_1)$, where $z \in \mathbf{V}_\mathrm{M}$, such that $\sigma_k \models_v \Psi$ (if Ψ_1 is valid, then in the future before Ψ_2 is valid, Ψ_1 is invalid),
- $\Psi_1-(al\text{-}prec)\rightarrow\Psi_2$: for each $j \in [1..n]$ and each $i \in [1..j]$, if $\sigma_i \models_v \Psi_1$ and $\sigma_j \models_v \Psi_2$, then for each $k \in [i..(j-1)]$, there does not exist a message formula Ψ whose message predicate is $\textsc{Msg}(\mathsf{M}_2, z, x_2, y_2)$, where $z \in \mathbf{V}_\mathrm{M}$, such that $\sigma_k \models_v \Psi$ (if Ψ_2 is valid, then in the past before Ψ_1 is valid, Ψ_2 is invalid),
- $\Psi_1-(al\text{-}succ)\rightarrow\Psi_2$: Both $\mathsf{B} \models_v \Psi_1-(al\text{-}resp)\rightarrow\Psi_2$ and $\mathsf{B} \models_v \Psi_2-(al\text{-}prec)\rightarrow\Psi_1$,
- $\Psi_1-(im\text{-}resp)\rightarrow\Psi_2$: for each $i \in [1..(n-1)]$, if $\sigma_i \models_v \Psi_1$, then $\sigma_{i+1} \models_v \Psi_2$ (if Ψ_1 is valid, then in the next step, Ψ_2 is valid),
- $\Psi_1-(im\text{-}prec)\rightarrow\Psi_2$: for each $i \in [2..n]$, if $\sigma_i \models_v \Psi_2$, then $\sigma_{i-1} \models_v \Psi_1$ (if Ψ_2 is valid, then in the previous step, Ψ_1 is valid),
- $\Psi_1-(im\text{-}succ)\rightarrow\Psi_2$: Both $\mathsf{B} \models_v \Psi_1-(im\text{-}prec)\rightarrow\Psi_2$ and $\mathsf{B} \models_v \Psi_2-(im\text{-}succ)\rightarrow\Psi_1$.

Example 6.2.19 Continue with Example 6.2.18; given v as an assignment , message constraint "$\textsc{Msg}(\texttt{CP}, z_{\texttt{CP}}, x, y) \wedge \textsc{some}(z_{\texttt{CP}}.\text{cart.item.price})>100 \ -(resp)\rightarrow \ \textsc{Msg}(\texttt{PC}, z_{\texttt{PC}}, y, w)$"

denotes that if the message instance $v(z_{\texttt{CP}})$ with an item having price greater than 100 from `Order` enactment $v(x)$ to `Purchase` enactment $v(y)$ is sent in the current c-state, then in the future, $v(y)$ needs to send a `PC` ("purchase complete") message instance to `Fulfillment` enactment $v(w)$.

Another restriction follows the Example 6.2.16: whenever an `OR` ("order request") message instance is sent (from "ext" to create a new `Order` enactment), in the future, there should be a `OC` ("order complete") message instance from the same `Order` enactment to "ext", and vice versa. This resection could be expressed with message constraint "$\textsc{Msg}(\texttt{OR}, z_{\texttt{OR}}, \text{ext}, x)\ -(succ)\rightarrow\ \textsc{Msg}[z_{\texttt{OR}}](\texttt{OC}, z_{\texttt{OC}}, x, \text{ext})$". ∎

Definition: Given a collaboration schema $C = (G, \Gamma, Msg)$, and an artifact in $\texttt{A}_0$ in G, a (*universally quantified*) *artifact sequence* (from $\texttt{A}_0$) is of form "$\forall x_0 \in \texttt{A}_0, x_1 \in x_0{\blacktriangleright}\texttt{A}_1, x_2 \in x_0{\blacktriangleright}\texttt{A}_2, ..., x_k \in x_0{\blacktriangleright}\texttt{A}_k$", where

- $k \in \mathbb{N}$,
- for each $i \in [0..k]$, $\texttt{A}_i$ is an artifact in G and $x_i \in \mathbf{V}_{\text{A}}$,
- for each distinct $i, j \in [0..k]$, $x_i \neq x_j$ and $\texttt{A}_i \neq \texttt{A}_j$, and
- for each $i \in [1..k]$, $\texttt{A}_0$ and $\texttt{A}_0$ are correlated (via an edge in G or by a correlation rule in Γ)

Given a collaboration schema $C = (G, \Gamma, Msg)$, an collaboration instance $\mathbb{I} = (\mathbb{T}, \mathbf{Corr}, \mathbf{DCorr})$, and an artifact sequence $s = \forall x_0 \in \texttt{A}_0, x_1 \in x_0{\blacktriangleright}\texttt{A}_1, x_2 \in x_0{\blacktriangleright}\texttt{A}_2, ..., x_k \in x_0{\blacktriangleright}\texttt{A}_k$ of C, the *identifier space* Σ of s with respect to $\mathbb{I}$ is a set of total mappings from $\{x_0, x_1, ..., x_k\}$ to $\mathbf{ID}_{\text{A}}$, such that a mapping v is in Σ if

- for each $i \in [0..k]$, there exists an artifact enactment I in $\mathbb{T}$ of $\texttt{A}_i$, such that $\textsc{id}(I) = v(x_i)$, and
- for each $i \in [1..k]$, $(v(x_0), v(x_i)) \in \mathbf{Corr} \cup \mathbf{DCorr}$.

Example 6.2.20 Considering the c-state described in Example 6.2.14, the artifact sequence "$\forall x_0 \in \texttt{Order}, x_1 \in x_0{}^{\blacktriangleright}\texttt{Payment}, x_2 \in x_0{}^{\blacktriangleright}\texttt{Purchase}$" has only one mapping v in its identifier space, i.e. $v(x_0) = \text{ID}(I^{\text{o}})$, $v(x_1) = \text{ID}(I^{\text{p}})$, and $v(x_2) = \text{ID}(I^{\text{c}})$, where I^{o}, I^{p}, and I^{c} are as stated in Example 6.2.14. ∎

Given an assignment v and a total mapping v' from a subset S of $\mathbf{V}_{\text{A}}$ to $\mathbf{ID}_{\text{A}}$, "$v \mid v'$" is an assignment such that for each $x \in S$, $v \mid v'(x) = v'(x)$ and for each $x \in \mathbf{ID}_{\text{A}} - S$, $v \mid v'(x) = v(x)$.

Definition: Given a collaboration schema C, a *choreography constraint* (of C) is of form "s, ξ", where

- $s = \forall x_0 \in \texttt{A}_0, x_1 \in x_0{}^{\blacktriangleright}\texttt{A}_1, ..., x_k \in x_0{}^{\blacktriangleright}\texttt{A}_k$ is an artifact sequence, where $k \in [0..2]$,
- ξ is a message constraint of C,
- for each artifact variable $x \in \mathbf{V}_{\text{A}} - \{\text{"ext"}\}$ occurring in ξ, $x \in \{x_0, ..., x_k\}$, and vice versa.

Given a collaboration schema C, a c-behavior $\mathsf{B} = \sigma_1\sigma_2...\sigma_n$ of C, where $\sigma_n = (\mathbb{I}, \mathbf{DCorr}), m, \overline{\text{M}}, \mathbf{MA}, \mathbf{MM})$ and a choreography constraint $\varphi = s, \xi$, φ is *valid* with respect to B (denoted as $\mathsf{B} \models \varphi$) if there exists an assignment v such that for each $v' \in \Sigma$, where Σ is the identifier space of s with respect to $\mathbb{I}$, $\mathsf{B} \models_{v \mid v'} \xi$.

Example 6.2.21 Continuing with Example 6.2.19, the two constraints can be expressed as

$$\begin{aligned}
&\forall y \in \texttt{Purchase}, w \in y{}^{\blacktriangleright}\texttt{Fulfillment}, x \in y{}^{\blacktriangleright}\texttt{Order},\\
&\qquad \text{MSG}(\texttt{CP}, z_{\texttt{CP}}, x, y) \wedge \text{SOME}(z_{\texttt{CP}}.\text{cart.item.price}){>}100\\
&\qquad -(\mathit{resp})\rightarrow \text{MSG}(\texttt{PC}, z_{\texttt{PC}}, y, w)\\
&\text{and}\\
&\forall x \in \texttt{Order},\\
&\qquad \text{MSG}(\texttt{OR}, z_{\texttt{OR}}, \text{ext}, x) \ -(\mathit{succ})\rightarrow \text{MSG}[z_{\texttt{OR}}](\texttt{OC}, z_{\texttt{OC}}, x, \text{ext})
\end{aligned}$$

The universal quantifier restricts that each constraint should be satisfied by all the combination of artifact enactments. ∎

Definition: A *choreography* (*specification*) is a tuple (G, Γ, Msg, κ) where (G, Γ, Msg) forms a collaboration schema C and κ a set of choreography constraints of C.

6.3 Realizability

In this section, we show that a subclass of choreographies defined in Section 6.2 can be realized. This is accomplished in two stages, we first translate a choreography into a "guarded conversation protocol" that is a conversation protocol of [58] extended with data contents and conditions. We then present a distributed algorithm that runs along with execution of each artifact, and show that an c-behavior is a possible execution with the algorithm iff it satisfies the choreography.

6.3.1 Guarded conversation protocols

A choreography $\mathcal{S} = (C, \kappa)$ is *one-to-one* (or *1-1*) if the correlation graph in C only has 1-1 correlations. The class of 1-1 choreographies contains choreographies allowed by existing languages, with a possible exception of BPEL4Chor [53, 59]. We focus on 1-1 choreographies in this section.

Definition: A *guarded (conversation) protocol* is a tuple (T, s, F, M, C, δ), where (i) T is a finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states; (ii) M is a finite set of messages type names, (iii) C is a set of data conditions, and (iv) $\delta \subseteq T \times M \times C \times T$ is a set of transitions.

A guarded protocol extends conversation protocol of [60] with data conditions on

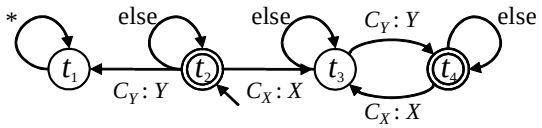

Figure 6.4: A guarded conversation protocol example

messages (and associated artifacts). The semantics of a guarded protocol is standard except that the data condition must be satisfied when making a transition.

Example 6.3.1 Fig. 6.4 shows an example guarded protocol with four states: t_2 is initial and t_2, t_4 are the final states. Two message types are involved, X and Y. C_X and C_Y are data conditions. The transition from t_2 to t_3 can be made if the condition C_X is true and message X is sent. An edge labeled with "else" stands for a collection of transitions other than the specified one(s) leaving the same state. An edge labeled with "*" represents all possible transitions leaving the state. ∎

Given an c-behavior B (of a correlation schema), and a guarded protocol τ, the notion of τ *accepting* B is defined in the standard way.

Theorem 6.3.2 Let $\mathcal{S}$ be a 1-1 choreography. One can effectively construct a guarded conversation protocol τ such that each c-behavior B satisfies $\mathcal{S}$ iff it is accepted by τ.

Since temporal operators in choreography constraints are operators in DecSerFlow that is contained in LTL [54], one can use a general technique to obtain Büchi automaton [61]. Guarded protocols can then be constructed. However, we use a simpler approach: translating each choreography constraint to a guarded protocol and then construct a product state machine for all constraints. Fig. 6.4 shows a guarded protocol for constraint "$X \wedge C_X - (succ) \rightarrow Y \wedge C_X$", where X, Y are message predicates and C_X, C_Y data conditions.

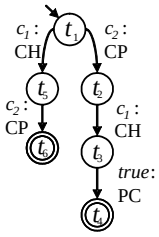

Figure 6.5: Model

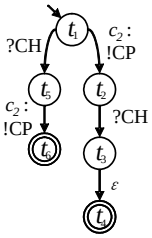

Figure 6.6: `Order`

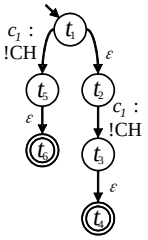

Figure 6.7: `Payment`

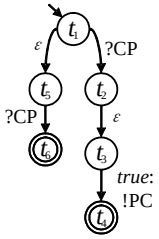

Figure 6.8: `Purchase`

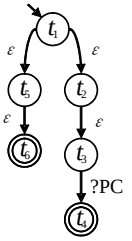

Figure 6.9: `Fulfillment`

Example 6.3.3 Fig. 6.5 shows a guarded protocol based on the example discussed in Section 6.1, where c_1 is "Payment.balance > CH.amount" and c_2 is "CP.items $\neq$ null". Since each participant can have at most one instance (1-1 choreography) type level notation is used here. The initial state is t_1, the final states are t_4 and t_6 (in the original guarded protocol, they are not final states but we make them final to show a complete example). Only two sequences of messages can be accepted in this example: either (1) CP CH PC, or (2) CH CP. Note that this is not realizable in [60]. ∎

The only-if direction of Theorem 6.3.2 fails if 1-1 condition on choreography is removed. This is because different instances of the same interface may progress in different paces and a guarded protocol cannot capture such situations.

6.3.2 Guarded peers

Guarded automaton was introduced in [60] to represent a state machine for a participant. We modify the notion to allow message predicates and data conditions. The following defines a projection of a guarded protocol to a participant.

Definition: Given a guarded conversation protocol $\tau = (T, s, F, M, C, \delta)$ and an artifact type α, a *guarded peer for* α wrt τ is a tuple $(T, s, F, M', C', \delta_s, \delta_r, \delta_\varepsilon)$, where (1) $M' \subseteq M$

such that each message in M' has α as a sender or receiver, (2) $C' \in C$ contains a condition c if there exists $t, t' \in T$ and a message m in M', where m is sent by α and $(t, c, m, t') \in \delta$, (3) $\delta_s \subseteq T \times C' \times M' \times T$ (sending transitions) contains elements (t, c, m, t') if $(t, c, m, t') \in \delta$ and m is sent by α, (4) $\delta_r \subseteq T \times M' \times T$ (receiving transitions) contains elements (t, m, t') if there exists $c \in C$, $(t, c, m, t') \in \delta$ and the receiver of m is α, and (5) $\delta_\varepsilon \subseteq T \times \{\varepsilon\} \times T$ (empty transitions) contains elements (t, ε, t') if there exists $c \in C$, $m \in M$, $(t, c, m, t') \in \delta$ and α is neither the receiver or sender of m.

Example 6.3.4 Fig. 6.6 – 6.9 show four guarded peers (`Order`, `Payment`, `Purchase`, and `Fulfillment`) projected from Fig. 6.5. The "?" mark denotes receiving a message, the "!" mark denotes sending a message. ∎

An artifact (enactment) sends or receives messages according to its guarded peer, i.e., each guarded peer is autonomous. If all guarded peers start from their initial states, make their transitions autonomously, the composition terminates when every guarded peer reaches a final state. Our composition model is basically the same as [60], except that FIFO queues are not used.

Example 6.3.5 Consider the sequence of messages "CH CP" that is accepted by the guarded protocol in Fig. 6.5 (Example 6.3.3). The projected peers are shown in Fig. 6.6 – 6.9. `Payment` can send a CH to `Order`. Then `Payment` follows an empty transition into its final state t_6 and `Payment` is now in t_5. Later on, `Order` sends a CP to `Purchase` and both of them can reach their final states (t_6). While for `Fulfillment`, it sends or receives nothing and follows two empty transitions to t_6. ∎

Naturally, given a guarded protocol τ, if a sequence of transitions is accepted by τ, the sequence is also accepted by all guarded peers of τ. In general, the other direction

may not necessarily hold [52].

Example 6.3.6 Continue with Examples 6.3.3 and 6.3.4. Suppose `Payment` sends a CH through edge (t_1, t_5) and ends at final state t_6. Then `Order` sends a CP, receives the CH sent by `Payment`, and reaches final state t_4. Correspondingly, `Purchase` receives the CP from `Order` and reaches final state t_4 by sending `Fulfillment` a PC. Finally, `Fulfillment` receives the PC and ends at t_4 as well.

Clearly, the above sequence of messages "CH CP PC" allows all guarded peers to reach their final states but cannot be accepted by the original guarded protocol. ∎

The *realizability problem* is to ensure that the collective transitions for all guarded peers are equivalent to transitions for the original guarded protocol. While this problem has not been investigated, a closely related problem of "realizability checking problem" [52] which tests if a conversation protocol can be restored from the product of its projected peers has been studied extensively (see [62]).

6.3.3 A realization mechanism

Instead of checking if a guarded protocol is the product of its guarded peers, we take a different approach. We develop a protocol (algorithm) that in addition to the original messages, it also adds a small number of "synchronization" messages to aid participants (peers) in their autonomous execution. We show that the synchronized execution generates equivalent behaviors as the original guarded protocol and that in every successful execution, the total number of synchronization messages is bounded by the sum of the number of messages in the guarded protocol and the number of guarded peers.

A naïve protocol simply broadcasts every message to all. However, this approach requires as many as $N^* \times (k-1)$ messages during the process, where N^* is the number of message instances (needed for the collaboration), and k is the number of peers.

To reduce synchronization messages, an improvement is developed that employs a "token passing" method: only the participant who owns a "token" can make a transition. Once a transition is conducted (or equivalently, a message is sent), the "token" will be passed to the next sender and this process repeats.

Given a guarded protocol $\tau = (T, s, F, M, C, \delta)$, we augment τ with a new message type named *sync* without any data attributes. We also introduce two functions, *Flag* and *State*. The function *Flag* maps message (including sync) instances to the set $\{\text{SND}, \text{RCV}, \text{FIN}\}$ such that if μ is an instance of sync, $\mathit{Flag}(\mu) \in \{\text{SND}, \text{FIN}\}$. Intuitively, SND is the token, RCV means the message that is regular, FIN instructs the receiver to terminate. The function *State* maps each message instance to T to indicate the current (global) state. Each message is sent along with its *Flag* and *State* values.

To implement the framework, a coordinator is used for each peer (instance) to help on transition decisions. Once a coordinator receives the token carried by a message, it makes a transition for its peer by sending a message with an appropriate *Flag*, and possibly passes the token to the next sender if different (via a flagged sync message).

As mentioned at the beginning of this section, once (the coordinator of) a sender sends a message with the flag RCV, it passes the token to (the coordinator of) the next sender through a message with the flag SND. In order to know who will be the possible sender, a concept "*sender set*" is defined first.

Given a guarded protocol $\tau = (T, s, F, M, C, \delta)$, the *sender set* of a state $t \in T$, denoted as $\mathit{sdr}(t)$, is a set containing all artifact interfaces α that is the sender of m where $(t, c, m, t') \in \delta$ for some $t' \in T$, $c \in C$, and $m \in M$. In Fig. 6.5, the sender sets for t_1 to t_6 are {`Order`, `Payment`}, {`Payment`}, {`Purchase`}, $\varnothing$, {`Order`}, and $\varnothing$, resp.

Algorithm 4 Coordinator for Peer p

Input: sdr, $p = (T, s, F, M, C, \delta_s, \delta_r, \delta_\varepsilon)$

1: **loop**
2: Wait for the next message m
3: **if** $\mathit{Flag}(m) = \text{SND}$ **then**
4: **if** $\exists c \in C, \exists m' \in M, \exists t \in T$, $(\mathit{State}(m), c, m', t) \in \delta_s$ **then**
5: Send m' (flag: RCV, state: t);
6: randomly select s from $sdr(t)$;
7: Send to s a sync message (flag: SND, state: t);
8: **end if**
9: **else if** $\mathit{Flag} = \text{RCV}$ **then**
10: **if** $\mathit{State}(m) \in F$ **then**
11: Boardcast sync message (flag: FIN, state: $\mathit{State}(m)$)
12: Terminate
13: **end if**
14: **else**
15: Terminate {"FIN" case}
16: **end if**
17: **end loop**

Sender sets are known at design time, the current sender can choose the next sender from the corresponding sender set of the current state at runtime. The initial sender should be delegated externally by, e.g., the environment. These steps are repeated until a peer (with the token) reaches a final state. This peer then informs all other peers to end the execution by sending messages of flag FIN. Alg. 4 accomplishes the coordinator that runs on individual peers.

Example 6.3.7 The following describes a possible execution. The user chooses and sends to `Order` (in the sender set for t_1). `Order` sends a CP (flag: RCV) to `Purchase` and inform `Payment` to be the next sender (through a sync message with flag SND) since $sdr(t_2) = \{\texttt{Payment}\}$. After `Payment` sends a CH to `Order`, it will pick `Purchase` from $sdr(t_3)$. Finally, `Purchase` sends a PC to `Fulfillment`. Once the `Fulfillment` reaches its

final state t_4, FIN messages will be broadcast. ∎

Theorem 6.3.8 Given a guarded conversation protocol τ, each sequence of ground messages, is accepted by τ iff it is generated by Alg. 4 running for guarded peers of τ.

The correctness and completeness of Alg. 4 can be guaranteed as the protocol is loyally simulating the transitions upon the global guarded conversation protocol.

Remark 6.3.9 Denote by N^* the number of regular messages that should be sent, and $\hat{N}$ as the number of regular and synchronization messages sent according to Alg. 4. It is easy to see that $\hat{N} < 2N^* + (k-1)$, where k is the number of peers. Furthermore, if FIN messages are not needed, the bound is reduced to $\hat{N}/N^* < 2$.

The result claimed in Remark 6.3.9 can be trivially proved, as the number of "SND" messages sent in Alg. 4 is no more than the one of "RCV" messages. Thus, Alg. 4 provides a bounded approximation against the minimum number of control messages that should be sent.

6.4 Summary

This chapter proposes a declarative choreography language that can express correlations and choreographies for artifact-centric BPs in both type and instance levels. It also incorporate data contents and cardinality on participant instances into choreography constraints. Furthermore, a subclass of the rule-based choreography is shown to be equivalent to a state-machine-based choreography.

Chapter 7

Satisfiability of Collaboration

The specification of business processes can be imperative or declarative. Comparing with imperative specification, declarative specification allows more execution and is flexible for runtime changes. Chapter 6 takes advantage of DecSerFlow, a declarative business process specification, to model choreography. DecSerFlow consists of a set of temporal predicates that can be translated into LTL but limited to finite sequences. An execution sequence is valid as long as it satisfies all given constraints. This chapter continues with the topic on DecSerFlow and focuses on the "non-trivial finite satisfiability problem": Given a set of DecSerFlow constraints, is there a finite execution sequence that satisfies all constraints in the set and meets the minimum requirement upon number of occurrences (in a process)?

Specifically, this chapter provides syntactical characterizations for non-trivial finite satisfiability of several classes of DecSerFlow constraints. These characterizations directly lead to polynomial time satisfiability testing. To achieve this, we first determine the "core" constraints of DecSerFlow for the conformance problem, i.e., a reduction from general DecSerFlow to DecSerFlow "Core" that does not contain existence constraints nor cardinality requirements, and we show that the conformance checking on these two

DecSerFlow specifications are equivalent (Theorem 7.2.3). For DecSerFlow Core, we formulate syntactic characterizations (sufficient and necessary for conformance) for constraints involving (1) ordering and immediate constraints (Theorem 7.3.9), (2) ordering and alternating constraints (Theorem 7.4.4), (3) alternating and immediate constraints (Theorem 7.4.23), or (4) ordering, alternating, and immediate constraints with only precedence (or only response) direction (Theorem 7.5.1). The general case (i.e., for all three types of constraints in both directions) remains an open problem.

The remainder of the chapter is organized as follows. Section 7.1 defines DecSerFlow constraints and the conformance problem. (Note that DecSerFlow studied in this chapter is the same as the one defined in Section sec:lang, only with notation altered in order to have a cleaner presentation). Section 7.2 shows that only "core" constraints are needed for determining conformance. Section 7.3 focuses on ordering and immediate constraints. The combinations of alternating constraints with other constraints are discussed in Section 7.4. Section 7.5 discusses all constraints with only one direction. Evaluation is given in Section 7.6 Conclusions are provided in Section 7.7.

7.1 DecSerFlow Constraints and Problem Definition

Business process models are either imperative or declarative [63]. Imperative processes typically employ graphs (e.g., automata, Petri Nets) to depict how a process should progress. Declarative processes usually use constraints [54]. Declarative models are more flexible and are easy to change during design time or runtime [16]. One practical question is whether a given specification as a set of constraints allows at least one execution. It is fundamental in business process modeling to test satisfiability of a given set of constraints.

A process execution is a sequence of activities through time, constraints are in general temporal. In this chapter, we focus on DecSerFlow [54], a language that uses a set of

temporal predicates as a process specification, and study the satisfiability problem. The temporal predicates in DecSerFlow can be translated into linear temporal logic (LTL) [9] *but limited to finite sequences*. A naive approach to DecSerFlow satisfiability checking is to construct automata representing individual constraints and determine if their cross product accepts a string. However the complexity of this approach is exponential with respect to the number of given constraints. In this chapter, we develop syntactical characterizations for DecSerFlow satisfiability that automatically lead to polynomial time complexity. Recently, the DECLARE system was developed [64] to support design and execution of DecSerFlow processes. Clearly efficient satisfiability testing provides an effective and efficient help to the user of DECLARE.

In the following, we briefly introduce how DecSerFlow works. The underlying semantics is the same as the one introduced Section 6.2; but working on different scenarios: the operands in this chapter are activities only, comparing with the message instances with data in Chapter 6.

Most DecSerFlow constraints can be categorized into two directions: "response" (*Res*), which specifies that an activity should happen in the future, and "precedence" (*Pre*), which specifies that an activity should happen in the past. For each direction, there are three types of constraints: (1) Ordering constraints, denoted as $Res(a, b)$ (or $Pre(a, b)$), where a and b are activity names, specify that if a occurs, then b should occur sometime in the future (resp. past). For example, sequence *cbacaa* satisfies $Pre(a, b)$, but *cabbaca* does not. A real life example might be that in a loan application for house purchase process, if activity "loan approval" happens, then in the past a "credit check" activity should have happened. (2) Alternating constraints, denoted as $aRes(a, b)$ (or $aPre(a, b)$), specify that the occurrence of a implies a future (resp. past) occurrence of b but before the occurrence of b, a cannot occur again (i.e., between two occurrences of a, there should exist an occurrence of b). For example, *babcbacb* satisfies $aRes(a, b)$, but

abbcaab does not. As an example, if a "house evaluation request" activity happens, a "house evaluation feedback" activity should happen in the future and before receiving the feedback, the applicant cannot submit another evaluation request, i.e., "request" and "feedback" should be alternating. (3) Immediate constraints, denoted as $iRes(a, b)$ (or $iPre(a, b)$), restrict that if a occurs, then b should immediately follow (resp. occur before). For example, *abcab* satisfies $iRes(a, b)$, but *abcacb* does not. In addition to "response" and "precedence" constraints, there is a special type of "existence" constraints that only require occurrences in any order. An existence constraint, $Exi(a, b)$ restricts that if a occurs, then b should occur either earlier or later. For example, *abcbc* satisfies $Exi(a, c)$, but *abbab* does not. In practice, a common existence constraint can be that a guest can either choose to pay the hotel expense online then check in, or check in first and pay the expense later, i.e., $Exi($"check in", "payment"$)$.

In addition to temporal constraints, there are cardinality requirements for each activity, i.e., an activity should occur at least once or does not have to occur. Such requirements are common in business processes. For example, in an online order process, "payment" activity is always required to occur; while "shipping" is not (a customers may pick up the ordered items in the nearest store). In order to reflect design requirements, we study the problem if a given set of constraints can be "non-trivially finitely" satisfied: is there a *finite* execution sequence that satisfies all given constraints as well as the cardinality requirements (in a process). We use the term "conformance" to denote this concept of non-trivial finite satisfiability.

In the remainder of this section, we define DecSerFlow formally together with the conformance probelm.

Let $\mathcal{A}$ be an infinite set of *activities*, $\mathbb{N}$ the set of natural numbers, $[i..j]$ the set $\{k \in \mathbb{N} \mid i, j \in \mathbb{N}, i \leqslant k \leqslant j\}$, and $A \subseteq \mathcal{A}$ a (finite) subset of $\mathcal{A}$. A *string over* A (or $\mathcal{A}$) is a finite sequence of 0 or more activities in A (resp. $\mathcal{A}$). Let ε denote the *empty* string,

A^* ($\mathcal{A}^*$) the set of all finite strings over A (resp. $\mathcal{A}$). Given a string s, the *length* of s is the number of activity occurrences in s and denoted as $len(s)$. Obviously, $len(\varepsilon) = 0$. If $s \neq \varepsilon$, for each $1 \leqslant i \leqslant len(s)$, let $s^{[i]}$ be the i-th activity occurrence in s.

A *subsequence* of $a_1a_2...a_n$ is a string $a_{k_1}a_{k_2}...a_{k_m}$, where (1) $m \in \mathbb{N}$ and $m \geqslant 1$, (2) $k_i \in [1..n]$ for each $i \in [1..m]$, and (3) $k_i < k_{i+1}$ for each $i \in [1..(m-1)]$. A *substring* of a string s is a subsequence $a_{k_1}a_{k_2}...a_{k_m}$ of s where for each $i \in [1..(m-1)]$, $k_i = k_{i+1} - 1$.

Let $A \subseteq \mathcal{A}$, $a, b \in A$. A *constraint on* a, b over sequences in A^* is one of the following, letting s be a string over A.

- *Existence* constraint $Exi(a, b)$: *s satisfies $Exi(a, b)$*, $s \models Exi(a, b)$, if either (1) $s = \varepsilon$, or (2) for each $i \in [1, len(s)]$, $s^{[i]} = a$ implies $s^{[j]} = b$ for some $j \in [i, len(s)]$.

- *Ordering response* constraint $Res(a, b)$: $s \models Res(a, b)$, if either (1) $s = \varepsilon$, or (2) $s^{[len(s)]} \neq a$ and for each $i \in [1, len(s)-1]$, $s^{[i]} = a$ implies $s^{[j]} = b$ for some $j \in [i+1, len(s)]$.

- *Ordering precedence* constraint $Pre(a, b)$: $s \models Pre(a, b)$ if either (1) $s = \varepsilon$, or (2) $s^{[1]} \neq a$ and for each $i \in [2, len(s)]$, $s^{[i]} = a$ implies $s^{[j]} = b$ for some $j \in [1, i-1]$.

- *Alternating response* constraint $aRes(a, b)$: $s \models aRes(a, b)$, if either (1) $s = \varepsilon$, or (2) $s^{[len(s)]} \neq a$ and for each $i \in [1, len(s)-1]$, $s^{[i]} = a$ implies that for some $j \in [i+1, len(s)]$, (i) $s^{[j]} = b$, and (ii) for each $k \in [i+1, j-1]$, $s^{[k]} \neq a$.

- *Alternating precedence* constraint $aPre(a, b)$: $s \models aPre(a, b)$ if either (1) $s = \varepsilon$, or (2) $s^{[1]} \neq a$ and for each $i \in [2, len(s)]$, $s^{[i]} = a$ implies for some $j \in [1, i-1]$, (i) $s^{[j]} = b$, and (ii) for each $k \in [j+1, i-1]$, $s^{[k]} \neq a$.

- *Immediate response* constraint $iRes(a, b)$: $s \models iRes(a, b)$, if either (1) $s = \varepsilon$, or (2) $s^{[len(s)]} \neq a$ and for each $i \in [1, len(s)-1]$, $s^{[i]} = a$ implies $s^{[i+1]} = b$.

- *Immediate precedence* constraint $iPre(a, b)$: $s \models iPre(a, b)$ if either (1) $s = \varepsilon$, or (2) $s^{[1]} \neq a$ and for each $i \in [2, len(s)]$, $s^{[i]} = a$ implies $s^{[i-1]} = b$.

	Response	Precedence
Ordering	$Res(a,b)$: each occurrence of a is followed by an occurrence of b	$Pre(a,b)$: each occurrence of a is preceded by an occurrence of b
Alternating	$aRes(a,b)$: in addition to $Res(a,b)$, a and b alternate	$aPre(a,b)$: in addition to $Pre(a,b)$, a and b alternate
Immediate	$iRes(a,b)$: each occurrence of a is immediately followed by an occurrence of b	$iPre(a,b)$: each occurrence of a is immediately preceded by an occurrence of b
Existence	$Exi(a,b)$: each occurrence of a implies an occurrence of b	

Figure 7.1: Summary of Constraints

Fig. 7.1 shows a summary of the constraints. For ordering precedence constraint $Pre(a,b)$, if "a" occurs in a string, then before "a", there must exist a "b", and between this "b" and "a", all activities are allowed to occur. Similarly, for alternating response constraint $aRes(a,b)$, after an occurrence of "a", no other a's can occur until a "b" occurs. For immediate precedence constraint $iPre(a,b)$, a "b" should occur immediately before "a". The existence constraints have no restrictions on temporal orders.

Definition: A (*DecSerFlow*) *schema* is a pair $S=(A,C,\kappa)$ where $A \subseteq \mathcal{A}$ is a finite set of activities, C a finite set of constraints on activities in A, and κ is a total mapping from A to $\{0,1\}$, called *cardinality*, to denote that an activity $a \in A$ should occur at least once (if $\kappa(a)=1$) or no occurrence requirement (if $\kappa(a)=0$).

For notation convenience, given a DecSerFlow schema $S=(A,C,\kappa)$, if for each $a \in A$, $\kappa(a)=1$, we simply use (A,C) to denote S.

Definition: A finite string s over A *conforms to* schema $S=(A,C,\kappa)$ if s satisfies every constraint in C and for each activity $a \in A$, s should contain a for at least $\kappa(a)$ times. If a string s conforms to S, s is a *conforming string* of S and S is *conformable*.

Conformance Problem: Given a schema S, is S conformable?

Proposition 7.1.1 The conformance problem is in PSPACE (in the size of the input schema). ∎

Proposition 7.1.1 is straightforward. The idea is to construct a finite state automaton A for each given constraint c (and each cardinality requirement r, i.e., an actvitiy occurring at least 0 or 1 times), such that A can accept all strings that satisfy c (resp. accept all strings that satisfy r) and reject all other strings. Then the conformance problem is translated to checking if the cross product of all constructed automata (that corresponding to the given constraints) can accept a non-empty string.

7.2 Core Constraints

In this section, we show that the conformance problem for DecSerFlow schemas is reducible to that for schemas with only "core" constraints, i.e., without existence and cardinality constraints. We thus need to focus only on the core constraints.

Let $S = (A, C, \kappa)$ be a DecSerFlow schema. We can effectively construct another schema $S' = (A', C')$ where C' contains no existence constraints and each activity in A' must occur at least once such that S is conformable iff S' is conformable. Specifically, $S' = (A', C')$ is constructed as follows:

- $A' = A_1 \cup A_2$ where $A_1 = \{\, a \in A \mid \kappa(a)=1 \}$ and $A_2 = \{\, a \in A \mid$ there exist a positive $n \in \mathbb{N}$, $b_0 \in A_1$, and $b_i \in A - A_1$ for each $i \in [1..n]$ such that $a = b_n$ and there is a constraint $\xi_i(b_{i-1}, b_i) \in C$ for each $i \in [1..n] \,\}$.
- $C' = \{\xi(a, b) \in C \mid a, b \in A'$ and ξ is not an existence constraint $\}$.

Note that $A' \subseteq A$ and $C' \subseteq C$.

Example 7.2.1 Consider a schema S with six activities, a, b, c, d, e, f, and six constraints, $Res(c, b)$, $iRes(f, a)$, $aPre(f, c)$, $aRes(b, e)$, $Pre(b, a)$, $Exi(a, c)$. S also specifies that a, b, f are optional but c, d, e must occur at least once. Using the above construction, we have $A_1 = \{c, d, e\}$ and A_2 contains b due to $Res(c, b)$ and also a due to $Res(c, b)$, $Pre(b, a)$. Note that f is not in A_2 (nor in A_1). Therefore, the reduced schema $S' = (A', C')$ where $A' = \{a, b, c, d, e\}$ and $C' = \{Res(c, b), aRes(b, e), Pre(b, a)\}$. In particular, C' does not contain the existence constraint $Exi(a, c)$. ∎

Lemma 7.2.2 *Let $S = (A, C, \kappa)$ be a schema and A_1, A_2 as constructed in the above from S. For each conforming string s of S, every activity in $A_1 \cup A_2$ occurs in s.*

Proof: Let $a \in A$ be an activity. If $a \in A_1$, then $\kappa(a) = 1$ and clearly a must occur in s. Consider now $a \in A_2$. According to the construction, there exist an $n > 0$, $b_0 \in A_1$, and $b_i \in A - A_1$ for each $i \in [1..n]$ where $a = b_n$ and there is a constraint $\xi_i(b_{i-1}, b_i) \in C$ for each $i \in [1..n]$. Using an inductive argument, it is easy to see that each b_i ($i \in [0..n]$) occurs in s. Since $a = b_n$, the lemma follows. ∎

Theorem 7.2.3 Let $S = (A, C, \kappa)$ be a schema and $S' = (A', C')$ as constructed in the above from S. Then, S is conformable iff S' is conformable.

Proof: Let A_1, A_2 be constructed from S as in the above.

("Only-if") Since $C' \subseteq C$, every string s conforming to S also satisfies every constraint in C'. By the construction, $A' = A_1 \cup A_2$ where By Lemma 7.2.2, every activity in $A_1 \cup A_2 = A'$ occurs in s. Thus, s conforms to S'.

("If") Let s be a conforming string of S'. Since $A' \supseteq \{\, a \mid \kappa(a){=}1 \}$, s satisfies the cardinality constraints κ. Now let $\xi(a, b)$ be a constraint in C but not in C'. Consider the following two cases.

(i) $a \notin A'$. In this case a does not occur in s, therefore s satisfies $\xi(a, b)$.

(ii) $a \in A'$. By an inductive argument, it is each to verify that $b \in A'$. It follows that $\xi(a, b)$ is an existential constraint $Exi(a, b)$. s conforming to S' implies that s satisfies $Exi(a, b)$, i.e., $\xi(a, b)$. ∎

Theorem 7.2.3 shows that conformance of arbitrary schemas can be reduced to conformance of schemas where each activity occurs at least once. If every activity in a given schema occurs at least once, the existence constraints are redundant. In the remainder of this chapter, we only focus on schemas with *core* constraints, i.e., from the set $\{Res, Pre, aRes, aPre, iRes, iPre\}$ and that each activity occurs at least once.

For the ease of presentation, we classify core constraints into three classes: *ordering* (ordering response and ordering precedence) *alternating* (alternating response and alternating precedence), and *immediate* (immediate response and immediate precedence) constraints. For each category, there are two directions: response (forward) and precedence (backward). We define response (or precedence) constraints to be all ordering, alternating, and immediate response (resp. precedence) constraints.

7.3 Characterizations for Ordering & Immediate Constraints

This section focuses on syntactical characterizations of conformable schemas that only contain ordering and/or immediate constraints.

For each schema $S = (A, C)$, we construct the *causality graph* $\mathcal{G}_S$ of S as a labeled graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ with the vertex set A and six edge sets where $E^{\text{x}}_{\blacktriangleright}$ ($E^{\text{x}}_{\blacktriangleleft}$) corresponds to response (resp. precedence) constraints of ordering (x = 'or'), alternating (x = 'al'), or immediate (x = 'im') flavor. Specifically, for all $a, b \in A$,

- $(a,b) \in E^{\text{or}}_{\blacktriangleright}$ iff $Res(a,b)$ is in C,
- $(a,b) \in E^{\text{al}}_{\blacktriangleleft}$ iff $aPre(a,b) \in C$,
- $(a,b) \in E^{\text{im}}_{\blacktriangleright}$ iff $iRes(a,b) \in C$, and the other three cases are similar.

Example 7.3.1 Consider a schema with activities: a, b, c, d, e, and the constraints:

	response	precedence
ordering	$Res(a,b),\ Res(b,d)$	$Pre(c,d)$
alternating	$aRes(c,d),\ aRes(c,e)$	$aPre(a,b),\ aPre(b,c)$
immediate	$iRes(c,a),\ iRes(d,e)$	$iPre(e,a)$

Its causality graph has vertices $\{a,b,c,d,e\}$ and edge sets $E^{\text{or}}_{\blacktriangleright} = \{(a,b),(b,d)\}, E^{\text{or}}_{\blacktriangleleft} = \{(c,d)\}, E^{\text{al}}_{\blacktriangleright} = \{(c,d),(c,e)\}, E^{\text{al}}_{\blacktriangleleft} = \{(a,b),(b,c)\}, E^{\text{im}}_{\blacktriangleright} = \{(c,a),(d,e)\}, E^{\text{im}}_{\blacktriangleleft} = \{(e,a)\}$. ∎

Given a causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, if an edge set is empty, we will conveniently omit it; for example, if $E^{\text{im}}_{\blacktriangleright} = E^{\text{im}}_{\blacktriangleleft} = \varnothing$, we write the causality graph simply as $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$.

For the technical development, we first review some well-known graph notions. Given a (directed) graph (V,E) with the vertex set V and the edge set $E \subseteq V \times V$, a *path* is a sequence $v_1 v_2 ... v_n$ where $n > 1$, for each $i \in [1..n]$, $v_i \in V$, and for each $i \in [1..(n-1)]$, $(v_i, v_{i+1}) \in E$; n is the *length* of the path $v_1 v_2 ... v_n$. A path $v_1 v_2 ... v_n$ is *simple* if v_i's are pairwise distinct except that v_1, v_n may be the same node. A (*simple*) *cycle* is a (resp. simple) path $v_1 v_2 ... v_n$ where $v_1 = v_n$. A graph is *cyclic* if it contains a cycle, *acyclic* otherwise. Given an acyclic graph (V,E), a *topological order* of (V,E) is an enumeration of V such that for each $(u,v) \in E$, u precedes v in the enumeration. A *subgraph* (V',E') of (V,E) is a graph, such that $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A graph is *strongly connected* if there is a path from each node in the graph to every other node. Given a graph $G = (V,E)$ and a set $V' \subseteq V$, the *projection* of G on V', $\pi_{V'}G$, is a subgraph (V',E') of G where $E' = E \cap (V' \times V')$. A *strongly connected component* (V',E') of a graph $G = (V,E)$

is a strongly connected subgraph $G' = (V', E')$ of G, such that (1) $G' = \pi_{V'}G$, and (2) for each $v \in V - V'$, $\pi_{V' \cup \{v\}}G$ is not strongly connected.

7.3.1 Ordering constraints alone

The following states a syntactical characterization for conformance of schemas containing only ordering constraints.

Theorem 7.3.2 For a schema $S = (A, C)$ with only ordering constraints and its causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft})$, S is conformable iff both graphs $(A, E^{\text{or}}_{\blacktriangleright})$ and $(A, E^{\text{or}}_{\blacktriangleleft})$ are acyclic.

Proof: Let $S, A, C, E^{\text{or}}_{\blacktriangleleft}$, and $E^{\text{or}}_{\blacktriangleright}$ be as stated in the theorem.

($\Rightarrow$) We prove by contradiction. Without loss of generality (w.l.o.g.), suppose that $(A, E^{\text{or}}_{\blacktriangleright})$ has a cycle and there is a string s that satisfies all the constraints in C. Then there exist an integer $n \geqslant 1$ and $a_i \in A$ for each $i \in [1..n]$ such that $Res(a_i, a_{i \bmod n+1}) \in C$ for each $i \in [1..n]$. Thus, for all $i \in [1..n]$, each occurrence of a_i in s must have a following occurrence of $a_{i \bmod n+1}$ in s, etc. It follows that s cannot be finite and therefore is not a string. The case when $(A, E^{\text{or}}_{\blacktriangleleft})$ is cyclic is similar.

($\Leftarrow$) Let $a_1 a_2 ... a_n$ be a topological sort of A wrt $(A, E^{\text{or}}_{\blacktriangleright})$, and $b_1 b_2 ... b_n$ be a topological sort of A wrt $(A, E^{\text{or}}_{\blacktriangleleft})$. It is easy to see that the string $s = b_n ... b_2 b_1 a_1 a_2 ... a_n$ satisfies every constraint in C. Since $a_1 ... a_n$ is a topological sort of $(A, E^{\text{or}}_{\blacktriangleright})$, for each $i, j \in [1..n]$, if $Res(a_i, a_j) \in C$, then $i < j$ and $a_1 ... a_n$ satisfies $Res(a_i, a_j)$. If for some $k \in [1..n]$, $b_k = a_i$, since b_k occurs before a_i and a_i occurs before a_j in $s = b_n ... b_1 a_1 ... a_n$, s also satisfies $Res(a_i, a_j)$. A similar argument holds for each precedence constraint. Moreover, as each activity occurs in s, this direction holds. ∎

Example 7.3.3 Consider a schema S with activities $A = \{a, b, c\}$ and constraints $Res(a, b)$, $Res(b, c)$, $Pre(b, c)$, and $Pre(b, a)$. Both graphs $(A, \{(a, b), (b, c)\})$ and $(A, \{(b, c), (b, a)\})$ are acyclic. Theorem 7.3.2 predicts conformance; conforming strings include *bcaabc* and *cabc*. However if we add constraint $Res(c, a)$ into S, it is no longer conformable since graph $(A, \{(a, b), (b, c), (c, a)\})$ now has a cycle *abca*; it is clear that no finite strings can satisfy $Res(a, b) \wedge Res(b, c) \wedge Res(c, a)$. ∎

7.3.2 Immediate constraints alone

Before we discuss characterizations for immediate constraints, we state the following property observed in [54].

Lemma 7.3.4 [54] Given two activities u and v, the following logical implications (denoted by "$\rightarrow$") always hold:

$$iRes(u, v) \rightarrow aRes(u, v) \qquad aRes(u, v) \rightarrow Res(u, v)$$
$$iPre(u, v) \rightarrow aPre(u, v) \qquad aPre(u, v) \rightarrow Pre(u, v)$$

Theorem 7.3.5 below provides a necessary and sufficient condition for schemas conformance with only immediate constraints.

Theorem 7.3.5 Given a schema $S = (A, C)$ containing only immediate constraints and its causality graph $(A, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, S is conformable iff the following all hold:

(1). $(A, E^{\text{im}}_{\blacktriangleright})$ and $(A, E^{\text{im}}_{\blacktriangleleft})$ are both acyclic,

(2). for each $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (or $E^{\text{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq u$ and $(v, w) \in E^{\text{im}}_{\blacktriangleleft}$ (resp. $E^{\text{im}}_{\blacktriangleright}$), and

(3). for each $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (or $E^{\text{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq v$ and $(u, w) \in E^{\text{im}}_{\blacktriangleright}$ (resp. $E^{\text{im}}_{\blacktriangleleft}$).

Example 7.3.6 Consider a schema S with 3 activities, a, b, c, and 2 constraints $\{iRes(a, b), iPre(c, b)\}$. S is conformable according to Theorem 7.3.5; a conforming string is "abc". However, adding $iPre(b, c)$ to S would cause S to be non-conformable due to the requirement that each occurrence of a should be immediately followed by an occurrence of b (constraint $iRes(a, b)$), and each occurrence of b should be immediately preceded by an occurrence of c (constraint $iPre(b, c)$). This is impossible since $a \neq c$. Similarly, addition of $iRes(a, c)$ into S also causes nonconformity. ∎

To prove Theorem 7.3.5, we need the following (data) structure. This structure is also used later in the proof of Theorem 7.3.9 and the next section.

Given a schema $S = (A, C)$, let $\pi_{\text{im}}(S) = (A, C')$ be a schema obtained from S, where C' is the set of all immediate constraints in C. The notation $\pi_{\text{im}}(S)$ holds the *projection* of S on immediate constraints. Similarly, let $\pi_{\text{al}}(S)$ be the *projection* of S on alternating constraints.

Given a schema $S = (A, C)$, if $\pi_{\text{im}}(S)$ satisfies the conditions stated in Theorem 7.3.5, then for each activity $a \in A$, denote $\bar{\text{s}}_{\text{im}}(a)$ as a string constructed iteratively as follows: (i) $\bar{\text{s}}_{\text{im}}(a) = a$ initially, (ii) for the leftmost (or rightmost) activity u of $\bar{\text{s}}_{\text{im}}(a)$, if there exists $v \in A$ such that $iPre(u, v) \in C$ (resp. $iRes(u, v) \in C$), then update $\bar{\text{s}}_{\text{im}}(a)$ to be $v\bar{\text{s}}_{\text{im}}(a)$ (resp. $\bar{\text{s}}_{\text{im}}(a)v$), i.e., prepend (resp. append) $\bar{\text{s}}_{\text{im}}(a)$ with v, and (iii) repeat step (ii) until no more changes can be made. For each $a \in A$, $\bar{\text{s}}_{\text{im}}(a)$ is unique due to Conditions (2) and (3) of Theorem 7.3.5 and is finite due to Condition (1). Let $\text{s}_{\text{im}}(a)$ be the set of activities that occur in $\bar{\text{s}}_{\text{im}}(a)$.

Proof: Theorem 7.3.5 Let $A, C, E^{\text{im}}_{\blacktriangleleft}$, and $E^{\text{im}}_{\blacktriangleright}$ be as stated in the theorem.

($\Rightarrow$) Condition (1) follows from Theorem 7.3.2 and Lemma 7.3.4. Suppose Condition (2) does not hold. There exist distinct $u, v, w \in A$ such that either (i) $iRes(u, v)$ and $iPre(v, w)$

are in C, or (ii) $iPre(u, v)$ and $iRes(v, w)$ are in C. Neither pair can be satisfied by a string. Similarly, suppose Condition (3) does not hold. There exist distinct $u, v, w \in A$ such that either (i) $iRes(u, v)$ and $iRes(u, w)$ are in C, or (ii) $iPre(u, v)$ and $iPre(u, w)$ are in C. Again, neither can be satisfied by a string.

($\Leftarrow$) Suppose that $A = \{a_1, a_2, ..., a_n\}$ where $n = |A|$. Notice that each activity in A should occur at least once in a conforming string. It is straightforward to verify that the string $\bar{s}_{\text{im}}(a_1)\bar{s}_{\text{im}}(a_2)...\bar{s}_{\text{im}}(a_n)$ satisfies each constraint in C. ∎

One remark here is that immediate constraints are expressible in the language $\text{LTL}^{\Box,\Diamond}_{krom}$, whose satisfiability under infinite semantics is NP-Complete [65]. Consequently, satisfiability of immediate constraints under the infinite semantics (i.e., by an infinite sequence of activities) is in class NP.

7.3.3 Ordering and immediate constraints

To obtain the syntactic conditions for deciding the conformance of ordering and immediate constraints, one possible candidate can be the condition that combines the characterization for both ordering and immediate constraints (i.e., the conjunction of conditions of Theorems 7.3.2 and 7.3.5). However, such combined condition will fail in terms of the "if" direction. Fortunately, the "if" direction does hold if a preprocessing is performed based on the given ordering and immediate constraints. Thus in this subsection, we first present a preprocessing upon a given schema and then show the syntactic conditions.

Lemma 7.3.7 Given a schema $S = (A, C)$ and its causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, for each $(u, v) \in E^{\text{im}}_{\blacktriangleright} \cup E^{\text{im}}_{\blacktriangleleft}$, if there exists $w \in A - \{u\}$, such that $(v, w) \in E^{\text{or}}_{\blacktriangleleft}$ (or $E^{\text{or}}_{\blacktriangleright}$), then for each conforming string s of S, s satisfies $aPre(u, w)$ (resp. $aRes(u, w)$).

Lemma 7.3.7 is straightforward. Based on Lemma 7.3.7, we can have the following preprocessing given a schema.

Definition: Given a schema $S = (A, C)$, the *immediate-plus* (or im^+) *schema* of S is a schema (A, C') constructed as follows: 1. Initially $C' = C$. 2. Repeat the following steps while C' is changed: for each distinct $u, v, w \in A$, if (1) $iPre(u, v)$ or $iRes(u, v)$ is in C' and (2) $Pre(v, w) \in C'$ (or $Res(v, w) \in C'$), then add $Pre(u, w)$ (resp. $Res(u, w)$ to C'.

It is easy to see that for each given schema, its corresponding im$^+$schema is unique. The following is a consequence of Lemma 7.3.7.

Corollary 7.3.8 A schema is conformable iff its im$^+$schema is conformable.

Before presenting syntactical characterization for schemas with ordering and immediate constraints, we introduce the following notations for reading convenience. Let x, y, z be one of 'or', 'al', 'im'; we denote $E^{\mathrm{x}}_{\blacktriangleright} \cup E^{\mathrm{y}}_{\blacktriangleright}$ as $E^{\mathrm{x}\cup\mathrm{y}}_{\blacktriangleright}$ and use similar notations $E^{\mathrm{x}\cup\mathrm{y}\cup\mathrm{z}}_{\blacktriangleright}$, $E^{\mathrm{x}\cup\mathrm{y}}_{\blacktriangleleft}$, and $E^{\mathrm{x}\cup\mathrm{y}\cup\mathrm{z}}_{\blacktriangleleft}$.

Theorem 7.3.9 Given a schema $S = (A, C)$ where C contains only ordering and immediate constraints, the im$^+$schema S' of S, and the causality graph $(A, E^{\mathrm{or}}_{\blacktriangleright}, E^{\mathrm{or}}_{\blacktriangleleft}, E^{\mathrm{im}}_{\blacktriangleright}, E^{\mathrm{im}}_{\blacktriangleleft})$ of S', S is conformable iff the following conditions all hold.

(1). $(A, E^{\mathrm{or}\cup\mathrm{im}}_{\blacktriangleright})$ and $(A, E^{\mathrm{or}\cup\mathrm{im}}_{\blacktriangleleft})$ are both acyclic,

(2). for each $(u, v) \in E^{\mathrm{im}}_{\blacktriangleright}$ (or $E^{\mathrm{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq u$ and $(v, w) \in E^{\mathrm{im}}_{\blacktriangleleft}$ (resp. $E^{\mathrm{im}}_{\blacktriangleright}$), and

(3). for each $(u, v) \in E^{\mathrm{im}}_{\blacktriangleright}$ (or $E^{\mathrm{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq v$ and $(u, w) \in E^{\mathrm{im}}_{\blacktriangleright}$ (resp. $E^{\mathrm{im}}_{\blacktriangleleft}$).

Example 7.3.10 A schema S has 3 activities, a, b, c, and 4 constraints $iRes(a, c)$, $iRes(b, c)$, $Pre(c, a)$, and $Pre(c, b)$. Let S' be the im$^+$schema of S. Due to $iRes(a, c)$ and $Pre(c, b)$, S' contains $Pre(a, b)$. Similarly, due to $iRes(b, c)$ and $Pre(c, a)$, S' also contains $Pre(b, a)$. Obviously $Pre(a, b)$ and $Pre(b, a)$ lead to non-conformability of S.

Note that conditions (1) – (3) will hold if they are applied on S directly instead of S'. Therefore, a preprocessing to obtain an im$^+$is necessary when determining conformability.

∎

Note that the "only if" direction of Theorem 7.3.9 is a direct result based on Theorems 7.3.2, 7.3.5, and Corollary 7.3.8. Therefore, in the remainder of this subsection, we focus on the proof of the "if" direction.

To discuss the "if" direction of Theorem 7.3.9, we first describe an algorithm to construct a string. And it will be proved in Lemma 7.3.11 that if a schema satisfies all conditions in Theorem 7.3.9, the string constructed by this algorithm satisfies all constraints in the schema.

Let S be a schema, S' the im$^+$schema of S, and $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ be the causality graph of S' as stated in Theorem 7.3.9. Suppose further that S satisfies all conditions in Theorem 7.3.9. We construct a string according to Alg. 5.

The main idea of Alg. 5 is similar to the construction in the proofs of Theorems 7.3.2 and 7.3.5, i.e., to rely on a topological order of both the "precedence" and "response" directions (to satisfy the ordering constraints); then replace each activity a by $\bar{\text{s}}_{\text{im}}(a)$ (in order to satisfy the immediate constraints). A subtlety is that if Alg. 5 is directly applied an arbitrary schema with only ordering and immediate constraints instead of its im$^+$schema version, then the constructed string may not be conformable as suggested in Example 7.3.10.

Algorithm 5

Input: A causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ of an im$^+$schema of a schema S that satisfies all conditions in Theorem 7.3.9

Output: A finite string that conforms to S

A. Let "$a_1a_2...a_n$" and "$b_1b_2...b_n$" be topological sequences of $(A, E^{\text{or}\cup\text{im}}_{\blacktriangleright})$ and $(A, E^{\text{or}\cup\text{im}}_{\blacktriangleleft})$, resp.

B. Return the string "$\bar{\text{s}}_{\text{im}}(b_n)...\bar{\text{s}}_{\text{im}}(b_1)\bar{\text{s}}_{\text{im}}(a_1)...\bar{\text{s}}_{\text{im}}(a_n)$".

Lemma 7.3.11 The "if" direction of Theorem 7.3.9 holds.

Proof: Let $S, S'A, C, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}$, and $E^{\text{im}}_{\blacktriangleleft}$ be as stated in Theorem 7.3.9 and S' satisfies Conditions (1)–(3). The proof is to show that the output string of Alg. 5 conforms to S. This is done by an analysis of Alg. 5 on input $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$.

Based on Step B of Alg. 5, it is straightforward to prove the returned string satisfies every immediate constraint in C. Therefore in the remainder of this proof, we focus on ordering constraints only.

According to the similar techniques used in the proof of Theorem 7.3.2, The string s constructed at Step A of Alg. 5 satisfies every ordering constraint in C. Suppose the string s' constructed at Step B does not satisfy every ordering constraint. Without loss of generality, suppose s' violates $\mathit{Res}(a, b)$, where $a, b \in A$ (the case $\mathit{Pre}(a, b)$ is symmetric). As $s \models \mathit{Pre}(a, b)$ and s' is obtained by replacing each activity d in s by $\bar{\text{s}}_{\text{im}}(d)$, there must exist $c \in A$ such that after some occurrence ω of c in s, there is no occurrence of b, and $a \in \text{s}_{\text{im}}(c)$. However, as $a \in \text{s}_{\text{im}}(c)$ and $(a, b) \in E^{\text{or}}_{\blacktriangleright}$, we have $(c, b) \in E^{\text{or}}_{\blacktriangleright}$ according to Lemma 7.3.7. Thus, in s, there must exist an occurrence of b after ω according to the topological order restricted in Step A, a contradiction. Hence, s' satisfies each ordering constraint in C.

Since each activity occurs in s, the lemma holds. ∎

7.4 Incorporating Alternating Constraints

This section focuses on syntactical conditions for conformance of schemas that contain alternating constraints (and possibly other constraints).

We begin with defining "pre-processing" for schemas such that the original schema is conformable if and only if the schema after the pre-processing also is.

Given a set of activities A, an activity a, and a string s over A, denote by $\#_a(s)$ the number of occurrences of a in s.

Lemma 7.4.1 Given a set A of activities, a string s over A, and $u, v \in A$, if s satisfies $aRes(u,v)$ or $aPre(u,v)$, then $\#_u(s) \leqslant \#_v(s)$.

Proof: Suppose s satisfies $aRes(u,v)$ ($aPre(u,v)$ is similar). If u does not occur in s, $\#_u(s) = 0$ and the lemma holds. Now assume that u occurs in s at least once. Let i be the least such that $s^{[i]} = u$ (the first occurrence of u), and σ be the suffix of s such that $s = s^{[1]} \cdots s^{[i-1]}\sigma$. Clearly, $\#_u(s) = \#_u(\sigma)$ and σ also satisfies $aRes(u,v)$. If $\#_u(\sigma) > \#_v(\sigma)$, either the last occurrence of u in σ is not followed by any occurrences of v or there are two occurrences of u in σ having no occurrences of v in-between, contradicting to σ satisfying $aRes(u,v)$. Thus, $\#_u(s) = \#_u(\sigma) \leqslant \#_v(\sigma) \leqslant \#_v(s)$. ∎

Lemma 7.4.2 Given a schema $S = (A, C)$ and its causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, for each $(u,v) \in E^{\text{al}}_{\blacktriangleright}$ (or $E^{\text{al}}_{\blacktriangleleft}$), if u and v are on a common cycle in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$, then for each conforming string s of S, s satisfies $aPre(v,u)$ (resp. $aRes(v,u)$).

Proof: Let $S = (A, C)$, $E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, s, u$, and v be as stated in the lemma. We consider the case when $(u,v) \in E^{\text{al}}_{\blacktriangleright}$ (the argument for $E^{\text{al}}_{\blacktriangleleft}$ is similar). The proof consists of two steps: We first show that $s \models Pre(v,u)$, and then prove that u and v are "alternating" in s.

$\underline{s \models Pre(v,u)}$ Due to $(u,v) \in E^{\text{al}}_{\blacktriangleright}$, $s \models aRes(u,v)$. Suppose s does not satisfy $Pre(v,u)$. There exists $n \in [1..len(s)]$ such that $s^{[n]} = v$ and for each $i \in [1..(n-1)]$, $s^{[i]} \neq u$. Therefore, by removing $s^{[n]}$ from s, $s' = s^{[1]}s^{[2]}...s^{[n-1]}s^{[n+1]}...s^{[len(s)]}$ still satisfies $aRes(u,v)$. By Lemma 7.4.1, $\#_u(s') \leqslant \#_v(s')$. Since $s^{[n]} = v$, $\#_u(s) < \#_v(s)$. As u and v are on some cycle in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$, by Lemma 7.4.1, $\#_u(s) = \#_v(s)$, a contradiction.

$\underline{u \text{ and } v \text{ are alternating}}$ It suffices to prove that between two occurrences of v in s, there exists an occurrence of u. Suppose that this is not the case, i.e., there exist $m, n \in [1..len(s)]$, such that $m < n$, $s^{[m]} = s^{[n]} = v$, and $s^{[i]} \neq u$ for each $m < i < n$. By removing $s^{[n]}$ from s, $s' = s^{[1]}s^{[2]}...s^{[n-1]}s^{[n+1]}...s^{[len(s)]}$ still satisfies $aRes(u,v)$. A contradiction can be derived using an analysis similar to the first step. ∎

Definition: Given a schema $S = (A, C)$ and its causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, the *alternating-plus* (or al^+) *schema* of S is a schema (A, C') where

$$C' = C \cup \{aPre(v,u) \mid (u,v) \in E^{\text{al}}_{\blacktriangleright}, u \text{ and } v \text{ are on a common cycle in } (A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})\}$$
$$\cup \{aRes(v,u) \mid (u,v) \in E^{\text{al}}_{\blacktriangleleft}, u \text{ and } v \text{ are on a common cycle in } (A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})\}$$

It is easy to see that for each given schema, its corresponding al$^+$schema is unique. The following is a consequence of Lemma 7.4.2.

Corollary 7.4.3 A schema S is conformable iff its al$^+$schema is conformable.

7.4.1 Ordering and alternating constraints

Theorem 7.4.4 below addresses the case when only ordering and alternating constraints are used in schemas.

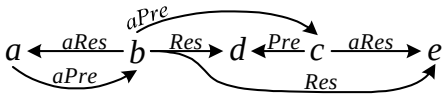

Figure 7.2: An al$^+$schema example

Theorem 7.4.4 Given a schema S that only contains ordering and alternating constraints, let $S' = (A, C)$ be the al$^+$schema of S and $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ the causality graph of S'. S is conformable iff both $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleright})$ and $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleleft})$ are acyclic.

Example 7.4.5 Consider a schema with 5 activities, a, b, c, d, e, and constraints in the form of a graph $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleright} \cup E^{\text{or}\cup\text{al}}_{\blacktriangleleft})$ as shown in Fig. 7.2, where the edge labels denote constraint types. Note that its al$^+$schema is itself.

Note that both conditions in Theorem 7.4.4 are satisfied, thus the schema is conformable. A conforming string is *dcebadce*. If we add the constraint $aPre(d, b)$ into the schema, it is no longer conformable since *bcd* forms a cycle in $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleleft})$, forcing the subsequence *bcd* to occur infinitely many times. ∎

The "only if" direction of Theorem 7.4.4 directly follows from Theorem 7.3.2, Lemma 7.3.4, and Corollary 7.4.3.

In the remainder of this subsection, we focus on the "if" direction, which is a bit involved. The main idea is to construct an algorithm (Alg. 6) that produces a conforming string of the input al$^+$schema (Lemma 7.4.12). A key step of the algorithm is to first create a topological order of precedence constraints and that of response constraints, then for each violated alternating constraint, insert a string to fix the violation.

Example 7.4.6 Continue on Example 7.4.5 with the schema in Fig. 7.2. The schema

Algorithm 6

Input: The causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ of an al$^+$schema S satisfying conditions of Theorem 7.4.4

Output: A string that conforms to S

A. Let $s_{\blacktriangleright} = a_1 a_2 ... a_n$ be a topological order of $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleright})$ and $s_{\blacktriangleleft} = b_n b_{n-1} ... b_1$ a reversed topological order of $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleleft})$.

B. For each $a \in A$, define $\text{R}(a)$ as the set of nodes in A reachable from a through edges in $E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft}$ (i.e., each $b \in \text{R}(a)$ is either a itself or reachable from a in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$), and denote $\bar{\text{R}}_{\blacktriangleright}(a)$ and $\bar{\text{R}}_{\blacktriangleleft}(a)$ the two enumerations of $\text{R}(a)$ such that $\bar{\text{R}}_{\blacktriangleright}(a)$ and $\bar{\text{R}}_{\blacktriangleleft}(a)$ are subsequences of $s_{\blacktriangleright}$ and $s_{\blacktriangleleft}$, resp.

C. Let $V_{\text{ns}} \subseteq C$ be the set of alternating constraints that are not satisfied by $s_{\blacktriangleleft} s_{\blacktriangleright}$, and $E_{\text{ns}} \subseteq V_{\text{ns}} \times V_{\text{ns}}$ such that an edge $(X(a,b), Y(c,d))$ is in E_{ns} iff $c \in \text{R}(b)$, where $X, Y \in \{aRes, aPre\}$ and $a, b, c, d \in A$. Denote $\bar{v}_{\text{ns}}$ to be a topological order of $(V_{\text{ns}}, E_{\text{ns}})$. (It will be shown in Lemma 7.4.11 that $(V_{\text{ns}}, E_{\text{ns}})$ is acyclic.)

D. For each edge $aRes(u,v)$ (or $aPre(u,v)$) in V_{ns} in the order of $\bar{v}_{\text{ns}}$, let $s_{\blacktriangleleft} = s_{\blacktriangleleft} \bar{\text{R}}_{\blacktriangleleft}(v)$ (resp. $s_{\blacktriangleright} = \bar{\text{R}}_{\blacktriangleright}(v) s_{\blacktriangleright}$).

E. Return $s_{\blacktriangleleft} s_{\blacktriangleright}$.

satisfies all conditions in Theorem 7.4.4. We now demonstrate a run of Alg. 6 on this schema.

A. $s_{\blacktriangleright} = bceda$ is a topological sequence of $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleright})$ where $E^{\text{or}\cup\text{al}}_{\blacktriangleright} = \{(b,a), (b,d), (b,e), (c,e)\}$ and $s_{\blacktriangleleft} = edcba$ is a reversed topological sequence of $(A, E^{\text{or}\cup\text{al}}_{\blacktriangleleft})$ where $E^{\text{or}\cup\text{al}}_{\blacktriangleleft} = \{(a,b), (b,c), (c,d)\}$.

B. $E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft} = \{(a,b), (b,c), (b,a), (c,e)\}$, $\text{R}(a) = \{a,b,c,e\}$. Hence $\bar{\text{R}}_{\blacktriangleright}(a) = bcea$ and $\bar{\text{R}}_{\blacktriangleleft}(a) = ecba$ (subsequences of $s_{\blacktriangleright}$ and $s_{\blacktriangleleft}$, resp.). Similarly, $\bar{\text{R}}_{\blacktriangleright}(b) = bcea$, $\bar{\text{R}}_{\blacktriangleleft}(b) = ecba$, $\bar{\text{R}}_{\blacktriangleright}(c) = ce$, $\bar{\text{R}}_{\blacktriangleleft}(c) = ec$, $\bar{\text{R}}_{\blacktriangleright}(d) = \bar{\text{R}}_{\blacktriangleleft}(d) = d$, and $\bar{\text{R}}_{\blacktriangleright}(e) = \bar{\text{R}}_{\blacktriangleleft}(e) = e$.

C. Now $s_{\blacktriangleleft} s_{\blacktriangleright} = edcbabceda$ violates $aRes(c,e)$ and $aPre(b,c)$. Thus $V_{\text{ns}} = \{aRes(c,e), aPre(b,c)\}$. Since $c \in \text{R}(c)$, $E_{\text{ns}} = \{(aPre(b,c), aRes(c,e))\}$. Note that $(V_{\text{ns}}, E_{\text{ns}})$ is acyclic. $\bar{v}_{\text{ns}} = aPre(b,c)\ aRes(c,e)$.

D. According to $\bar{v}_{\text{ns}}$, $aPre(b,c)$ is considered first. We replace $s_{\blacktriangleleft}$ by $s_{\blacktriangleleft} \bar{\text{R}}_{\blacktriangleleft}(c) = edcbaec$. Now $s_{\blacktriangleleft} s_{\blacktriangleright} = edcbaecbceda$ and satisfies $aPre(b,c)$, but still violates $aRes(c,e)$. We then replace $s_{\blacktriangleright}$ by $\bar{\text{R}}_{\blacktriangleright}(e) s_{\blacktriangleright} = ebceda$. Now, $s_{\blacktriangleleft} s_{\blacktriangleright} = edcbaecebceda$ and satisfies $aRes(c,e)$.

E. The final result for $s_{\blacktriangleleft}s_{\blacktriangleright}$ is *edcbaecebceda*.

It can be easily verified that the resulting string *edcbaecebceda* satisfies every constraint in the input schema and contains every activity. ∎

Lemma 7.4.7 Let $S=(A,C)$ be an al$^+$schema that satisfies all conditions in Theorem 7.4.4 and $\mathcal{G}_S=(A,E^{\text{or}}_{\blacktriangleright},E^{\text{or}}_{\blacktriangleleft},E^{\text{al}}_{\blacktriangleright},E^{\text{al}}_{\blacktriangleleft})$ its casualty graph, $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ the strings constructed in Step A of Alg. 6 on $\mathcal{G}_S$. Then for each (possibly empty) string τ over A, $s_{\blacktriangleleft}\tau s_{\blacktriangleright}$ satisfies each ordering constraint in C.

Proof: Let $A, c, \tau, E^{\text{al}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleright}, s_{\blacktriangleleft}$, and $s_{\blacktriangleright}$ be as stated in the lemma. If $\mathit{Res}(u,v)\in C$, we prove that $s_{\blacktriangleleft}\tau s_{\blacktriangleright}$ satisfies $\mathit{Res}(u,v)$ (the $\mathit{Pre}(u,v)$ case is similar).

Since $s_{\blacktriangleright}$ is a topological sequence of $(A,E^{\text{or}\cup\text{al}}_{\blacktriangleright})$ and $E^{\text{or}}_{\blacktriangleright}\subseteq E^{\text{or}\cup\text{al}}_{\blacktriangleright}$, $s_{\blacktriangleright}$ is also a topological sequence of $(A,E^{\text{or}}_{\blacktriangleright})$. Thus, $s_{\blacktriangleright}$ satisfies $\mathit{Res}(u,v)$. And it is easy to verify that $s_{\blacktriangleleft}\tau s_{\blacktriangleright}$ satisfies $\mathit{Res}(u,v)$, since no u in $s_{\blacktriangleleft}\tau$ can occur after the v in $s_{\blacktriangleright}$. ∎

Corollary 7.4.8 Let S be an al$^+$schema that satisfies all conditions in Theorem 7.4.4 and $\mathcal{G}_S$ its casualty graph. The string returned by Alg. 6 on input $\mathcal{G}_S$ satisfies every ordering constraint in S.

Proof: Let $S=(A,C)$ and $\mathcal{G}_S$ be as stated in the Corollary and let $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ be the strings constructed in Step A of Alg. 6 on $\mathcal{G}_S$. Based on Lemma 7.4.7, for each string τ over A, $s_{\blacktriangleleft}\tau s_{\blacktriangleright}$ satisfies each ordering constraint in C. Moreover, according to Step D of Alg. 6, the returned string of Alg. 6 will be of form $s_{\blacktriangleleft}ts_{\blacktriangleright}$, where t is a string over A; therefore, the returned string of Alg. 6 on $\mathcal{G}_S$ satisfies every ordering constraint in C. ∎

We now turn to alternating constraints. Given a set A of activities, c an alternating constraint of form $a\mathit{Res}(u,v)$ (or $a\mathit{Pre}(u,v)$) where $u,v\in A$, and s some nonempty string

over A that satisfies $Res(u, v)$ (resp. $Pre(u, v)$) but not c. A pair of distinct integers $1 \leqslant i < j \leqslant len(s)$ is *violating in s wrt c* if $s^{[i]} = s^{[j]} = u$ and for each $i < k < j$, $s^{[k]} \neq v$ (no occurrences of v between two occurrences of u).

Lemma 7.4.9 Let A be a set of activities, c an alternating constraint of form $aRes(u, v)$ (or $aPre(u, v)$) where $u, v \in A$, and s a string over A that satisfies $Res(u, v)$ (resp. $Pre(u, v)$) and contains exactly one violating pair (μ_1, μ_2) wrt c. If $s = s_1 s_2$, $\mu_1 \leqslant len(s_1) < \mu_2$, and τ is a string over A containing at least one occurrence of v but no occurrences of u, then $s_1 \tau s_2 \models c$.

Proof: The proof is straightforward. We only consider the case when c is $Res(u, v)$; the case $Pre(u, v)$ is symmetric. Since τ does not contain u, there is no occurrence of u between μ_1 and $\mu_2 + len(\tau)$ in $s_1 \tau s_2$. Moreover, since τ contains an occurrence of v, $(\mu_1, \mu_2 + len(\tau))$ is not a violating pair in $s_1 \tau s_2$ wrt c. Due to the assumption that (μ_1, μ_2) is the only violating pair in $s = s_1 s_2$, $s_1 \tau s_2$ contains no violating pair wrt c. Further, as $s = s_1 s_2$ satisfies $Res(u, v)$ and τ contains no u, $s_1 \tau s_2$ satisfies $Res(u, v)$; and thus satisfies $c = aRes(u, v)$. ∎

Lemma 7.4.10 Let $S = (A, C)$ be an al$^+$schema that satisfies conditions in Theorem 7.4.4, $\mathcal{G}_S$ the casualty graph of S, $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ the strings constructed in Step A of Alg. 6 on $\mathcal{G}_S$, and $\mathrm{R}(a)$ (where $a \in A$) the set of nodes constructed in Step B of Alg. 6 on $\mathcal{G}_S$. Then, for each $aRes(u, v)$ or $aPre(u, v)$ in C that is not satisfied by $s_{\blacktriangleleft} s_{\blacktriangleright}$, $u \notin \mathrm{R}(v)$.

Proof: Let $S = (A, C), \mathcal{G}_S = (A, E^{\mathrm{or}}_{\blacktriangleright}, E^{\mathrm{or}}_{\blacktriangleleft}, E^{\mathrm{al}}_{\blacktriangleright}, E^{\mathrm{al}}_{\blacktriangleleft}), s_{\blacktriangleleft}, s_{\blacktriangleright}$, and $\mathrm{R}(a)$ (where $a \in A$) be as stated in the lemma. We argue that for each $aRes(u, v) \in C$ that is not satisfied by $s_{\blacktriangleleft} s_{\blacktriangleright}$, $u \notin \mathrm{R}(v)$ (the $aPre(u, v)$ case is similar).

Suppose that $u \in \mathrm{R}(v)$; together with $(u, v) \in E^{\mathrm{al}}_{\blacktriangleright}$ (due to $aRes(u, v)$), we have u and v in a common cycle in $(A, E^{\mathrm{al}}_{\blacktriangleright} \cup E^{\mathrm{al}}_{\blacktriangleleft})$. Thus $aPre(v, u)$ is in C since S is an al$^+$schema; and therefore, u occurs before v in $s_{\blacktriangleleft}$. As u occurs before v in $s_{\blacktriangleright}$ based on the topological order, $s_{\blacktriangleleft}s_{\blacktriangleright}$ satisfies $aRes(u, v)$, a contradiction. ∎

Lemma 7.4.11 Let $S = (A, C)$ be an al$^+$schema that satisfies all conditions in Theorem 7.4.4, $\mathcal{G}_S$ the casualty graph of S, $\mathrm{R}(a)$ (where $a \in A$) the set of nodes constructed in Step B of Alg. 6 on $\mathcal{G}_S$, and $(V_{\mathrm{ns}}, E_{\mathrm{ns}})$ the graph constructed in Step C of Alg. 6 on $\mathcal{G}_S$. Then, $(V_{\mathrm{ns}}, E_{\mathrm{ns}})$ is acyclic.

Proof: Let $S = (A, C), \mathcal{G}_S = (A, E^{\mathrm{or}}_{\blacktriangleright}, E^{\mathrm{or}}_{\blacktriangleleft}, E^{\mathrm{al}}_{\blacktriangleright}, E^{\mathrm{al}}_{\blacktriangleleft}), V_{\mathrm{ns}}, E_{\mathrm{ns}}$, and $\mathrm{R}(a)$ (where $a \in A$) be as stated in the lemma, $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ the strings constructed in Step A of Alg. 6 on $\mathcal{G}_S$. Note that each constraint in V_{ns} is not satisfied by $s_{\blacktriangleleft}s_{\blacktriangleright}$ according to Step C Alg. 6.

Suppose that $(V_{\mathrm{ns}}, E_{\mathrm{ns}})$ is not acyclic. Then, there exist $X(a, b), Y(c, d) \in V_{\mathrm{ns}}$, where $X, Y \in \{aRes, aPre\}$ and $a, b, c, d \in A$, such that $c \in \mathrm{R}(b)$ and $a \in \mathrm{R}(d)$. Thus, there is a path from b to c and there is a path from d to a in $(A, E^{\mathrm{al}}_{\blacktriangleright} \cup E^{\mathrm{al}}_{\blacktriangleleft})$. As a result, a, b, c, and d are on a common cycle in $(A, E^{\mathrm{al}}_{\blacktriangleright} \cup E^{\mathrm{al}}_{\blacktriangleleft})$. Therefore, $a \in \mathrm{R}(b)$ and $c \in \mathrm{R}(d)$. Since $X(a, b)$ and $Y(c, d)$ are not satisfied by $s_{\blacktriangleleft}s_{\blacktriangleright}$, according to Lemma 7.4.10, $a \notin \mathrm{R}(b)$ and $c \notin \mathrm{R}(d)$, a contradiction. ∎

Lemma 7.4.12 The "if" direction of Theorem 7.4.4 holds.

Proof: According to Corollary 7.4.8, all ordering constraints have been taken care of; thus, we only need to focus on alternating constraints. The main idea is to prove by induction upon each iteration in Step D of Alg. 6 that each iteration will "fix" a violation

of constraint and the fix of each violated constraint will not make other satisfied or violated constraints "become worse".

Let S, $S' = (A, C)$ and $\mathcal{G}_{S'} = (A, E^{\text{or}}_{\blacktriangleright}, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ be as stated in Theorem 7.4.4. Our goal is to show that if all conditions in the theorem hold, the string s constructed by Alg. 6 on input $\mathcal{G}_{S'}$ satisfies every constraint in C.

Corollary 7.4.8 states that s satisfies each ordering constraint in C. We now only focus on alternating constraints in C.

Combining Corollary 7.4.8 and Lemma 7.3.4, for each alternating constraint $aRes(u, v)$ or $aPre(u, v)$ (where $u, v \in A$), s satisfies $Res(u, v)$ or $Pre(u, v)$ (resp.). What remains to prove is that between each two occurrences of u in s, there exists an occurrence of v.

Let $\textsc{r}(a)$, $\bar{\textsc{r}}_{\blacktriangleright}(a)$, $\bar{\textsc{r}}_{\blacktriangleleft}(a)$ (where $a \in A$), $\bar{v}_{\text{ns}}$ $s_{\blacktriangleleft}$, and $s_{\blacktriangleright}$, be stated in Alg. 6. We show by induction upon each iteration in Step D that each iteration will "fix" a violation of constraint and the fix of each violated constraint will not make other satisfied or violated constraints "become worse".

For notation convenience, for each $i > 0$, denote $\bar{v}_{\text{ns}}(i)$ to be the i^{th} constraint in $\bar{v}_{\text{ns}}$ (i.e., the constraint to be processed during the i^{th} iteration). Denote $s^0_{\blacktriangleleft}$ and $s^0_{\blacktriangleright}$ as the $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ respectively right before the first iteration. And for each $i > 0$, denote $s^i_{\blacktriangleleft}$ and $s^i_{\blacktriangleright}$ to be the constructed $s_{\blacktriangleleft}$ and $s_{\blacktriangleright}$ respectively right after the i^{th} iteration ($i > 0$). Formally, we prove by induction that the following three properties should hold wrt the i^{th} iteration:

(i). For each $j \in [1..i]$, $s^i_{\blacktriangleleft} s^i_{\blacktriangleright}$ satisfies $\bar{v}_{\text{ns}}(j)$,

(ii). If $s^0_{\blacktriangleleft} s^0_{\blacktriangleright}$ satisfies an alternating constraint c in C, then both $s^i_{\blacktriangleleft}$ and $s^i_{\blacktriangleright}$ satisfies c.

(iii). For each alternating constraint $c \in C$, there is at most one violating pair (μ_1, μ_2) in $s^i_{\blacktriangleleft} s^i_{\blacktriangleright}$ wrt c, where $1 \leqslant \mu_1 \leqslant len(s^i_{\blacktriangleleft}) < \mu_2 \leqslant len(s^i_{\blacktriangleleft} s^i_{\blacktriangleright})$.

Basis: Consider the case when $i = 0$.

• Correctness of property (i): Self-explanatory (i.e., $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ satisfies nothing in $\bar{v}_{\rm ns}$)

• Correctness of property (ii): This property is easy to verify for $i = 0$. Since each activity in A occurs exactly once in both $s^0_{\blacktriangleright}$ and $s^0_{\blacktriangleleft}$, the only way for an alternating constraint to be satisfied by $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ is to be satisfied by both $s^0_{\blacktriangleright}$ and $s^0_{\blacktriangleleft}$.

• Correctness of property (iii): Since each activity in A occurs exactly once in both $s^0_{\blacktriangleright}$ and $s^0_{\blacktriangleleft}$, each alternating constraint that is not satisfied by $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ can have at most one violating pair, where the two corresponding activities occur in $s^0_{\blacktriangleright}$ and $s^0_{\blacktriangleleft}$ respectively. For those alternating constraints that are satisfied by $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$, apparently there is no violating pairs. Therefore, property (iii) holds for $i = 0$.

Hypothesis: Suppose that properties (i), (ii), and (iii) hold wrt the $(i-1)^{\rm th}$ iteration, where $i - 1 \geqslant 0$.

Induction: Consider the $i^{\rm th}$ iteration. Suppose that $\bar{v}_{\rm ns}(i) = aRes(u, v)$, where $u, v \in A$ (for $aPre(u, v) \in C$, the analysis is similar).

• Correctness of property (i): We separate this property into two parts: (A) $s^i_{\blacktriangleleft}s^i_{\blacktriangleright}$ satisfies $\bar{v}_{\rm ns}(i) = aRes(u, v)$, and (B) For each $j \in [1..(i-1)]$, $s^i_{\blacktriangleleft}s^i_{\blacktriangleright}$ satisfies $\bar{v}_{\rm ns}(j)$; and prove each of them in the following.

- Part (A): According to Step C in Alg. 6, $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ does not satisfy $aRes(u, v)$. Therefore, based on Lemma 7.4.10, $u \notin \textsc{r}(v)$; thus, $\bar{\textsc{r}}_{\blacktriangleright}(v)$ does not contain u (where apparently $\bar{\textsc{r}}_{\blacktriangleright}(v)$ contains v).

Two cases should be addressed: (1) $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ does not satisfy $aRes(u, v)$ and (2) $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ satisfies $aRes(u, v)$.

(1). Consider when $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ does not satisfy $aRes(u, v)$. Since the only violating pair (μ_1, μ_2) in $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ wrt $aRes(u, v)$ satisfies $1 \leqslant \mu_1 \leqslant len(s^{i-1}_{\blacktriangleleft}) < \mu_2 \leqslant len(s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright})$ according to the property (iii) in the hypothesis, based on Lemma 7.4.9, $s^i_{\blacktriangleleft}s^i_{\blacktriangleright} = s^{i-1}_{\blacktriangleleft}\bar{\textsc{r}}_{\blacktriangleright}(v)s^{i-1}_{\blacktriangleright}$ satisfies $aRes(u, v)$.

(2). Consider when $s_{\blacktriangleleft}^{i-1}s_{\blacktriangleright}^{i-1}$ satisfies $aRes(u,v)$. Due to the fact that $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ does not contain u, it is easy to verify that $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$ satisfies $aRes(u,v)$.

As a result, for both cases (1) and (2), Part (A) holds.

- Part (B): For each $j \in [1..(i-1)]$, suppose $\bar{v}_{\mathrm{ns}}(j)$ is $aRes(x,y)$, where $x, y \in A$. (For $aPre(x,y)$, the analysis is similar).

Let V_{ns} and E_{ns} be as stated in Step C of Alg. 6 on $\mathcal{G}_S$. Recall that $\bar{v}_{\mathrm{ns}}(i)$ is $aRes(u,v)$; it can be shown that $x \notin \mathrm{R}(v)$ (otherwise there is an edge from $aRes(u,v)$ to $aRes(x,y)$ in graph $(V_{\mathrm{ns}}, E_{\mathrm{ns}})$, which contradicts the order of $\bar{v}_{\mathrm{ns}}$ according to Step C of Alg. 6). Therefore, x does not occur in $\bar{\mathrm{R}}_{\blacktriangleright}(v)$. Since $s_{\blacktriangleleft}^{i-1}s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x,y)$ according to the property (i) in the hypothesis, $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x,y)$ as well. Therefore, Part (B) holds.

In summary, property (i) holds.

• Correctness of property (ii): For each $aRes(x,y)$ in C, where $x, y \in C$, if $s_{\blacktriangleleft}^{0}s_{\blacktriangleright}^{0}$ satisfies $aRes(x,y)$. We prove that both $s_{\blacktriangleleft}^{i}$ and $s_{\blacktriangleright}^{i}$ satisfy $aRes(x,y)$ as well. (For case $aPre(x,y)$, the proof is similar).

Since $s_{\blacktriangleleft}^{0}s_{\blacktriangleright}^{0}$ satisfies $aRes(x,y)$, x occur before y in both $s_{\blacktriangleleft}^{0}$ and $s_{\blacktriangleright}^{0}$. Thus, for each $a \in A$, either $\mathrm{R}(a)$ contains (1) only y but not x, (2) neither x nor y, or (3) both x and y, where x occurs before y in both $\bar{\mathrm{R}}_{\blacktriangleright}(a)$ and $\bar{\mathrm{R}}_{\blacktriangleleft}(b)$ due to the fact that $\bar{\mathrm{R}}_{\blacktriangleright}(a)$ and $\bar{\mathrm{R}}_{\blacktriangleleft}(b)$ are subsequences of $s_{\blacktriangleleft}^{0}$ and $s_{\blacktriangleright}^{0}$ respectively. Note that $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ cannot only contain x but not y, since y is reachable from x in $(A, E_{\blacktriangleright}^{\mathrm{al}} \cup E_{\blacktriangleleft}^{\mathrm{al}})$.

Moreover, based on the property (ii) in the hypothesis, both $s_{\blacktriangleleft}^{i-1}$ and $s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x,y)$. Then $s_{\blacktriangleleft}^{i} = s_{\blacktriangleleft}^{i-1}$ satisfies $aRes(x,y)$. For $s_{\blacktriangleleft}^{i} = \bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleleft}^{i-1}$, it can be easily verified that in one of the following three cases, $s_{\blacktriangleleft}^{i} = \bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleleft}^{i-1}$ satisfies $aRes(x,y)$: (1) $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains only y but not x, (2) $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains neither x nor y, or $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains both x and y, where x occurs before y in $\bar{\mathrm{R}}_{\blacktriangleright}(v)$. Thus property (ii) holds.

• Correctness of property (iii): Consider each alternating constraint c in C. If $s^i_{\blacktriangleleft}s^i_{\blacktriangleright}$ satisfies c, apparently there is violating pair in $s^i_{\blacktriangleleft}s^i_{\blacktriangleright}$ wrt c (resp. $aRes(x,y)$); and thus property (iii) holds. Thus, in the following we assume that $s^i_{\blacktriangleleft}s^i_{\blacktriangleright}$ does not satisfies c.

Before showing the correctness of property (iii), we need to make an important claim that will be used in the remainder of this proof. Let c be $aRes(x,y)$ or $aPre(x,y)$ in C, where $x, y \in A$. Now consider $\text{R}(v)$; recall that according to Step B in Alg. 6, $\text{R}(v)$ contains all the reachable nodes in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$. Therefore, $\bar{\text{R}}_{\blacktriangleright}(v)$ can contain: (1) y but not x, (2) neither x nor y, or (3) both x and y. Note that $\bar{\text{R}}_{\blacktriangleright}(v)$ cannot only contain x but not y, since y is reachable from x in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$. Consider the case (3) when x and y both occur in $\bar{\text{R}}_{\blacktriangleright}(v)$; two cases can be obtained:

(a). $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ does not satisfy $aRes(x,y)$; since the order of $\bar{\text{R}}_{\blacktriangleright}(v)$ is consistent with the order of $s^0_{\blacktriangleright}$ and x occurs before y in $s_{\blacktriangleright}$ due to $(x,y) \in E^{\text{al}}_{\blacktriangleright}$, x occurs before y in $\bar{\text{R}}_{\blacktriangleright}(v)$.

(b). $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ does not satisfy $aPre(x,y)$; Since $(x,y) \in E^{\text{al}}_{\blacktriangleleft}$, y occurs before x in $s^0_{\blacktriangleleft}$; therefore, x must occur before y in $s^0_{\blacktriangleright}$ in order to make $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ not satisfying $aPre(x,y)$. As a result, x occurs before y in $\bar{\text{R}}_{\blacktriangleright}(v)$.

Thus, in both cases (a) and (b), if x and y both occur in $\bar{\text{R}}_{\blacktriangleright}(v)$, x occurs before y.

CLAIM: $\bar{\text{R}}_{\blacktriangleright}(v)$ contains either

(1) y but not x (2) neither x nor y, or
(3) both x and y, where x occurs before y in $\bar{\text{R}}_{\blacktriangleright}(v)$

Now we continue the proof. Two cases are to be addressed: (A) $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ does not satisfy c, and (B) $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ satisfies c.

- Case (A): $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ does not satisfy c. According to the property (ii) in the hypothesis, if $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ does satisfies an alternating constraint $c' \in C$, then both $s^{i-1}_{\blacktriangleleft}$ and $s^{i-1}_{\blacktriangleright}$ satisfies c' and thus $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ satisfies c'. As a result, if $s^{i-1}_{\blacktriangleleft}s^{i-1}_{\blacktriangleright}$ does not satisfy c, $s^0_{\blacktriangleleft}s^0_{\blacktriangleright}$ does not satisfy c.

Based on the property (iii) in the hypothesis, there is exactly one violating pair (η_1, η_2) in $s_{\blacktriangleleft}^{i-1}s_{\blacktriangleright}^{i-1}$ wrt c, where $1 \leqslant \eta_1 \leqslant len(s_{\blacktriangleleft}^{i-1}) < \eta_2 \leqslant len(s_{\blacktriangleright}^{i-1})$. For each case (1), (2), or (3) stated in the claim above, we have the following analysis:

(1). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains y but not x; according to Lemma 7.4.9, $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x, y)$ or $aPre(x, y)$.

(2). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains neither x nor y; since for each integer j ranging from $\eta_1 + 1$ to $\eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)) - 1$, $(s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i})^{[j]} = (s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1})^{[j]}$ does not equal x or y, $(\eta_1, \eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)))$ is the only violating pair in $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i}$ wrt $aRes(x, y)$ or $aPre(x, y)$. And Note that $1 \leqslant \eta_1 \leqslant len(s_{\blacktriangleleft}^{i}) < \eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)) \leqslant len(s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i})$.

(3). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains both x and y, where x occurs before y in $\bar{\mathrm{R}}_{\blacktriangleright}(v)$; suppose $(\bar{\mathrm{R}}_{\blacktriangleright}(v))^{[m]} = x$ and $(\bar{\mathrm{R}}_{\blacktriangleright}(v))^{[n]} = y$, where $1 \leqslant m < n \leqslant len(\bar{\mathrm{R}}_{\blacktriangleright}(v))$. Then in $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$, $(\eta_1, \eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)))$ and $(len(s_{\blacktriangleleft}^{i}) + m, \eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)))$ are not violating pairs wrt $aRes(x, y)$ or $aPre(x, y)$, since $\eta_1 < len(s_{\blacktriangleleft}^{i}) + m < len(s_{\blacktriangleleft}^{i}) + n < \eta_2 + len(\bar{\mathrm{R}}_{\blacktriangleright}(v))$ and $(s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i})^{[len(s_{\blacktriangleleft}^{i})+n]} = y$. However, $(\eta_1, len(s_{\blacktriangleleft}^{i}) + m)$ is a violating pair in $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i}$ wrt $aRes(x, y)$ or $aPre(x, y)$, since for each integer j ranging from η_1 to $len(s_{\blacktriangleleft}^{i}) + m$, $(s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i})^{[j]} \neq y$. Therefore $(\eta_1, len(s_{\blacktriangleleft}^{i}) + m)$ is the only violating pair in $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i}$ wrt $aRes(x, y)$ or $aPre(x, y)$. And Note that $1 \leqslant \eta_1 \leqslant len(s_{\blacktriangleleft}^{i}) < len(s_{\blacktriangleleft}^{i})+m \leqslant len(s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i})$.

In all three cases (1), (2), and (3), property (iii) for Case (A) holds.

- Case (B): $s_{\blacktriangleleft}^{i-1}s_{\blacktriangleright}^{i-1}$ satisfies c. For each case (1), (2), or (3) in the claim above, we have the following analysis:

(1). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains y but not x; then $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x, y)$ or $aPre(x, y)$.

(2). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains neither x nor y; then $s_{\blacktriangleleft}^{i}s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1}\bar{\mathrm{R}}_{\blacktriangleright}(v)s_{\blacktriangleright}^{i-1}$ satisfies $aRes(x, y)$ or $aPre(x, y)$.

(3). $\bar{\mathrm{R}}_{\blacktriangleright}(v)$ contains both x and y, where x occurs before y in $\bar{\mathrm{R}}_{\blacktriangleright}(v)$; suppose $(\bar{\mathrm{R}}_{\blacktriangleright}(v))^{[m]}$

$= x$ and $(\bar{\mathrm{R}}_{\blacktriangleright}(v))^{[n]} = y$, where $1 \leqslant m < n \leqslant len(\bar{\mathrm{R}}_{\blacktriangleright}(v))$. Further, let $(s_{\blacktriangleleft}^{i-1})^{[p]} = x$ and $(s_{\blacktriangleright}^{i-1})^{[q]} = x$, where $1 \leqslant p \leqslant len(s_{\blacktriangleleft}^{i-1})$ and $1 \leqslant q \leqslant len(s_{\blacktriangleright}^{i-1})$, such that for each j ranging from p to $len(s_{\blacktriangleleft}^{i-1})$, $(s_{\blacktriangleleft}^{i-1})^{[j]} \neq x$ and each k ranging from 1 to q, $(s_{\blacktriangleright}^{i-1})^{[k]} \neq x$. Thus in $s_{\blacktriangleleft}^{i} s_{\blacktriangleright}^{i} = s_{\blacktriangleleft}^{i-1} \bar{\mathrm{R}}_{\blacktriangleright}(v) s_{\blacktriangleright}^{i-1}$, $(len(s_{\blacktriangleleft}^{i}) + m, len(s_{\blacktriangleleft}^{i}) + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)) + q)$ is not a violating pair wrt $aRes(x, y)$ or $aPre(x, y)$ since $len(s_{\blacktriangleleft}^{i}) + m < len(s_{\blacktriangleleft}^{i}) + n < len(s_{\blacktriangleleft}^{i}) + len(\bar{\mathrm{R}}_{\blacktriangleright}(v)) + q$ and $(s_{\blacktriangleleft}^{i} s_{\blacktriangleright}^{i})^{[len(s_{\blacktriangleleft}^{i})+n]} = y$. Thus, only $(p, len(s_{\blacktriangleleft}^{i}) + m)$ may or may not be a violating pair in $s_{\blacktriangleleft}^{i} s_{\blacktriangleright}^{i}$ wrt $aRes(x, y)$ or $aPre(x, y)$. And Note that $1 \leqslant p \leqslant len(s_{\blacktriangleleft}^{i}) < len(s_{\blacktriangleleft}^{i}) + m \leqslant len(s_{\blacktriangleleft}^{i} s_{\blacktriangleright}^{i})$.

In all three cases (1), (2), and (3), property (iii) for Case (B) holds. In summary, property (iii) holds.

According the above induction, properties (i), (ii), and (iii) are valid. Let s be the string constructed based on Alg. 6 on $\mathcal{G}_S$. Because of property (i), if an alternating constraint is not satisfied by $s_{\blacktriangleleft}^{0} s_{\blacktriangleright}^{0}$, it is satisfied by s; and as a corollary of property (ii), if an alternating constraint is satisfied by $s_{\blacktriangleleft}^{0} s_{\blacktriangleright}^{0}$, it is also satisfied by s. Therefore, after all the iterations finish, s satisfies each alternating constraint in C. (Note that the correctness of property (i) is based on property (iii), thus property (iii) cannot be omitted).

As mentioned at the beginning of this proof that s satisfies each ordering constraint in C, together with that each activity occurs in s at least once, s conforms S'. Based on Corollary 7.4.3, s conforms S. ∎

7.4.2 Immediate and alternating constraints

Before discussing conformity conditions for schemas with alternating and immediate constraints, we introduce a "pre-processing" for the given al$^+$schema such that the original al$^+$schema is conformable if and only if after the pre-processing, the schema is con-

formable.

Lemma 7.4.13 Given an al$^+$schema $S=(A,C)$ that only contains alternating and immediate constraints, the causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ of S, and two activities $u, v \in A$ such that there is a path from v to u in the graph $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$, then (1) $iRes(u,v) \in C$ implies if a string s satisfies $iRes(u,v)$, then $s \models iPre(v,u)$, and (2) $iPre(u,v) \in C$ implies if a string s satisfies $iPre(u,v)$, then $s \models iRes(v,u)$

Proof: Let $S, A, C, s, u, v, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}$, and $E^{\text{im}}_{\blacktriangleleft}$ be as stated in the lemma. We give a proof for (1) since (2) is similar. We show that if s satisfies $iRes(u,v)$, then s satisfies $iPre(v,u)$. Suppose that $iRes(u,v)$ is satisfied by s but $iPre(v,u)$ is not.

Since s violates $iPre(v,u)$, there exists a $j \in [2..len(s)]$, such that $s^{[j]} = v$ and $s^{[j-1]} \neq u$. However, s satisfies $iRes(u,v)$, for each $i \in [1..(len(s)-1)]$, if $s^{[i]} = u$, then $s^{[i+1]} = v$. Hence, $\#_v(s) > \#_u(s)$. By assumption, there is a path from v to u in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ and $(u,v) \in E^{\text{im}}_{\blacktriangleright}$, thus u and v are on a common cycle in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$, Lemmas 7.3.4 and 7.4.1 imply $\#_u(s) = \#_v(s)$, a contradiction. ∎

Let u and v be as stated in Lemma 7.4.13. Note that if u and v satisfy the condition in the lemma, u and v will always "occur together" in a conforming string as if they were one activity. With such an observation, we can then pre-process a given schema by "collapsing" such nodes according to in Lemma 7.4.13. However, two nodes satisfying Lemma 7.4.13 does not necessarily mean they are "safe" to be collapsed. For example, if nodes u and v in some schema are eligible to be combined based on Lemma 7.4.13 and there is a node w in the same schema that has constraint $iRes(w,u)$. The collapsing of u and v implies that $iRes(w,v)$ is also a constraint that should be satisfied. According to Theorem 7.3.5, the schema is not satisfiable. Thus, in the following definition, we define when two nodes are "safe" to collapse (i.e., "collapsable").

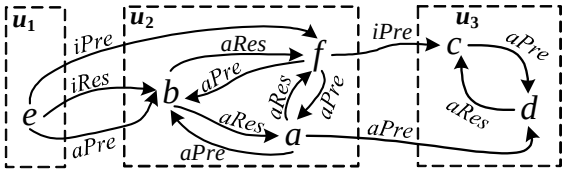

Figure 7.3: A collapsed schema example

Definition: Given an al$^+$schema $S = (A, C)$ that contains only alternating and immediate constraints, and its causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$, S is *collapsable* if S satisfies all of the following.

(1). $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright})$ and $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ are acyclic,

(2). for each $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (or $E^{\text{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq u$ and $(v, w) \in E^{\text{im}}_{\blacktriangleleft}$ (resp. $E^{\text{im}}_{\blacktriangleright}$),

(3). for each $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (or $E^{\text{im}}_{\blacktriangleleft}$), there does not exist $w \in A$ such that $w \neq v$ and $(u, w) \in E^{\text{im}}_{\blacktriangleright}$ (resp. $E^{\text{im}}_{\blacktriangleleft}$), and

(4). for each distinct $u, v, w \in A$, if $(u, w), (v, w) \in E^{\text{im}}_{\blacktriangleright}$ or $(u, w), (v, w) \in E^{\text{im}}_{\blacktriangleleft}$, then there is no path from w to either u or v in graph $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$.

Note that Conditions (1)–(3) in the above definition are similar to the characterization stated Theorem 7.3.9.

Example 7.4.14 Consider an al$^+$schema with 6 activities, a, b, c, d, e, f, and the constraints shown in Fig. 7.3 as $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ where the edge labels denote types of constraints. (Ignore the dashed boxes labeled $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3$ for now.) The schema is collapsable. However, if constraint $iPre(a, c)$ is added to the schema, Condition (4) (in the collapsability definition) is violated and thus the new schema is not collapsable, since $(f, c), (a, c) \in E^{\text{im}}_{\blacktriangleleft}$ and there is a path cda from c to a in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$. ∎

Lemma 7.4.15 Given an al$^+$schema S that contains only alternating and immediate constraints, S is conformable only if S is collapsable.

Proof: Let $S = (A, C)$ be as stated in the lemma. Suppose $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ is the causality graph of S. The goal is to show that if S is conformable, then Conditions (1)–(4) (in the definition of collapsability) all hold wrt S.

According to Lemma 7.3.2 and Lemma 7.3.4, Condition (1) hold. According to Lemma 7.3.5 and Lemma 7.3.4, Conditions (2) and (3) hold.

Now we focus on Condition (4). Suppose that Conditions (1)–(3) hold while Condition (4) fails. Assume that s is a conformable string of S. Then assume that there exist distinct $u, v, w \in A$, such that $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (or $E^{\text{im}}_{\blacktriangleleft}$), u and v are on the same cycle in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$, and (w, v) is in $E^{\text{im}}_{\blacktriangleright}$ (resp. $E^{\text{im}}_{\blacktriangleleft}$). To disprove the assumption, we consider the case when $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (where case $E^{\text{im}}_{\blacktriangleleft}$ is similar). According to Lemma 7.4.13, s also satisfies $iPre(v, u)$. Since s satisfies $iRes(w, v)$ according to the assumption, this violates Condition (2); thus, a contradiction. ∎

Definition: Given a collapsable schema $S = (A, C)$ that contains only alternating and immediate constraints, the *collapsed schema* of S is a schema (A', C') constructed as follows:

1. Initially $A' = A$ and $C' = C$.
2. Repeat the following steps while (A', C') is changed:
   i. Let $(A', E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ be the corresponding causality graph of (A', C').
   ii. for each $u, v \in A$ on a common cycle in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$, If $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ or $E^{\text{im}}_{\blacktriangleleft}$, then (1) remove each $X(u, v)$ or $X(v, u)$ from C', where X ranges over *aRes*, *aPre*, *iRes*, and *iPre*. (2) Create node w_{uv}; let $A' := A' - \{u, v\} \cup \{w_{uv}\}$, and (3) replace each u and v in C' by w_{uv}.

It is easy to show that given a collapsable al$^+$schema, the corresponding collapsed schema is unique.

The following lemma (Lemma 7.4.16) is easy to verify.

Lemma 7.4.16 Given a collapsable al$^+$schema S that only contains alternating and immediate constraints, and the collapsed schema S' of S, S is conformable iff S' is conformable.

By Corollary 7.4.3, Lemmas 7.4.15, and 7.4.16, conformance checking of a schema that only contains alternating and immediate constraints can be reduced to conformance checking of its collapsed schema. Thus, in the remainder of this subsection, we mainly focus on the collapsed schemas.

In order to have a clean statement of the necessary and sufficient condition, we introduce a concept of "gap-free". Essentially, "gap-free" is to deal with a special case of a schema that contains alternating and immediate constraints. the special case is illustrated in the following Example 7.4.17.

Example 7.4.17 Continue with Example 7.4.14; note that the schema in Fig. 7.3 is a collapsed schema. Consider a schema $S^{\boldsymbol{u}_2}$ that only contains activities a, b, and f, together with the constraints among them shown in Fig. 7.3 (i.e., a "subschema" bounded by the dashed box labeled as "$\boldsymbol{u}_2$"). Based on Theorem 7.4.4, $S^{\boldsymbol{u}_2}$ is conformable and a conforming string is baf. Now consider a schema $S^{\boldsymbol{u}_{1,2}}$ that only contains activities e, a, b, and f, together with the constraints among them shown in Fig. 7.3 (i.e., a "subschema" bounded by the dashed boxes labeled as "$\boldsymbol{u}_1$" and "$\boldsymbol{u}_2$" together with the constraints crossing $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$). Due to constraints $iRes(e, b)$ and $iPre(e, f)$, if $S^{\boldsymbol{u}_{1,2}}$ is conformable, then each conforming string of $S^{\boldsymbol{u}_{1,2}}$ must contain substring "feb". This requirement

leads to some restriction upon schema $S^{\boldsymbol{u}_2}$, i.e., if we take out activity "e" from $S^{\boldsymbol{u}_{1,2}}$ and focus on schema $S^{\boldsymbol{u}_2}$ again, one restriction would be: is there a conforming string of $S^{\boldsymbol{u}_2}$ that contains a substring fb? If the answer is negative, then apparently, $S^{\boldsymbol{u}_{1,2}}$ is not conformable, since no substring feb can be formed. ∎

With the concern shown in Example 7.4.17, we need a checking mechanism to decide if two activities can occur as a substring (i.e., "gap-free") in some conforming string. More specifically, given $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ as a causality graph of a collapsed schema S, we are more interested in checking if two activities that in the same strongly connected component in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$ can form a substring in a conforming string of S. Note that in Example 7.4.17, activities a, b, and f are in the same strongly connected component labeled with $\boldsymbol{u}_2$ in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$.

Definition: Let $S = (A, C)$ be a schema that only contains alternating constraints and $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ the causality graph of S, such that $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$ is strongly connected. Given two distinct activities $u, v \in A$, u, v are *gap-free* (wrt S) if for each $w, x, y \in A$, the following conditions should all hold wrt graph $(A, E^{\text{al}}_{\blacktriangleright})$:

(a). if there is a path p with length greater than 2 from u to v, the following all hold:

  (i). if w is on p, then $(u, v) \notin E^{\text{al}}_{\blacktriangleright}$,

  (ii). if there is a path from x to u, then $(x, v) \notin E^{\text{al}}_{\blacktriangleright}$,

  (iii). if there is a path from v to y, then $(u, y) \notin E^{\text{al}}_{\blacktriangleright}$,

  (iv). if there are paths from x to u and v to y, and then $(x, y) \notin E^{\text{al}}_{\blacktriangleright}$, and

(b). if there is a path from v to u, then the following all hold:

  (i). if there is a path from x to v, then $(x, u) \notin E^{\text{al}}_{\blacktriangleright}$,

  (ii). if there is a path from u to y, then $(v, y) \notin E^{\text{al}}_{\blacktriangleright}$, and

  (iii). if there are paths from x to v and u to y, then $(x, y) \notin E^{\text{al}}_{\blacktriangleright}$.

Let S be as stated in the above definition; in the following Lemma 7.4.18 we show that two activities in S can appear in a conforming string as substrings if and only if they are gap-free.

Lemma 7.4.18 Given a conformable al$^+$schema $S = (A, C)$ that only contains alternating constraints, the causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ of S, such that $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$ is strongly connected, and two activities $u, v \in A$, "uv" can appear as a substring in some a conforming string of S iff u, v are gap-free wrt S.

The proof of Lemma 7.4.18 is quite involved. The main idea of the proof is first to show that the a conforming string of a strongly connected component (wrt to a collection of alternating constraints) can only be of form that is a repetition of a cycle in the same strongly connected component. Then we show that with such form, only nodes that are gap-free can occur together as a substring.

In the following, we present the proof of Lemma 7.4.18. The proof relies on Corollary 7.4.19 and Lemma 7.4.21.

Based on the definition of al$^+$schemas, it is easy to establish the following.

Corollary 7.4.19 Given $S = (A, C)$ as a al$^+$schema that only contains alternating constraints, and the causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ of S, such that $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$ is strongly connected, $(u, v) \in E^{\text{al}}_{\blacktriangleright}$ implies $(v, u) \in E^{\text{al}}_{\blacktriangleleft}$, and vice versa.

Given a string s, and a set of activities A, the *projection* of s on A, denoted as $\pi_A(s)$, is a string obtained by removing each activity occurring in s but not in A.

Example 7.4.20 Let $s = ababcddeed$ be a string. Then $\pi_{\{a,b\}}(s) = abab$, $\pi_{\{a,d,e\}}(s) = aaddeed$, and $\pi_{\{d\}}(s) = ddd$. ∎

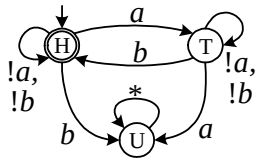

Figure 7.4: Automata

Given a string s, denote $(s)^*$ (or simply, s^* if the context is clear) to be a countably infinite set $\{\varepsilon, s, ss, sss, ...\}$, where ε is the empty string.

Lemma 7.4.21 Given an al$^+$schema $S = (A, C)$ that only contains alternating constraints, the causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft})$ of S, such that $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$ is strongly connected, T the set of all conforming strings of S, and n activities $a_1, a_2, ..., a_n \in A$, where $n > 1$, such that $(a_1, a_2), ..., (a_{n-1}, a_n) \in E^{\text{al}}_{\blacktriangleright}$ and $(a_1, a_n) \in E^{\text{al}}_{\blacktriangleright}$, then $\bigcup_{s \in T}\{\pi_{\{a_1,a_2,...,a_n\}}(s)\} \subseteq (a_1...a_n)^*$.

Proof: Let S, A, C, $E^{\text{al}}_{\blacktriangleright}$, $E^{\text{al}}_{\blacktriangleleft}$, and n be as stated in the lemma. As $(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n) \in E^{\text{al}}_{\blacktriangleright}$ and $(a_1, a_n) \in E^{\text{al}}_{\blacktriangleright}$, according to Corollary 7.4.19, we have $(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n) \in E^{\text{al}}_{\blacktriangleleft}$ and $(a_1, a_n) \in E^{\text{al}}_{\blacktriangleleft}$.

We consider an automaton shown in Fig. 7.4. The automaton has three states H, T, and U. the initial and the (only) accepting state are both H. The transitions are labeled on edges, where a and b are activities. "$!a, !b$" denotes the transition where an activity other than a or b occurs; and "$*$" denotes an arbitrary activity. It is easy to verify that the automaton shown in Fig. 7.4 can accept a string that satisfies $aRes(a, b)$ and $aPre(b, a)$, and reject a string that does not satisfy $aRes(a, b)$ or $aPre(b, a)$.

Let M_1, M_2, ..., M_n be n automata, where for each $i \in [1..(n-1)]$, M_i is obtained by replacing a and b in Fig. 7.4 by a_i and a_{i+1} respectively and M_n is obtained by replacing a and b in Fig. 7.4 by a_1 and a_n respectively. Then a conforming string of S should be

accepted by the cross product of M_1, M_2, ..., M_n.

With the above concern, we consider automata M_1, M_2, ..., M_n. Denote $\langle s_1, s_2, ..., s_n\rangle$ to be a vector of size n to record the states of M_1, M_2, ..., M_n respectively, where s_i ($i \in [1..n]$) ranges over H, T, and U. The initial state is $\langle H, H, ..., H\rangle$. Notice that given a string s over A, starting from the initial state, for each $i \in [1..n]$, if during the execution, $s_i = U$, then s will not be accepted by the cross product of M_1, M_2, ..., M_n (due to the fact that there is no outgoing transition from U and U is not an accepting state), and therefore not conformable. Thus, we only consider the transitions between states H and T for each M_i.

During the execution of M_1, M_2, ..., M_n upon a string s over A, if the next occurring activity is a_i, we have the following three cases:

(a). if $i = 1$, then the current state should be $\langle H, s_2, s_3, ..., s_{n-1}, H\rangle$, where $s_j \in \{H, T\}$ ($j \in [2..(n-1)]$); otherwise if s_1 (or s_n) is T, then in the next state s_1 (or s_n) will be in state U. And apparently after consuming a_i, the state becomes $\langle T, s_2, s_3, ..., s_{n-1}, T\rangle$.

(b). if $i = n$: then the current state should be $\langle s_1, s_2, ..., s_{n-2}, T, T\rangle$, where $s_j \in \{H, T\}$ ($j \in [1..(n-3)]$); otherwise if s_{n-1} (or s_n) is T, then in the next state s_{n-1} (or s_n) will be in state U. And apparently after consuming a_i, the state becomes $\langle s_1, s_2, ..., s_{n-2}, H, H\rangle$.

(c). if $i \in [2..(n-1)]$: then the current state should be $\langle s_1, s_2, ..., s_{i-2}, T, H, s_{i+1}...s_n\rangle$, where $s_j \in \{H, T\}$ ($j \in [1..(i-2)] \cup [(i+1)..n]$); otherwise if s_{i-1} (or s_i) is T, then in the next state s_{i-1} (or s_i) will be in state U. And apparently after consuming a_i, the state becomes $\langle s_1, s_2, ..., s_{i-2}, H, T, s_{i+1}...s_n\rangle$.

Without loss of generality, we fix s to be a string over $\bigcup_{i=1}^{n}\{a_i\}$, since each activity in $A - \bigcup_{i=1}^{n}\{a_i\}$ will not make the states of M_1, M_2, ..., M_n change. Starting from the initial

state $\langle H, H, ..., H\rangle$, the first activity of s can only be a_1 (case (a) above); otherwise, s will not be satisfied. And after consuming a_1, the state becomes $\langle T, H, H, ..., H, T\rangle$. Then the next activity of s can only be a_2 (case (c) above), and state becomes $\langle H, T, H, H, ..., H, T\rangle$. Similarly, the next activity can only be a_3 (case (c) above), the new state is $\langle H, H, T, H, H, ..., H, T\rangle$. By induction, it is easy to show that the i^{th} activity can only be a_i ($i \in [2..(n-1)]$). Thus, after consuming a_{n-1}, the state becomes $\langle H, ..., H, H, T, T\rangle$. And the next activity can only be a_n (case (b) above); the new state is $\langle H, H, ..., H\rangle$, which is the initial state. Therefore, s can only be a string that is a repetition of string $a_1a_2...a_n$. ∎

We now present a proof of Lemma 7.4.18 that shows the main property of "gap-free".

Proof: (Lemma 7.4.18) Let S, A, C, $E^{\text{al}}_{\blacktriangleright}$, $E^{\text{al}}_{\blacktriangleleft}$, u, and v be as stated in the lemma.
($\Rightarrow$) Denote T to be the set of all conforming strings of S. Without loss of generality, in the following proof, if two activities have different names, then by default we mean that they are distinct. For example, if x and y are two activities, then x and y are two distinct activities. The goal is to prove that if some string in T contains uv as a substring, then u, v are gap-free wrt S.

Consider Condition (a) (in the definition of gap-free), i.e., there is a path with length greater than 2 from u to v. Consider Condition (a)-(ii) (the analysis of (a)-(i), (a)-(iii), and (a)-(iv) are similar), i.e., if there is a path from $x \in A$ to u, then there is no edge from x to v. Suppose this is not the case, i.e., $(x, v) \in E^{\text{al}}_{\blacktriangleright}$. According to to Lemma 7.4.21, $\bigcup_{s\in T}\{\pi_{\{x,u,w,v\}}(s)\} = \{(xuwv)^*\}$, which implies that uv cannot appear as a substring in each conforming string of S; a contradiction.

Consider Condition (b), i.e., there is a path from v to u. Consider Condition (b)-(i) (the analysis of (b)-(ii) and (b)-(iii) are similar), i.e., if there is a path from $x \in A$ to v, then there is no edge from x to u. Suppose this is not the case, i.e., $(x, u) \in E^{\text{al}}_{\blacktriangleright}$. According to to Lemma 7.4.21, $\bigcup_{s\in T}\{\pi_{\{x,u,v\}}(s)\} = \{(xvu)^*\}$, which implies that uv

cannot appear as a substring in each conforming string of S; a contradiction.

As a summary, uv can appear as a substring in a conforming string of S only if u, v are gap-free.

($\Leftarrow$) The reasoning can be divided into three cases: (I) there is no path from u to v or v to u in $(A, E^{\text{al}}_{\blacktriangleright})$, (II) there is a path from u to v in $(A, E^{\text{al}}_{\blacktriangleright})$, and (III) there is a path from v to u in $(A, E^{\text{al}}_{\blacktriangleright})$.

Suppose case (I) holds. Let $A_1 \subseteq A - \{u, v\}$ be a set, where an activity $x \in A$ is in A_1 iff there is a path from x to u or v in $(A, E^{\text{al}}_{\blacktriangleright})$; let $A_3 \subseteq A - \{u, v\}$ be a set, where an activity $x \in A$ is in A_3 iff there is a path from u or v to x in $(A, E^{\text{al}}_{\blacktriangleright})$; and let A_2 be a set that is defined as $A - (A_1 \cup A_2 \cup \{u, v\})$. Apparently, A_1, A_2, and A_3 are pairwise disjoint. Moreover, it is easy to verify that for each $x \in A_2$ and each $y \in A_1$, there is no path from x to y in $(A, E^{\text{al}}_{\blacktriangleright})$; otherwise, x will be in A_1. Similarly, let sets $B_1 = A_1$, $B_2 = \{u\}$, $B_3 = \{v\}$, $B_4 = A_2$, and $B_5 = A_3$; it is easy to verify that for each $i, j \in [1..5]$, if $i < j$, then for each $x \in B_i$ and each $y \in B_j$, there is no path from x to y. Let $\bar{s}_1$, $\bar{s}_2$, and $\bar{s}_3$ be sequences over A_1, A_2, and A_3 respectively, such that $\bar{s}_1$, $\bar{s}_2$, and $\bar{s}_3$ are subsequences of some topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. We have $\bar{s}_1 u v \bar{s}_2 \bar{s}_3$ as a topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. According to Corollary 7.4.19, $\bar{s}_1 u v \bar{s}_2 \bar{s}_3$ is a conforming string (which contains uv as a substring).

Suppose case (II) holds. Let A_1, A_2, A_3, $\bar{s}_1$, $\bar{s}_2$, and $\bar{s}_3$ be as stated in case (I). Let $A_4 \subseteq A - \{u, v\}$ be a set, where an activity $x \in A$ is in A_4 iff there is a path from x to u in $(A, E^{\text{al}}_{\blacktriangleright})$; let $A_6 \subseteq A - \{u, v\}$ be a set, where an activity x is in A_6 iff there is a path from v to x in $(A, E^{\text{al}}_{\blacktriangleright})$; and let A_5 be a set that is defined as $A - (A_4 \cup A_6 \cup \{u, v\})$. Since there is a path from u to v in $(A, E^{\text{al}}_{\blacktriangleright})$, there is no path from v to u in $(A, E^{\text{al}}_{\blacktriangleright})$, which implies that $A_4 \cap A_6 = \varnothing$. Thus, A_4, A_5, and A_6 are pairwise disjoint. Let $\bar{s}_4$, $\bar{s}_5$, and $\bar{s}_6$ be sequences over A_4, A_5, and A_6 respectively, such that $\bar{s}_4$, $\bar{s}_5$, and $\bar{s}_6$ are subsequences of some topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. It is easy to verify that $\bar{s}_4 u \bar{s}_5 v \bar{s}_6$ is a

topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. We argue that either string $s = \bar{s}_4 u \bar{s}_5 \bar{s}_4 u v \bar{s}_6 \bar{s}_5 v \bar{s}_6$ or string $t = \bar{s}_1 u v \bar{s}_2 \bar{s}_3$ conforms S.

Suppose this is not the case, i.e., there exists $aRes(x, y) \in C$, such that neither s nor t satisfies $aRes(x, y)$ (for constraint $aPre(x, y)$, the reasoning the similar). Let sets $B_1 = A_4$, $B_2 = \{u\}$, $B_3 = A_5$, $B_4 = \{v\}$, and $B_5 = A_6$. The reasoning can be divided into 15 cases: i.e., considering the combination of (x, y) in one of the 15 elements in set $\bigcup_{1 \leqslant i \leqslant j \leqslant 5} \{B_i \times B_j\}$. Note that it is impossible to have $(x, y) \in B_i \times B_j$, where $i > j$, because $\bar{s}_4 u \bar{s}_5 v \bar{s}_6$ is a topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. Moreover, we divide the 15 cases into three main categories: (1) (including 5 cases) $(x, y) \in B_i \times B_i$ ($i \in [1..5]$), (2) (including 6 cases) $(x, y) \in C$, where C ranges over $\{A_4 \times A_5, A_4 \times \{u\}, \{u\} \times A_5, A_5 \times \{v\}, A_5 \times A_6, \{v\} \times A_6\}$, and (3) (including 4 cases) $(x, y) \in C$, where C ranges over $\{\{u\} \times \{v\}, A_4 \times \{v\}, \{u\} \times A_6, A_4 \times A_6\}$.

Category (1): It is impossible to have $(x, y) \in \{(u, u)\}$ or $\{(v, v)\}$; otherwise $(A, E^{\text{al}}_{\blacktriangleright})$ will be cyclic, which contradicts Condition (1) in Theorem 7.4.23. Thus we consider case where $(x, y) \in A_i \times A_i$ ($i \in \{4, 5, 6\}$). Since for each $i \in \{4, 5, 6\}$, $\bar{s}_i$ complies with the topological of $(A, E^{\text{al}}_{\blacktriangleright})$, $\bar{s}_i$ satisfies $aRes(x, y)$. Moreover, as B_k and B_j ($k, j \in [1..5]$) are pairwise disjoint, x or y will not occur in $\bigcup_{j=1}^{5} B_j - A_i$. Therefore, $s = \bar{s}_4 u \bar{s}_5 \bar{s}_4 u v \bar{s}_6 \bar{s}_5 v \bar{s}_6$ satisfies $aRes(x, y)$, a contradiction.

Category (2): Note that $\bar{s}_4$ and $\bar{s}_5$ are "alternating" in s, i.e., s contains subsequence $\bar{s}_4 \bar{s}_5 \bar{s}_4 \bar{s}_5$. Thus for each $(w, z) \in A_4 \times A_5$, s satisfies $aRes(w, z)$. Therefore, if $(x, y) \in A_4 \times A_5$, s satisfies $aRes(x, y)$. Similarly, it is easy to verify that for each $(w, z) \in A_4 \times \{u\}$, $\{u\} \times A_5$, $A_5 \times \{v\}$, $A_5 \times A_6$, or $\{v\} \times A_6$, s satisfies $aRes(w, z)$, which leads to a contradiction.

Category (3): Suppose that $(x, y) \in A_4 \times \{v\}$. Recall that for each $z \in A_4$, there is a path from z to u in $(A, E^{\text{al}}_{\blacktriangleright})$; further, according to the assumption of case (II), there is a path from u to v in $(A, E^{\text{al}}_{\blacktriangleright})$; thus, there is a path from x to $y = v$ in $(A, E^{\text{al}}_{\blacktriangleright})$.

According to Condition (b)-(ii) in the definition of gap-free, $(x, y) \in E^{\text{al}}_{\blacktriangleright}$, which leads to a contradiction. Similarly, it is easy to verify that if $(x, y) \in \{u\} \times A_6$ or $A_4 \times A_6$, $(x, y) \notin E^{\text{al}}_{\blacktriangleright}$. Now consider when $(x, y) = (u, v)$, i.e., $(u, v) \in E^{\text{al}}_{\blacktriangleright}$. According to Condition (b)-(i) in the definition of gap-free, for each path p from u to v, no node in $A - \{u, v\}$ is on p, i.e., edge (u, v) is the only path from u to v. Therefore, similar to case (I), it is easy to verify that $t = \bar{s}_1 u v \bar{s}_2 \bar{s}_3$ is a topological order of $(A, E^{\text{al}}_{\blacktriangleright})$, which satisfies $aRes(u, v)$ according to Corollary 7.4.19, a contradiction.

Suppose case (III) holds. Let $A_1 \subseteq A - \{u, v\}$ be a set, where an activity $x \in A$ is in A_1 iff there is a path from x to v in $(A, E^{\text{al}}_{\blacktriangleright})$; let $A_3 \subseteq A - \{u, v\}$ be a set, where an activity x is in A_3 iff there is a path from u to x in $(A, E^{\text{al}}_{\blacktriangleright})$; and let A_2 be a set that is defined as $A - (A_1 \cup A_2 \cup \{u, v\})$. Let $\bar{s}_1$, $\bar{s}_2$, and $\bar{s}_3$ be sequences over A_1, A_2, and A_3 respectively, such that $\bar{s}_1$, $\bar{s}_2$, and $\bar{s}_3$ are subsequences of some topological order of $(A, E^{\text{al}}_{\blacktriangleright})$. Similar to the analysis of case (II), it is easy to verify that $\bar{s}_1 v \bar{s}_2 \bar{s}_1 u v \bar{s}_3 \bar{s}_2 u \bar{s}_3$ is a conforming string of S. ∎

The above Lemma 7.4.18 provides a sufficient and necessary condition to decide if two activities can appear "together" in some comforming strings given a set of constraints.

Given a graph $G = (V, E)$, for each $v \in V$, denote $\text{SV}(v)$ to be the set of all the nodes in the strongly connected component of G that contains v.

Let (A, C) be a collapsed schema and $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ its causality graph. Consider graph $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft} \cup E^{\text{im}}_{\blacktriangleright} \cup E^{\text{im}}_{\blacktriangleleft})$; given an activity $a \in A$, denote $S(a)$ to be a schema defined as $(\text{SV}(a), \{aRes(u, v) \mid (u, v) \in E^{\text{al}}_{\blacktriangleright} \wedge \text{SV}(a) = \text{SV}(u) = \text{SV}(v)\} \cup \{aPre(u, v) \mid (u, v) \in E^{\text{al}}_{\blacktriangleleft} \wedge \text{SV}(a) = \text{SV}(u) = \text{SV}(v)\})$.

Example 7.4.22 Continue with Example 7.4.14; consider the schema in Fig. 7.3. Note that the schema is a collapsed schema. $\text{SV}(a) = \text{SV}(b) = \text{SV}(f)$ is the strongly connected

component of the graph in Fig. 7.3 with nodes a, b, and f. Moreover, $S(a) = S(b) = S(f)$ is a schema that only contains activities a, b, and f, together with the constraints among them in Fig. 7.3. ∎

The following Theorem 7.4.23 provides a necessary and sufficient condition for conformability of schema with only alternating and immediate constraints.

Theorem 7.4.23 Given a schema S that only contains alternating and immediate constraints, S is conformable iff the following conditions all hold.

(1). S is collapsable,

(2). $\pi_{\text{al}}(\tilde{S})$ is conformable (recall that π_{al} denotes the "projection" only upon alternating constraints), where $\tilde{S}$ is the collapsed schema of S, and

(3). Let $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ be the causality graph of the collapsed schema $\tilde{S}$, for each $u, v, w \in A$, if there is a path from u to w in $(A, E^{\text{im}}_{\blacktriangleright})$, there is a path from u to v in $(A, E^{\text{im}}_{\blacktriangleleft})$, and $\text{SV}(w) = \text{SV}(v)$ wrt $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$, then either (1) v, w are gap-free wrt $S(v)$ if $v \neq w$, or (2) v has no outgoing edge in graph $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ if $v = w$.

Example 7.4.24 Continue with Example 7.4.22; Consider the schema in Fig. 7.3. This schema satisfies the conditions in Theorem 7.4.23 and is conformable. A conforming string can be $bdacfebdacf$. ∎

A proof of the "only if" direction of Theorem 7.4.23 is shown below. The detailed proof of the "if" direction is very complicated; thus examples and a proof are shown after the proof of the "only if" direction.

Proof: ($\Rightarrow$) Let $\tilde{S}$, u, v, w, A, $E^{\text{al}}_{\blacktriangleright}$, $E^{\text{al}}_{\blacktriangleleft}$, $E^{\text{im}}_{\blacktriangleright}$, and $E^{\text{im}}_{\blacktriangleleft}$ be as stated in Theorem 7.4.23. Condition (1) follows from Lemmas 7.4.15 and 7.4.16. Condition (2) follows from Lemma 7.4.16. Let s be a conforming string of S. For Condition (3), since u occurs at least once in s, s must contain substring $va_1a_2...a_mub_1b_2...b_nw$, where $(v,a_1),(a_1,a_2),...,(a_m,u) \in E^{\text{al}}_{\blacktriangleleft}$ and $(u,b_1),(b_1,b_2),...,(b_{n-1},b_n),(b_n,w) \in E^{\text{al}}_{\blacktriangleright}$. Now we consider two cases: (1) $v = w$ and (2) $v \neq w$.

(1) ($v = w$) Suppose v has an outgoing edge in $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ to a node, say $x \in A$. If $x \notin \{a_1,...,a_m,u,b_1,...,b_n\}$, then s cannot satisfy constraint derived from edge (v,x), as substring $va_1a_2...a_mub_1b_2...b_nv$ will violate it. If $x \in \{a_1,...,a_m,u,b_1,...,b_n\}$, then there is a cycle from v to x and x to v, where either a_1 or b_n is on the cycle. According to Lemma 7.4.13, since $iPre(a_1,x)$, $iRes(b_n,x) \in C$, x can be collapsed (with either a_1 or b_n). Therefore, $\tilde{S}$ is not a collapsed schema, a contradiction.

(2) ($v \neq w$) Since $\tilde{S}$ is a collapsed schema, for each $x \in \{a_1,...,a_m,u,b_1,...,b_n\}$, $x \notin \text{SV}(v)$ (or $\text{SV}(w)$); otherwise x can be "collapsed". Therefore, $\pi_{\text{SV}(v)}(s)$ contains substring vw. As s is a conforming string of S, $\pi_{\text{SV}(v)}(s)$ satisfies every constraint "related" to $\text{SV}(v)$, i.e., every constraint in schema $S(v)$; we have $\pi_{\text{SV}(v)}(s)$ is a conforming string of $\text{SV}(v)$. Based on Lemma 7.4.18, u, w are gap-free wrt $S(v)$. ∎

In the remainder of this subsection, we only focus on the proof of the "if" direction of Theorem 7.4.23. The main idea of the proof of the "if" direction is done by providing a procedure that constructs a conforming string (wrt the input schema).

Based on Lemma 7.4.16, in this subsection, we only focus on the collapsed schemas.

Given a collapsed schema $S = (A, C)$ that satisfies all conditions stated in Theorem 7.4.23, we construct a string s with input S, such that s conforms to S. The construction process is divided into two main steps: (1) create a "grouped graph" for S, and (2) introduce a procedure of constructing strings, so that the produced string can satisfies

each constraint in C.

1. Grouped graphs

Let $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ be a causality graph of a collapsed schema that satisfies all conditions in Theorem 7.4.23. If we "group" each strongly connected component in $G = (A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$ into a single node, the *grouped* graph will become acyclic and each grouped node contains only alternating constraints. Since the conformability checking for schemas containing only alternating constraints have been resolved in Lemma 7.4.4, we can apply a divide-and-conquer approach by first constructing conforming strings for each grouped node (i.e., each strongly connected component in G) and then assemble them together for the entire graph.

Definition: Let S be a collapsed schema and $G = (A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ be the causality graph of S. The *grouped graph* of G is a triple (V, E, μ), where V is a finite set of *grouped activities*, $E \subset V \times V$ is the edge set, and μ is a total mapping from $A \cup E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft}$ to $V \cup E$ that is constructed according to the following procedures.

1. Initially $V = E = \varnothing$ and the domain of μ is empty.
2. For each node $u \in A$, if there exists another node $v \in A$, such that v is in the domain of μ, and u, v are in a common cycle in $(A, E^{\text{al}}_{\blacktriangleright} \cup E^{\text{al}}_{\blacktriangleleft})$, then let $\mu(u)$ be $\mu(v)$; otherwise, create a new node $\boldsymbol{w}$ in V and let $\mu(u)$ be $\boldsymbol{w}$.
3. For each edge $(u, v) \in E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft}$, if $\mu(u) = \mu(v)$, then $\mu((u, v)) = \mu(u)$; otherwise (i.e., $\mu(u) \neq \mu(v)$), if there is no edge $(\mu(u), \mu(v))$ in E, then create a new edge $(\mu(u), \mu(v))$ in E; let $\mu(u, v)$ be $(\mu(u), \mu(v))$.

Example 7.4.25 Continue with Example 7.4.24; Consider the schema in Fig. 7.3. Note that the schema is a collapsed schema. There are two cycles in graph $(A, E^{\text{al}\cup\text{im}}_{\blacktriangleright} \cup E^{\text{al}\cup\text{im}}_{\blacktriangleleft})$:

abf and cd. Thus, for the corresponding grouped graph (V, E, μ) we create three nodes $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, and $\boldsymbol{u}_3$ in V and let $\mu(e) = \boldsymbol{u}_1$, $\mu(a) = \mu(b) = \mu(f) = \boldsymbol{u}_2$, and $\mu(c) = \mu(d) = \boldsymbol{u}_3$ (shown as the dashed boxes). Further, the mapping of the edges are $\mu(e, b) = \mu(e, f) = (\boldsymbol{u}_1, \boldsymbol{u}_2)$, $\mu(b, f) = \mu(f, a) = \mu(a, b) = \boldsymbol{u}_2$, $\mu(f, c) = \mu(a, d) = (\boldsymbol{u}_2, \boldsymbol{u}_3)$, and $\mu(c, d) = \mu(d, c) = \boldsymbol{u}_3$. ∎

We note that a grouped graph is always acyclic.

Given a causality graph $(A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ and its corresponding grouped graph (V, E, μ), denote μ^{-1} to be a total mapping from $V \cup E$ to $2^{A \cup E^{\text{al} \cup \text{im}}_{\blacktriangleright} \cup E^{\text{al} \cup \text{im}}_{\blacktriangleleft}}$, such that for each $x \in V \cup E$, $y \in A \cup E^{\text{al} \cup \text{im}}_{\blacktriangleright} \cup E^{\text{al} \cup \text{im}}_{\blacktriangleleft}$ is in $\mu^{-1}(x)$ iff $\mu(y) = x$. Moreover, for each $x \in V \cup E$, denote $\mu^{-1}_V(x)$ as $\mu^{-1}(x) \cap A$ and $\mu^{-1}_E(x)$ as $\mu^{-1}(x) \cap (E^{\text{al} \cup \text{im}}_{\blacktriangleright} \cup E^{\text{al} \cup \text{im}}_{\blacktriangleleft})$.

Example 7.4.26 Continuing with Example 7.4.25, we have $\mu^{-1}(\boldsymbol{u}_1) = \{e\}$, $\mu^{-1}((\boldsymbol{u}_1, \boldsymbol{u}_2)) = \{(e, f), (e, b)\}$, and $\mu^{-1}(\boldsymbol{u}_3) = \{c, d, (c, d), (d, c)\}$. Moreover, $\mu^{-1}_V(\boldsymbol{u}_3) = \{c, d\}$ and $\mu^{-1}_E(\boldsymbol{u}_3) = \{(c, d), (d, c)\}$. ∎

Given a collapsed schema $S = (A, C)$ that satisfies every condition in Theorem 7.4.23, the causality graph $\mathcal{G}_S = (A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ of S, and the grouped graph (V, E, μ) of $\mathcal{G}_S$. We divide C into three different types of constraints:

1. immediate constraints (denoted as C_{im}): a set of all the immediate constraints in C (e.g., $iPre(f, c)$, $iRes(e, b)$ in Example 7.4.14),
2. internal alternating constraints (denoted as C^{I}_{al}):

$$\{aRes(a, b) \mid (a, b) \in E^{\text{al}}_{\blacktriangleright} \wedge \mu(a) = \mu(b)\} \cup \{aPre(a, b) \mid (a, b) \in E^{\text{al}}_{\blacktriangleleft} \wedge \mu(a) = \mu(b)\}$$

(e.g., $aRes(b, f)$, $aPre(c, d)$ in Example 7.4.14), and

3. external alternating constraints (denoted as $C^{\mathrm{X}}_{\mathrm{al}}$):

$$\{aRes(a,b) \mid (a,b) \in E^{\mathrm{al}}_{\blacktriangleright} \wedge \mu(a) \neq \mu(b)\} \cup \{aPre(a,b) \mid (a,b) \in E^{\mathrm{al}}_{\blacktriangleleft} \wedge \mu(a) \neq \mu(b)\}$$

(e.g., $aPre(a,d)$ and $aPre(e,b)$ in Example 7.4.14).

In the following, we will handle each kind of constraints.

2. Constructing Strings

The main idea to prove the "if" direction of Theorem 7.4.23 is to construct a string for each grouped node, such that the string satisfies each internal alternating constraint, each "related" immediate constraint, and each "related" external alternating constraint. Then we "assemble" these strings together and prove that the assembled string conforms the given schema.

Given a collapsed schema $S = (A, C)$ that satisfies every condition in Theorem 7.4.23 and a set $B \subseteq A$, denote $C_{\mathrm{im}}(B)$, $C^{\mathrm{I}}_{\mathrm{al}}(B)$, and $C^{\mathrm{X}}_{\mathrm{al}}(B)$ to be the following sets:

$$\begin{aligned}
C_{\mathrm{im}}(B) = \; & \{iRes(a,b) \mid iRes(a,b) \in C_{\mathrm{im}} \wedge a \in B\} \cup \\
& \{iPre(a,b) \mid iPre(a,b) \in C_{\mathrm{im}} \wedge a \in B\} \\
C^{\mathrm{I}}_{\mathrm{al}}(B) = \; & \{aRes(a,b) \mid aRes(a,b) \in C^{\mathrm{I}}_{\mathrm{al}} \wedge a \in B\} \cup \\
& \{aPre(a,b) \mid aPre(a,b) \in C^{\mathrm{I}}_{\mathrm{al}} \wedge a \in B\} \\
C^{\mathrm{X}}_{\mathrm{al}}(B) = \; & \{aRes(a,b) \mid aRes(a,b) \in C^{\mathrm{X}}_{\mathrm{al}} \wedge a \in B\} \cup \\
& \{aPre(a,b) \mid aPre(a,b) \in C^{\mathrm{X}}_{\mathrm{al}} \wedge a \in B\}
\end{aligned}$$

We call that $C_{\mathrm{im}}(B)$, $C^{\mathrm{I}}_{\mathrm{al}}(B)$, or $C^{\mathrm{X}}_{\mathrm{al}}(B)$ are immediate, internal alternating, or external alternating constraints (resp.) that are *related* to B.

In the remainder of this section, we will introduce several data structures, an algorithm, and several lemmas/corollaries to show how to construct a string that can satisfy

all the (related) immediate, internal alternating, and external alternating constraints.

We first introduce two data structures to handle the internal alternating constraints.

Let $S = (A, C)$ be a collapsed schema that satisfies the conditions in Theorem 7.4.23, $\mathcal{G}_S = (A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ the causality graph of S, and (V, E, μ) grouped graph of $\mathcal{G}_S$. Based on Condition (2) of Theorem 7.4.23, graph $(A, E^{\text{al}}_{\blacktriangleright})$ is acyclic. Then, for each $\boldsymbol{u} \in V$, we create the following two data structures:

(1) denote $\bar{\text{S}}_{\text{al}}(\boldsymbol{u})$ as a topological order of graph $(\mu_V^{-1}(\boldsymbol{u}), \mu_E^{-1}(\boldsymbol{u}) \cap E^{\text{al}}_{\blacktriangleright})$. It is easy to prove that $\bar{\text{S}}_{\text{al}}(\boldsymbol{u})$ satisfies each constraint in C^{I}_{al};

(2) further, let $\text{S}^{v,w}_{\text{al}}(\boldsymbol{u})$ be a set of strings, where $v, w \in \mu_V^{-1}(\boldsymbol{u})$, such that each string in $\text{S}^{v,w}_{\text{al}}(\boldsymbol{u})$ (i) is a conforming string of schema $S(v)$, (ii) only contains activities in $\mu_V^{-1}(\boldsymbol{u})$, and (iii) contains vw as a substring. Note that $\text{S}^{v,w}_{\text{al}}(\boldsymbol{u})$ is empty if and only if v, w are not gap-free wrt $S(v)$ according to Lemma 7.4.18.

Example 7.4.27 Continue with Example 7.4.26; $\bar{\text{S}}_{\text{al}}(\boldsymbol{u}_1) = e$, $\bar{\text{S}}_{\text{al}}(\boldsymbol{u}_2) = baf$, and $\bar{\text{S}}_{\text{al}}(\boldsymbol{u}_3) = dc$. $\text{S}^{d,c}_{\text{al}}(\boldsymbol{u}_3) = \{dc, dcdc, ...\}$ $\text{S}^{f,b}_{\text{al}}(\boldsymbol{u}_2) = \{bafbaf, bafbafbaf, ...\}$. ∎

As introduced above, the two data structures are to handle the each internal alternating constraints in the given (collapsed) schema. In the following, we focus on the construction of strings that satisfy immediate and external alternating constraints while still keeping the satisfiability of internal alternating constraints.

The following lemma serves as a basis for constructing strings (to be introduced in Alg. 7 later in this section).

Lemma 7.4.28 Given $S = (A, C)$ as a collapsed schema, $\mathcal{G}_S = (A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ the causality graph of S, (V, E, μ) grouped graph of $\mathcal{G}_S$, and an activity $a \in A$, for each

$b \in \mathrm{s}_{\mathrm{im}}(a)$, there exists at most one activity $c \in \mathrm{s}_{\mathrm{im}}(a)$, such that $b \neq c$ and $\mu(b) = \mu(c)$; and if b exists, then there is a path from a to b in $(A, E^{\mathrm{im}}_{\blacktriangleright})$ (or $(A, E^{\mathrm{im}}_{\blacktriangleleft})$) and there is a path from a to c in $(A, E^{\mathrm{im}}_{\blacktriangleleft})$ (resp. $(A, E^{\mathrm{im}}_{\blacktriangleright})$).

Proof: Let S, A, C, $E^{\mathrm{im}}_{\blacktriangleright}$, $E^{\mathrm{im}}_{\blacktriangleleft}$, V, E, μ, a, b, and c be as stated in the lemma.

Suppose there exists $d \in \mathrm{s}_{\mathrm{im}}(a)$, such that d is distinct from b and c, and $\mu(b) = \mu(c) = \mu(d)$. Then there must exist two nodes x, y in $\{b, c, d\}$, such that there is a path from x to y in $(A, E^{\mathrm{im}}_{\blacktriangleright})$ or $(A, E^{\mathrm{im}}_{\blacktriangleleft})$. Since the two cases are symmetric, we only analyze the case $(A, E^{\mathrm{im}}_{\blacktriangleright})$. As $\mu(x) = \mu(y)$, x and y must be on the same cycle in $(A, E^{\mathrm{al}}_{\blacktriangleright} \cup E^{\mathrm{al}}_{\blacktriangleleft})$. Thus, x and y can be collapsed, which contradicts the fact that S is a collapsed schema.

Suppose c exists and $\mu(c) = \mu(b)$. Similar to the above analysis, it is easy to have that there is a path from a to b in $(A, E^{\mathrm{im}}_{\blacktriangleright})$ (or $(A, E^{\mathrm{im}}_{\blacktriangleleft})$) and there is a path from a to c in $(A, E^{\mathrm{im}}_{\blacktriangleleft})$ (resp. $(A, E^{\mathrm{im}}_{\blacktriangleright})$). ∎

In the remainder of this section, we may use some "dummy activities" for simple explanation of the algorithms. Intuitively, a dummy activity is an activity that serves as a place holder for a chunk of activities or dummy activities.

Let S be a collapsed schema that satisfies the conditions in Theorem 7.4.23, $\mathcal{G}_S = (A, E^{\mathrm{al}}_{\blacktriangleright}, E^{\mathrm{al}}_{\blacktriangleleft}, E^{\mathrm{im}}_{\blacktriangleright}, E^{\mathrm{im}}_{\blacktriangleleft})$ the causality graph of S, and (V, E, μ) grouped graph of $\mathcal{G}_S$. For each activity $a \in A$, we construct a string $\hat{\mathrm{s}}(a)$ according to Alg. 7 below. Unlike Alg. 5 or 6 discussed earlier in this chapter, the returned string of Alg. 7 will not conform the input schema. Rather, it will satisfy some useful properties; and based on these properties, we are able to construct a conforming string, which will be discussed at the end of this subsection.

Note that Steps B and C follow from Lemma 7.4.28.

Example 7.4.29 Continue with Example 7.4.27; we execute Algorithm 7 for node e

Algorithm 7

Input: (1) The causality graph $\mathcal{G} = (A, E^{\text{al}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleleft})$ of a schema satisfying conditions of Theorem 7.4.23, (2) the grouped graph (V, E, μ) of $\mathcal{G}$, and (3) a node $a \in A$

Output: A string

A. Denote $\text{S}^{\mu}_{\text{im}}(a) \subseteq V$ as a set wrt a and μ, such that $\boldsymbol{u} \in V$ is in $\text{S}^{\mu}_{\text{im}}(a)$ iff there exists $b \in \text{S}_{\text{im}}(a) - \{a\}$ and $\mu(b) = \boldsymbol{u}$.

B. For each $\boldsymbol{u} \in \text{S}^{\mu}_{\text{im}}(a)$, if there exists exactly one node $b \in \text{S}_{\text{im}}(a)$, such that $\mu(b) = \boldsymbol{u}$, then create two strings $s^{\boldsymbol{u}}_{\blacktriangleleft}$ and $s^{\boldsymbol{u}}_{\blacktriangleright}$, such that $s^{\boldsymbol{u}}_{\blacktriangleleft} b s^{\boldsymbol{u}}_{\blacktriangleright} = \bar{\text{S}}_{\text{al}}(\boldsymbol{u})$.

C. For each $\boldsymbol{u} \in \text{S}^{\mu}_{\text{im}}(a)$, if there exist two distinct nodes $b, c \in \text{S}_{\text{im}}(a)$, such that $\mu(b) = \mu(c) = \boldsymbol{u}$ and there is a path from a to b (or c) in $(A, E^{\text{im}}_{\blacktriangleleft})$ (resp. $(A, E^{\text{im}}_{\blacktriangleright})$), then arbitrarily pick a string s in $\text{S}^{b,c}_{\text{al}}(\boldsymbol{u})$ and create two strings $s^{\boldsymbol{u}}_{\blacktriangleleft}$ and $s^{\boldsymbol{u}}_{\blacktriangleright}$, such that $s^{\boldsymbol{u}}_{\blacktriangleleft} bcs^{\boldsymbol{u}}_{\blacktriangleright} = s$.

D. Let $n = |\text{S}^{\mu}_{\text{im}}(a)|$. Suppose $\{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n\} = \text{S}^{\mu}_{\text{im}}(a)$ and $\boldsymbol{u}_1 ... \boldsymbol{u}_n$ is a subsequence of the topological order of (V, E).

E. Create a dummy activity $\hat{a}$ for a.

F. Let $\hat{\text{S}}(a)$ be $s^{\boldsymbol{u}_1}_{\blacktriangleleft} s^{\boldsymbol{u}_2}_{\blacktriangleleft} ... s^{\boldsymbol{u}_n}_{\blacktriangleleft} \hat{a} s^{\boldsymbol{u}_n}_{\blacktriangleright} ... s^{\boldsymbol{u}_1}_{\blacktriangleright}$.

G. For each $b \in A$, if $aRes(a, b) \in C^{\text{X}}_{\text{al}}$ (or $aPre(a, b) \in C^{\text{X}}_{\text{al}}$), then let $\hat{\text{S}}(a)$ be $\hat{\text{S}}(a)\bar{\text{S}}_{\text{al}}(\mu(b))$ (resp. $\bar{\text{S}}_{\text{al}}(\mu(b))\hat{\text{S}}(a)$).

H. Return $\hat{\text{S}}(a)$.

---

given the causality graph and the grouped graph in Fig. 7.3.

A. $\text{S}^{\mu}_{\text{im}}(e) = \{\boldsymbol{u}_2, \boldsymbol{u}_3\}$. (Notice that $\text{S}_{\text{im}}(e) = \{e, b, f, c\}$).

B. There is exactly one node c in $\text{S}_{\text{im}}(e)$, such that $\mu(c) = \boldsymbol{u}_3$. Since $\bar{\text{S}}_{\text{al}}(\boldsymbol{u}_3) = dc$, we have $s^{\boldsymbol{u}_3}_{\blacktriangleleft} = d$ and $s^{\boldsymbol{u}_3}_{\blacktriangleright}$ is an empty string.

C. There exist two nodes $b, f \in \text{S}_{\text{im}}(e)$, such that $\mu(b) = \mu(f) = \boldsymbol{u}_2$. We arbitrarily pick a string, say $bafbaf$, from $\text{S}^{f,b}_{\text{al}}(\boldsymbol{u}_2)$ and have $s^{\boldsymbol{u}_2}_{\blacktriangleleft} = ba$ and $s^{\boldsymbol{u}_2}_{\blacktriangleright} = af$.

D. Let $n = |\{\boldsymbol{u}_2, \boldsymbol{u}_3\}| = 2$; and $\boldsymbol{u}_2\boldsymbol{u}_3$ is a subsequence of the topological order of (V, E).

E. Create a dummy activity $\hat{e}$.

F. $\hat{\text{S}}(e) = s^{\boldsymbol{u}_2}_{\blacktriangleleft} s^{\boldsymbol{u}_3}_{\blacktriangleleft} \hat{e} s^{\boldsymbol{u}_3}_{\blacktriangleright} s^{\boldsymbol{u}_2}_{\blacktriangleright} = bad\hat{e}af$.

G. We have $aPre(e, b) \in C^{\text{X}}_{\text{al}}$.

H. $\hat{\text{S}}(e) = \bar{\text{S}}_{\text{al}}(\mu(b))bad\hat{e}af = \bar{\text{S}}_{\text{al}}(\boldsymbol{u}_2)bad\hat{e}af = bafbad\hat{e}af$

Similarly, $\hat{\text{S}}(b) = \hat{b}$, $\hat{\text{S}}(a) = dc\hat{a}$, $\hat{\text{S}}(f) = d\hat{f}$, $\hat{\text{S}}(c) = \hat{c}$, and $\hat{\text{S}}(d) = \hat{d}$. ∎

The following Lemma 7.4.30 is straightforward to prove based on Alg. 7; thus proof is omitted.

Lemma 7.4.30 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. For each $a \in A$ and each activity (or dummy activity) b occurring in $\hat{\mathrm{S}}(a)$, either $b = \hat{a}$ or there is a path from $\mu(a)$ to $\mu(b)$ in graph (V, E).

Let (V, E, μ) and (A, C) be as stated in Lemma 7.4.30. Lemma 7.4.30 states that each activity in $\hat{\mathrm{S}}(a)$, where $a \in A$, is either a dummy activity $\hat{a}$ or an activity that "orders after" a based on the topological order of (V, E).

Let (V, E, μ) be a grouped graph of the causality graph of a collapsed schema that satisfies the conditions in Theorem 7.4.23, for each $\boldsymbol{u} \in V$, we create a corresponding string $\hat{\mathrm{S}}(\boldsymbol{u})$ based on the following procedure:

1. Let $\hat{\mathrm{S}}(\boldsymbol{u})$ initially be $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u})$.
2. Then (non-recursively) replace each activity a occurring in $\hat{\mathrm{S}}(\boldsymbol{u})$ by $\hat{\mathrm{S}}(a)$.

Given a string s of (dummy) activities, denote $\mathrm{IM}(s)$ to be a string obtained by (non-recursively) replacing each dummy activity $\hat{a}$ in s by $\bar{\mathrm{S}}_{\mathrm{im}}(a)$.

Example 7.4.31 Continue with Example 7.4.29; $\hat{\mathrm{S}}(\boldsymbol{u}_1)$ initially is $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u}_1) = e$; then replace e by $\hat{\mathrm{S}}(e)$; we have $\hat{\mathrm{S}}(\boldsymbol{u}_1) = \hat{\mathrm{S}}(e) = bafbad\hat{e}af$. Accordingly, $\hat{\mathrm{S}}(\boldsymbol{u}_2) = \hat{\mathrm{S}}(b)\hat{\mathrm{S}}(a)\hat{\mathrm{S}}(f) = \hat{b}dc\hat{a}d\hat{f}$ and $\hat{\mathrm{S}}(\boldsymbol{u}_3) = \hat{\mathrm{S}}(d)\hat{\mathrm{S}}(c) = \hat{d}\hat{c}$. Further, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}_1)) = \mathrm{IM}(bafbad\hat{e}af) = bafbad\,\bar{\mathrm{S}}_{\mathrm{im}}(e)af = bafbadcfebaf$. Note that $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}_1))$ satisfies each internal alternating constraint in Fig. 7.3 as well as immediate and external alternating constraints that are related to $\mu_V^{-1}(\boldsymbol{u}_1) = \{e\}$, i.e., $aPre(e, b)$, $iPre(e, f)$, and $iRes(e, b)$. ∎

Up to this point, we have presented an approach to construct a string for each grouped activity, such that the string satisfies each internal alternating constraint, each related immediate constraint, and each related external alternating constraint (where the proofs are presented later). In the following, we step further to show that there is a way to construct a string that satisfies every given immediate and alternating constraint.

Let (V, E, μ) be a grouped graph of a causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. Given s as a string over (dummy) activities, for each activity $a \in A$, denote $s|_a$ to be a string obtained by (non-recursively) replacing each a in s by $\hat{\mathrm{S}}(a)$. Similarly, given $\{a_1, a_2, ..., a_n\} \in A$, we recursively define $s|_{a_1,a_2,...,a_n}$ as a string obtained by (non-recursively) replacing each a_n in $s|_{a_1,a_2,...,a_{n-1}}$ by $\hat{\mathrm{S}}(a_n)$. Let string $t = a_1a_2...a_n$, we may simply write $s|_{a_1,a_2,...,a_n}$ as $s|_t$ if the context is clear.

Example 7.4.32 Continue with Example 7.4.31; we have $\hat{\mathrm{S}}(\boldsymbol{u}_1)|_f = baf bad\hat{e}af|_f = ba\hat{\mathrm{S}}(f)bad\hat{e}a\hat{\mathrm{S}}(f) = bad\hat{f}bad\hat{e}ad\hat{f}$ and $\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{f,d} = bad\hat{f}bad\hat{e}ad\hat{f}|_d = ba\hat{\mathrm{S}}(d)\hat{f}ba\hat{\mathrm{S}}(d)\hat{e}a\hat{\mathrm{S}}(d)\hat{f} = bad\hat{f}bad\hat{e}ad\hat{f}$. ∎

Let (V, E, μ) be a grouped graph of a causality graph of a collapsed schema (A, C) that satisfies all conditions in Theorem 7.4.23, denote $\bar{A}_\mu$ to be a permutation of A, such that for each $a, b \in A$, a occurs before b in $\bar{A}_\mu$, if $\mu(a)$ occurs before $\mu(b)$ in the topological order of (V, E).

Example 7.4.33 Continue with Example 7.4.32; As the topological order of the grouped graph in Fig. 7.3 is $\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_3$, $\bar{A}_\mu$ can be $ebafcd$. ∎

Let (V, E, μ) be the grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. In the remainder of this section, we prove

that for each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in C. The proof of Theorem 7.4.23 is done by showing that $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in C_{im}, $C_{\mathrm{al}}^{\mathrm{X}}$, and $C_{\mathrm{al}}^{\mathrm{I}}$ respectively. And further extend the satisfiability to conformance to prove the correctness of the "if" direction of Theorem 7.4.23.

Example 7.4.34 Continue with Example 7.4.33; we consider $\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{\bar{A}_\mu}$ in the following, where the underline denotes the newly replaced strings (based on the previous result).

$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_e = bafbad\hat{e}af|_e = bafbad\hat{e}af$$
$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{e,b} = \underline{\hat{b}}af\underline{\hat{b}}ad\hat{e}af$$
$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{e,b,a} = \hat{b}\underline{dc\hat{a}}f\hat{b}\underline{dc\hat{a}}d\hat{e}\underline{dc\hat{a}}f$$
$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{e,b,a,f} = \hat{b}dc\hat{a}\underline{d\hat{f}}\hat{b}dc\hat{a}d\hat{e}dc\hat{a}\underline{d\hat{f}}$$
$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{e,b,a,f,c} = \hat{b}d\underline{\hat{c}}\hat{a}d\hat{f}\hat{b}d\underline{\hat{c}}\hat{a}d\hat{e}d\underline{\hat{c}}\hat{a}d\hat{f}$$
$$\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{e,b,a,f,c,d} = \hat{b}\underline{\hat{d}}\hat{c}\hat{a}\underline{\hat{d}}\hat{f}\hat{b}\underline{\hat{d}}\hat{c}\hat{a}\underline{\hat{d}}\hat{e}\underline{\hat{d}}\hat{c}\hat{a}\underline{\hat{d}}\hat{f}$$

Notice that $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{\bar{A}_\mu}) = \mathrm{IM}(\hat{b}\hat{d}\hat{c}\hat{a}\hat{d}\hat{f}\hat{b}\hat{d}\hat{c}\hat{a}\hat{d}\hat{e}\hat{d}\hat{c}\hat{a}\hat{d}\hat{f}) = bdcadcfbdcadcfebdcadcf$ satisfies every constraint in the schema shown in Fig. 7.3. ∎

3. Proof of the Conformance

Lemma 7.4.35 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies all the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in C.

To prove Lemma 7.4.35, we treat three different types of constraints: immediate, internal alternating, and external alternating constraints separately; and prove that all

three types of constraints can be satisfied (in Corollary 7.4.38, Lemma 7.4.39, and Lemma 7.4.47 respectively).

Satisfiability of all immediate constraints

Lemma 7.4.36 Given a schema (A, C) and a string s over $\{\hat{a} \mid a \in A\}$, $\mathrm{IM}(s)$ satisfies every constraint in C_{im}.

Proof: According to the definition of "IM", each activity $\hat{a}$ that occurs in s will be replaced by $\bar{\mathrm{s}}_{\mathrm{im}}(a)$, which satisfies every constraint in C_{im}. It is straightforward to show that $\mathrm{IM}(s)$ satisfies every constraint in C_{im}. ∎

Lemma 7.4.37 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema S that satisfies the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\hat{\mathrm{s}}(\boldsymbol{u})|_{\bar{A}_\mu}$ is a string over $\{\hat{a} \mid a \in A\}$.

Proof: Let $\boldsymbol{u}$, V, E, μ and $\bar{A}_\mu$ be as stated in the lemma; suppose that $\bar{A}_\mu = a_1 a_2 ... a_n$. We prove by induction that given $j \in [1..n]$, for each $i \in [1..j]$, a_i will not occur in $\hat{\mathrm{s}}(\boldsymbol{u})|_{a_1 a_2 ... a_j}$.

Basis: For $\hat{\mathrm{s}}(\boldsymbol{u})|_{a_1}$, all a_1's in $\hat{\mathrm{s}}(\boldsymbol{u})$ are replaced by $\hat{\mathrm{s}}(a_1)$ that contains no a_1's but $\hat{a}_1$'s. Hence $\hat{\mathrm{s}}(\boldsymbol{u})|_{a_1}$ does not contain a_1.

Induction: Suppose that $\hat{\mathrm{s}}(\boldsymbol{u})|_{a_1 a_2 ... a_{j-1}}$ contains no activity in $\{a_1, a_2, ..., a_{j-1}\}$. Consider string $\hat{\mathrm{s}}(\boldsymbol{u})|_{a_1 a_2 ... a_j}$. Based on Lemma 7.4.30, each activity b in $\hat{\mathrm{s}}(a_j)$ is either $\hat{a}_j$ or there is path from $\mu(a)$ to $\mu(b)$ in (V, E). According to the definition of $\bar{A}_\mu = a_1 a_2 ... a_n$, each activity in $\{a_1, a_2, ..., a_{j-1}\}$ will not occur in $\hat{\mathrm{s}}(a_j)$; otherwise it contradicts the topological order of (V, E). Therefore, $\hat{\mathrm{s}}(a_j)$ contains no activity in $\{a_1, a_2, ..., a_j\}$.

Based on the above induction, $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_n}$ contains no activity in $\{a_1, a_2, ..., a_n\}$ but only with activities in $\{\hat{a}_1, \hat{a}_2, ...\hat{a}_n\}$. Therefore, the lemma holds. ∎

The following (Corollary 7.4.38) is a direct consequence of Lemmas 7.4.36 and 7.4.37.

Corollary 7.4.38 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies all the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in C_{im}.

Satisfiability of all external alternating constraints

Lemma 7.4.39 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies all the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}$.

Proof: Let $\boldsymbol{u}$, V, E, μ and $\bar{A}_\mu$ be as stated in the lemma; suppose that $\bar{A}_\mu = a_1a_2...a_n$. We prove by induction that for each $i \in [1..n]$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i}\{a_j\})$.

Basis: For string $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1}$, according to Step G of Alg. 7, if $aRes(a_1, b) \in C^{\mathrm{X}}_{\mathrm{al}}$ (or $aPre(a_1, b) \in C^{\mathrm{X}}_{\mathrm{al}}$), where $b \in A$, then b will occur (may not immediately) to the right (left, resp.) of $\hat{a}_1$ in $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1}$. Thus, $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1}$ satisfies $aRes(\hat{a}_1, b)$ ($aPre(\hat{a}_1, b)$, resp.), which means that $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1})$ satisfies $aRes(a_1, b)$ ($aPre(a_1, b)$, resp.). Therefore, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\{a_1\})$.

Induction: Suppose $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_{i-1}})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i-1}\{a_j\})$.

Consider string $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i}$. Suppose that $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_{i-1}} = s_1a_is_2a_is_3...a_is_m$, where for each $j \in [1..m]$, s_j is a substring of $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i}$ such that s_j does not contain a_i. In other word, s_1, s_2, ..., s_m form a "partition" of $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i}$ based on a_i. Accordingly,

$\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i} = s_1\hat{\mathrm{S}}(a_i)s_2\hat{\mathrm{S}}(a_i)s_3...\hat{\mathrm{S}}(a_i)s_m$. Denote $\mathrm{IM}(\hat{\mathrm{S}}(a_i)) = s^a_{\blacktriangleleft}a_i s^a_{\blacktriangleright}$, where $s^a_{\blacktriangleleft}$ and $s^a_{\blacktriangleright}$ are substrings of $\mathrm{IM}(\hat{\mathrm{S}}(a_i))$ that does not contain a_i.

According to Lemma 7.4.30, $\hat{\mathrm{S}}(a_i)$ does not contain an activity in $\{a_1, a_2, ..., a_{i-1}\}$; Further, $\bar{\mathrm{S}}_{\mathrm{im}}(a_i)$, does not contain an activity in $\{a_1, a_2, ..., a_{i-1}\}$ as well; we have $\mathrm{IM}(\hat{\mathrm{S}}(a_i))$ does not contain an activity in $\{a_1, a_2, ..., a_{i-1}\}$, which indicates that neither $s^a_{\blacktriangleleft}$ nor $s^a_{\blacktriangleright}$ contains an activity in $\{a_1, a_2, ..., a_{i-1}\}$. Consider string $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i}) = \mathrm{IM}(s_1)$ $\mathrm{IM}(\hat{\mathrm{S}}(a_i))\mathrm{IM}(s_2)...\mathrm{IM}(\hat{\mathrm{S}}(a_i))\mathrm{IM}(s_m) = \mathrm{IM}(s_1)s^a_{\blacktriangleleft}a_i s^a_{\blacktriangleright}\mathrm{IM}(s_2)...s^a_{\blacktriangleleft}a_i s^a_{\blacktriangleright}\mathrm{IM}(s_m)$. Note that based on the hypothesis, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_{i-1}}) = \mathrm{IM}(s_1)a_i\mathrm{IM}(s_2)...a_i\mathrm{IM}(s_m)$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i-1}\{a_j\})$. Since both $s^a_{\blacktriangleleft}$ and $s^a_{\blacktriangleright}$ do not contain a_1, a_2, ..., or a_{i-1}, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i-1}\{a_j\})$.

Moreover, $\mathrm{IM}(\hat{\mathrm{S}}(a_i))$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\{a_i\})$ based on Step G of Alg. 7, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_i})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i-1}\{a_j\}) \cup C^{\mathrm{X}}_{\mathrm{al}}(\{a_j\}) = C^{\mathrm{X}}_{\mathrm{al}}(\bigcup_{j=1}^{i}\{a_i\})$.

Based on the above induction, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in $C^{\mathrm{X}}_{\mathrm{al}}$. ∎

Satisfiability of all internal alternating constraints

Lemma 7.4.40 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. For each $a \in A$, $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}} - C^{\mathrm{I}}_{\mathrm{al}}(\mu_V^{-1}(\mu(a)))$.

Proof: Let V, E, μ, A, and $\boldsymbol{u}$ be as stated in the lemma. Let b and c be two activities in A, such that $aRes(b, c) \in C$ (for case $aPre(b, c)$, the analysis is similar), $\mu(b) = \mu(c)$, and $\mu(b) \neq \mu(a)$. Note that if we can prove $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$, the lemma holds.

If b and c do not occur in $\mathrm{IM}(\hat{\mathrm{S}}(a))$, then $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$. Otherwise, we need to analyze when b and c are "introduced" to $\mathrm{IM}(\hat{\mathrm{S}}(a))$ and why $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$. Note that in Alg. 7, there are only two steps (F and G) that are to construct

$\hat{\mathrm{S}}(a)$, where possibly b and c are introduced into $\mathrm{IM}(\hat{\mathrm{S}}(a))$. Hence, in the following, we analyze these two steps separately.

Case (1): b and c are introduced in Step F. Let string $\hat{\mathrm{S}}(a) = s_{\blacktriangleleft}^{\boldsymbol{u}_1} s_{\blacktriangleleft}^{\boldsymbol{u}_2} ... s_{\blacktriangleleft}^{\boldsymbol{u}_n} \hat{a} s_{\blacktriangleright}^{\boldsymbol{u}_n} ... s_{\blacktriangleright}^{\boldsymbol{u}_1}$ be as stated in Step F. Based on Step D, $\mathrm{S}_{\mathrm{im}}^{\mu}(a) = \{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n\}$ is a set; hence, for each distinct $i, j \in [1..n]$, $\boldsymbol{u}_i \neq \boldsymbol{u}_j$, which further indicates that there exists exactly one $i \in [1..n]$, such that $\mu(b) = \mu(c) = \boldsymbol{u}_i$. Note that $\mathrm{IM}(\hat{\mathrm{S}}(a)) = \mathrm{IM}(s_{\blacktriangleleft}^{\boldsymbol{u}_1} s_{\blacktriangleleft}^{\boldsymbol{u}_2} ... s_{\blacktriangleleft}^{\boldsymbol{u}_n} \hat{a} s_{\blacktriangleright}^{\boldsymbol{u}_n} ... s_{\blacktriangleright}^{\boldsymbol{u}_1}) = s_{\blacktriangleleft}^{\boldsymbol{u}_1} s_{\blacktriangleleft}^{\boldsymbol{u}_2} ... s_{\blacktriangleleft}^{\boldsymbol{u}_n} \bar{\mathrm{S}}_{\mathrm{im}}(a) s_{\blacktriangleright}^{\boldsymbol{u}_n} ... s_{\blacktriangleright}^{\boldsymbol{u}_1}$. According to Steps B and C, we have $s_{\blacktriangleleft}^{\boldsymbol{u}_i} \bar{\mathrm{S}}_{\mathrm{im}}(a) s_{\blacktriangleright}^{\boldsymbol{u}_i}$ satisfies $aRes(b, c)$. Therefore, $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$ at the end of Step F.

Case (2): b and c are introduced in Step G. Let $\hat{\mathrm{S}}(a)$ be as stated at the end of Step F. According to Case (1), $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$. Note that for this step, b and c may have already been introduced into $\mathrm{IM}(\hat{\mathrm{S}}(a))$. Suppose for some $d \in A$, $\mu(b) = \mu(c) = \mu(d)$, which denotes that both b and c will occur in $\bar{\mathrm{S}}_{\mathrm{al}}(\mu(d))$. If $aRes(a, d) \in C_{\mathrm{al}}^{\mathrm{X}}$, based on Step G, the new $\hat{\mathrm{S}}(a)$ (rename to $\hat{\mathrm{S}}(a)^{\mathrm{new}}$ to avoid confusion) will become $\hat{\mathrm{S}}(a)\bar{\mathrm{S}}_{\mathrm{al}}(\mu(d))$, which is the only way b and c are introduced into $\hat{\mathrm{S}}(a)^{\mathrm{new}}$. (For case $aPre(a, d)$, the analysis is similar.) Now consider $\mathrm{IM}(\hat{\mathrm{S}}(a)^{\mathrm{new}}) = \mathrm{IM}(\hat{\mathrm{S}}(a))\mathrm{IM}(\bar{\mathrm{S}}_{\mathrm{al}}(\mu(d))) = \mathrm{IM}(\hat{\mathrm{S}}(a))\bar{\mathrm{S}}_{\mathrm{al}}(\mu(d))$. According to Case (1), $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$; further, $\bar{\mathrm{S}}_{\mathrm{al}}(\mu(d))$ satisfies $aRes(b, c)$ by definition. As a result, $\mathrm{IM}(\hat{\mathrm{S}}(a)^{\mathrm{new}})$ satisfies $aRes(b, c)$. By induction, it is easy to show that at the end of Step G, the newly constructed $\hat{\mathrm{S}}(a)$ preserves the property that $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies $aRes(b, c)$.

As a summary, the lemma holds. ∎

Lemma 7.4.41 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}))$ satisfies each constraint in $C_{\mathrm{al}}^{\mathrm{I}}$.

Proof: Let V, E, μ, A, and $\boldsymbol{u}$ be as stated in the lemma. Note that $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}))$ can be obtained by (non-recursively) replacing each activity a in $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u})$ by $\mathrm{IM}(\hat{\mathrm{S}}(a))$. According to Lemma 7.4.40, for each $a \in \mu_V^{-1}(\boldsymbol{u})$, $\mathrm{IM}(\hat{\mathrm{S}}(a))$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}} - C^{\mathrm{I}}_{\mathrm{al}}(\mu_V^{-1}(\boldsymbol{u}))$. It is easy to have $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}))$ satisfying each constraint in $C^{\mathrm{I}}_{\mathrm{al}} - C^{\mathrm{I}}_{\mathrm{al}}(\mu_V^{-1}(\boldsymbol{u}))$. Moreover, as $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u})$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}}(\mu_V^{-1}(\boldsymbol{u}))$ and for each $a \in \mu_V^{-1}(\boldsymbol{u})$, a occurs exactly once in $\mathrm{IM}(\hat{\mathrm{S}}(a))$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}))$ satisfy each constraint in $C^{\mathrm{I}}_{\mathrm{al}}$. ∎

Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. Similar to the techniques used in the proofs of Lemmas 7.4.37 and 7.4.39, mathematical induction can be used to take the result of Lemma 7.4.41 as the basis to prove that for each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ also satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}}$. However, the induction cannot be trivially achieved based on the size of $\bar{A}_\mu$. To have better presentation of our proof, in the following, we introduce a tree structure for string $\hat{\mathrm{S}}(a)$ (where $a \in A$), $\hat{\mathrm{S}}(\boldsymbol{u})$, and $\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu}$; and present the induction based on the tree.

Let V, E, μ, A, a, and $\boldsymbol{u}$ be as stated above. The tree structure is an "extension" of strings $\hat{\mathrm{S}}(a)$, $\hat{\mathrm{S}}(\boldsymbol{u})$, and $\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu}$ that contains more information when constructing them (according to Alg. 7). More precisely, an *extension tree* for $\hat{\mathrm{S}}(a)$, denoted as $\mathrm{T}(\hat{\mathrm{S}}(a))$ is a tree constructed based on the following procedures:

1. Initially $\mathrm{T}(\hat{\mathrm{S}}(a))$ is with a root labeled with "$\hat{\mathrm{S}}(a)$".
2. Let string $s_{\blacktriangleleft}^{\boldsymbol{u}_1} s_{\blacktriangleleft}^{\boldsymbol{u}_2} ... s_{\blacktriangleleft}^{\boldsymbol{u}_n} \hat{a} s_{\blacktriangleright}^{\boldsymbol{u}_n} ... s_{\blacktriangleright}^{\boldsymbol{u}_1}$ be as stated in Step F of Alg. 7 (with the corresponding inputs for activity $a \in A$). Assign nodes "$s_{\blacktriangleleft}^{\boldsymbol{u}_1}$", "$s_{\blacktriangleleft}^{\boldsymbol{u}_2}$", ..., "$s_{\blacktriangleleft}^{\boldsymbol{u}_n}$", "$\hat{a}$", "$s_{\blacktriangleright}^{\boldsymbol{u}_n}$", ..., "$s_{\blacktriangleright}^{\boldsymbol{u}_1}$" to be the children of root "$\hat{\mathrm{S}}(a)$" orderly from left to right.
3. For each $i \in [1..n]$, if $s_{\blacktriangleleft}^{\boldsymbol{u}_i} = a_1 a_2 ... a_m$, assign "$a_1$", "$a_2$", ..., "$a_n$" to be the children of node "$s_{\blacktriangleleft}^{\boldsymbol{u}_i}$" orderly from left to right; apply the similar approach for $s_{\blacktriangleright}^{\boldsymbol{u}_i}$'s.
4. For each iteration of Step G of Alg. 7, do the following:

(a) $\mathrm{T}(\hat{\mathrm{S}}(e))$ (b) $\mathrm{T}(\hat{\mathrm{S}}(b))$ (c) $\mathrm{T}(\hat{\mathrm{S}}(a))$

Figure 5: Extension trees for a single activity

i. Let $aRes(a, b)$ (or $aPre(a, b)$) be as stated in Step G. Assign node "$\bar{\mathrm{S}}_{\mathrm{al}}(\mu(b))$" to be the rightmost (resp., leftmost) child of root "$\hat{\mathrm{S}}(a)$".

ii. If $\bar{\mathrm{S}}_{\mathrm{al}}(\mu(b)) = b_1 b_2 ... b_k$, assign "$b_1$", "$b_2$", ..., "$b_k$" to be the children of node "$\bar{\mathrm{S}}_{\mathrm{al}}(\mu(b))$" orderly from left to right.

Example 7.4.42 Continue with Example 7.4.29. Fig. 5 shows the extension tress for $\hat{\mathrm{S}}(e)$, $\hat{\mathrm{S}}(b)$, $\hat{\mathrm{S}}(a)$ respectively.

It is easy to observe that the leaf nodes under the depth-first search order [4] of such trees above form the corresponding strings of the roots. For example, if we do a depth-first search upon $\mathrm{T}(\hat{\mathrm{S}}(e))$ in Fig. 5(a) and print out the leaf nodes, the string will be "$bafbad\hat{e}af$", which is the same as $\hat{\mathrm{s}}(e)$. (Note that "ε" is an empty string). ∎

Let V, E, μ, A, a, and $\boldsymbol{u}$ be as stated above. An *extension tree* for $\hat{\mathrm{S}}(\boldsymbol{u})$, denoted as $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}))$ is a tree constructed based on the following procedures:

1. Initially $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}))$ is with a root labeled with "$\hat{\mathrm{S}}(\boldsymbol{u})$".
2. For each activity a in $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u})$ from left to right, attach $\mathrm{T}(\hat{\mathrm{S}}(a))$ as a subtree to root "$\hat{\mathrm{S}}(\boldsymbol{u})$" (from left to right).

[4] In the remainder of this chapter, we assume the depth-first search follows the order from left to right.

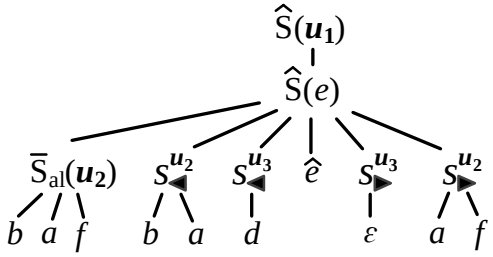

Figure 6: An extension tree for a grouped node

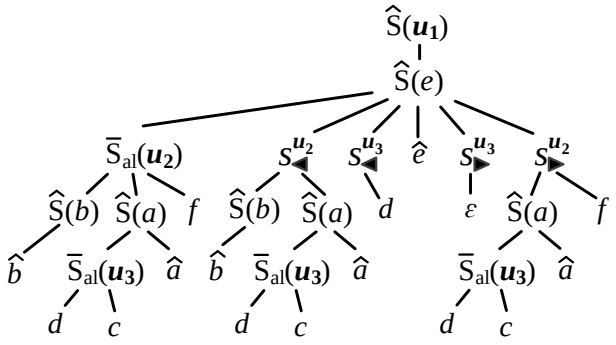

Figure 7: The extension tree for $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{eba})$

Example 7.4.43 Continue with Examples 7.4.42 and 7.4.31. Fig. 6 shows the extension tree $\hat{\mathrm{S}}(\boldsymbol{u}_1)$. ∎

Let V, E, μ, A, a, and $\boldsymbol{u}$ be as stated above. Suppose $a_1, a_2, ..., a_n$ are distinct activities in A. An *extension tree* for $\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_n}$, denoted as $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_2...a_n})$ is a tree constructed based on the following procedures:

1. $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1})$ is constructed by replacing each leaf node with label "a_1" in $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}))$ by tree $\mathrm{T}(\hat{\mathrm{S}}(a_1))$.
2. For each i from 1 to n, $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_1...a_i})$ is constructed by replacing each leaf node with label "a_i". in $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_1a_1...a_{i-1}})$ by tree $\mathrm{T}(\hat{\mathrm{S}}(a_i)$.

Example 7.4.44 Continue with Examples 7.4.43 and 7.4.34. Fig. 7 shows the extension tree $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{eba})$. ∎

Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. Let t be an extension tree with respect to (V, E, μ) and (A, C); we extend the mapping function μ to each node of t as follows:

- If the node is of form a, where $a \in A$, then $\mu(a)$ is the same as defined before,

- If the node is of form $\hat{a}$, where $a \in A$, then let $\mu(\hat{a})$ be $\mu(a)$,
- If the node is of form $\hat{\mathrm{S}}(a)$, where $a \in A$, then let $\mu(\hat{\mathrm{S}}(a))$ be $\mu(a)$,
- If the node is of form $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u})$, then let $\mu(\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u}))$ be $\boldsymbol{u}$, and
- If the node is of form $s^{\boldsymbol{u}}_{\blacktriangleleft}$ or $s^{\boldsymbol{u}}_{\blacktriangleright}$, then let $\mu(s^{\boldsymbol{u}}_{\blacktriangleleft})$ or $\mu(s^{\boldsymbol{u}}_{\blacktriangleright})$ be $\boldsymbol{u}$.

The following two Lemmas 7.4.45 and 7.4.46 are trivial to prove; thus the proof is omitted. Specifically, the correctness of the two lemmas is directly based on the definition of extension trees.

Lemma 7.4.45 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. Let t be an extension tree with respect to (V, E, μ) and (A, C). For each pair of node A and B in t, if A is the ancestor of B, then $\mu(A)$ is before or the same as $\mu(B)$ with respect to the topological order of (V, E).

Lemma 7.4.46 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies the conditions in Theorem 7.4.23. The leaf nodes under the depth-first search order of trees $\mathrm{T}(\hat{\mathrm{S}}(a))$ (where $a \in A$), $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u}))$ (where $\boldsymbol{u} \in V$), and $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ are the same as the strings $\hat{\mathrm{S}}(a)$, $\hat{\mathrm{S}}(\boldsymbol{u})$, and $\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu}$ respectively.

Lemma 7.4.47 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies all the conditions in Theorem 7.4.23. For each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}}$.

Proof: Let $\boldsymbol{u}$, V, E, μ and $\bar{A}_\mu$ be as stated in the lemma; suppose that $\bar{A}_\mu = a_1 a_2 ... a_n$. Denote a_0 to be an activity that is not in A. We prove by induction that for each $i \in [0..n]$, $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 ... a_i})$ satisfies each constraint in C^{I}_{al}. Note that since $a_0 \notin A$, $\hat{\text{S}}(\boldsymbol{u})|_{a_0} = \hat{\text{S}}(\boldsymbol{u})$, which means that if the induction holds, the lemma holds.

Basis: For string $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0})$, it is equivalent to $\text{IM}(\hat{\text{S}}(\boldsymbol{u}))$. Based on Lemma 7.4.41, $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0})$ satisfies every constraint in C^{I}_{al}.

Induction: Suppose $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_{i-1}})$ satisfies each constraint in C^{I}_{al}. Consider string $\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i}$, or correspondingly, the tree $\text{T}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$. Suppose there exist $b, c \in A$, such that $aRes(b, c) \in C^{\text{I}}_{\text{al}}$ and $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$ does not satisfy $aRes(b, c)$ (where case $aPre(b, c)$ can be analyzed symmetrically). Suppose $V = \{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n\}$ and sequence $\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n$ follows the topological order of (V, E). Without loss of generality, assume $\mu(b) = \mu(c) = \boldsymbol{u}_k$, where $k \in [1..n]$.

Based on the hypothesis, $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_{i-1}})$ satisfies $aRes(b, c)$. Due to the fact that $\text{T}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$ is constructed by replacing each leaf node a_i in $\text{T}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_{i-1}})$ by tree $\text{T}(\hat{\text{S}}(a_i))$, $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$ can violate $aRes(b, c)$ only because of two cases: (1) $\text{IM}(\hat{\text{S}}(a_i))$ only contains b but not c, and (2) $\text{IM}(\hat{\text{S}}(a_i))$ contains both b and c. Note that if $\text{IM}(\hat{\text{S}}(a_i))$ contains neither b or c, or only c but not b, $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$ still satisfies $aRes(b, c)$.

Case (1): According to Steps F and G, of Alg. 7, if an activity $d \in A$ is "introduced" to $\text{IM}(\hat{\text{S}}(a_i))$ during Step F or G, then all the activities in $\mu_V^{-1}(\mu(d))$ will occur in $\text{IM}(\hat{\text{S}}(a_i))$. Since $\mu(b) = \mu(c)$ and only b occurs in $\text{IM}(\hat{\text{S}}(a_i))$, based on Lemma 7.4.30, b can only be a_i. Therefore, $\text{T}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_i})$ is constructed by replacing each leaf node b in $\text{T}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_{i-1}})$ by tree $\text{T}(\hat{\text{S}}(b))$, which indicates that $\text{IM}(\hat{\text{S}}(\boldsymbol{u})|_{a_0 a_1 a_2 ... a_{i-1}})$ does not satisfies $aRes(b, c)$, a contradiction.

Case (2): Similar to Case (1), based on Lemma 7.4.30, there is path from $\mu(a_i)$ to $\boldsymbol{u}_k$. According to Lemma 7.4.40, since $\mu(a_i) \neq \boldsymbol{u}_k$, $\text{IM}(\hat{\text{S}}(a_i))$ satisfies $aRes(b, c)$.

Therefore the only way for $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_i})$ to violate $aRes(b,c)$ is that for tree $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_{i-1}})$, there exists a node b or $\hat{b}$ and a node a_i in $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_{i-1}})$, such that there is no node c between $b/\hat{b}$ and a_i based on the depth-first search order. In the following, we develop our proof by considering the parent of node b or $\hat{b}$ in tree $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_{i-1}})$. There are four cases: (i) $\hat{\mathrm{S}}(b)$, (ii) $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u}_k)$, (iii) $s_{\blacktriangleright}^{\boldsymbol{u}_k}$, or (iv) $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$.

Case (i): if $\hat{\mathrm{S}}(b)$ occurs in tree $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_{i-1}})$, then $b \in \{a_1,...,a_{i-1}\}$. This is impossible as $\mu(b) = \boldsymbol{u}_k$ is after $\mu(a_i)$ in terms of the topological order of (V,E).

Case (ii): for node $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u}_k)$, according to the definition of "$\bar{\mathrm{S}}_{\mathrm{al}}$", c must be a child of $\bar{\mathrm{S}}_{\mathrm{al}}(\boldsymbol{u}_k)$ and is to the (may not be directly) right of node b or $\hat{b}$. Since $\boldsymbol{u}_k$ is after $\mu(a_i)$ in terms of the topological order of (V,E), according to Lemma 7.4.46, a_i cannot be a child or descendant of a node that is between $b/\hat{b}$ and c. Therefore, a_i can only be after c based on the depth-first search order, a contradiction.

Case (iii): based on the definition of $s_{\blacktriangleright}^{\boldsymbol{u}_k}$, c is also a child of $s_{\blacktriangleright}^{\boldsymbol{u}_k}$ and is to the (may not be directly) right of node b or $\hat{b}$. Similar to the analysis of Case (ii), a_i can only be after c based on the depth-first search order, a contradiction.

Case (iv): If $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$ is the parent of b or $\hat{b}$, we need to consider the parent of $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$, say $\hat{\mathrm{S}}(e)$, where $e \in A$. According to Step F of Alg. 7, $s_{\blacktriangleright}^{\boldsymbol{u}_k}$ is also a child of $\hat{\mathrm{S}}(e)$ and is to the (may not be directly) right of $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$. Based on the definition of $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$ and $s_{\blacktriangleright}^{\boldsymbol{u}_k}$ in Step B or C in Alg. 7, c is a child of $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$ or $s_{\blacktriangleright}^{\boldsymbol{u}_k}$. If c is a child of $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$, then the analysis is the same as Case (iv), we can achieve a contradiction. If c is a child of $s_{\blacktriangleright}^{\boldsymbol{u}_k}$, consider a node $s_{\blacktriangleleft}^{\boldsymbol{u}_j}$ between $s_{\blacktriangleleft}^{\boldsymbol{u}_k}$ and $s_{\blacktriangleright}^{\boldsymbol{u}_k}$, where according to Step F of Alg. 7, $j \in [(k+1)...n]$. (For case $s_{\blacktriangleright}^{\boldsymbol{u}_j}$, the analysis is the same). Suppose $\mu(a_i) = s_{\blacktriangleright}^{\boldsymbol{u}_l}$, where $l \in [1..n]$. Since b is node in tree $\mathrm{T}(\hat{\mathrm{S}}(a_i))$, it is easy to have $l < k$, which indicates that $j > l$. According to Lemma 7.4.46, it is impossible for a_i to occur as child or descendant in the subtree of $s_{\blacktriangleright}^{\boldsymbol{u}_j}$. Hence, a_i can only occur after $s_{\blacktriangleright}^{\boldsymbol{u}_k}$ based on the depth-first search order, which means that a_i can only occur after c based on the depth-first search order, a contradiction.

As a summary, for each pair of node b or $\hat{b}$ and a_i in tree $\mathrm{T}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_{i-1}})$, there is a node c in between based on the depth-first search order. As a result, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{a_0a_1a_2...a_i})$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}}$.

Based on the above induction, for each $\boldsymbol{u} \in V$, $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u})|_{\bar{A}_\mu})$ satisfies each constraint in $C^{\mathrm{I}}_{\mathrm{al}}$. ∎

The correctness of Lemma 7.4.35 directly follows from Corollary 7.4.38, Lemma 7.4.39, and Lemma 7.4.47.

The following Corollaries 7.4.48 directly follows from Lemma 7.4.35.

Corollary 7.4.48 Let (V, E, μ) be grouped graph of the causality graph of a collapsed schema (A, C) that satisfies all the conditions in Theorem 7.4.23. Suppose $V = \{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n\}$. $\mathrm{IM}(\hat{\mathrm{S}}(\boldsymbol{u}_1)|_{\bar{A}_\mu}\hat{\mathrm{S}}(\boldsymbol{u}_2)|_{\bar{A}_\mu}...\hat{\mathrm{S}}(\boldsymbol{u}_n)|_{\bar{A}_\mu})$ conforms S.

The "if" direction of Theorem 7.4.23 holds based on Corollary 7.4.48.

In summary, we have provided the syntactical conditions for conformance involving (1) ordering and immediate, (2) ordering and alternating, or (3) alternating and immediate constraints. Syntactical characterization of conformance for schemas that contain all three types constraints remains an open problem.

7.5 Response or Precedence Constraints

In this section, we study comformity of either response constraints or precedence constraints but not combined. The following Theorem 7.5.1 states the syntactical condition for conformity of schemas containing only response constraints or only precedence constraints.

Algorithm 8

Input: A causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleright})$ of a schema S that satisfies both conditions in Theorem 7.5.1

Output: A finite string that conforms to S

A. Let string s be a topological order of $(A, E^{\text{or}\cup\text{al}\cup\text{im}}_{\blacktriangleright})$. For each $a \in A$, let $\hat{s}(a)$ be the substring $s^{[k]}s^{[k+1]}...s^{[len(s)]}$ of s such that $s^{[k]} = a$ (clearly $k \in [1..len(s)]$). Let $i = 1$.

B. While $i \leqslant len(s)$, repeat the following step:

B1. If $(s^{[i]}, v) \in E^{\text{im}}_{\blacktriangleright}$ for some $v \in A$ and either $i = len(s)$ or $s^{[i+1]} \neq v$, then replace $s^{[i]}$ in s by $s^{[i]}\hat{s}(v)$.

B2. Increment $i = i + 1$.

C. Return s.

Theorem 7.5.1 Given a schema $S = (A, C)$ where C contains only response (or only precedence) constraints, and its causality graph $(A, E^{\text{or}}_{\blacktriangleright}, E^{\text{al}}_{\blacktriangleright}, E^{\text{im}}_{\blacktriangleright})$ (resp. $(A, E^{\text{or}}_{\blacktriangleleft}, E^{\text{al}}_{\blacktriangleleft}, E^{\text{im}}_{\blacktriangleleft})$), S is conformable iff the following conditions both hold:

(1). $(A, E^{\text{or}\cup\text{al}\cup\text{im}}_{\blacktriangleright})$ (resp. $E^{\text{or}\cup\text{al}\cup\text{im}}_{\blacktriangleleft}$) is acyclic, and

(2). for each $(u, v) \in E^{\text{im}}_{\blacktriangleright}$ (resp. $E^{\text{im}}_{\blacktriangleleft}$), there does not exist any $w \in A$ such that $w \neq v$ and $(u, w) \in E^{\text{im}}_{\blacktriangleright}$ (resp. $E^{\text{im}}_{\blacktriangleleft}$).

The "only if" direction follows from Lemma 7.3.4, Theorems 7.3.2 and 7.3.5. The "if" direction is involved and is established as Lemma 7.5.5.

In the following, we only focus on the schema with only response constraints (the case for precedence constraints is similar).

Alg. 8 is used to construct a conforming string from an input schema that satisfies both conditions in Theorem 7.5.1. The main idea is again to build a topological order based on the causality graph and then fix each violated immediate constraint in the string. Note that the execution of Alg. 8 replies on Theorem 7.5.1, where Conditions (1) is to ensure the topological order in Step A is achievable, and Condition (2) is to guarantee Step B1 is unique.

Example 7.5.2 Given a schema with activities a, b, c, d, and e, and the following constraints.

ordering	$Res(a,d),\ Res(d,b)$
alternative	$aRes(a,c),\ aRes(c,e),\ aRes(b,c)$
immediate	$iRes(a,b),\ iRes(b,e)$

Alg. 8 on the above schema as the input produces the following execution details (with each "while" iteration in Step B unfolded).

A. $s = adbce$, $\hat{s}(a) = adbce$, $\hat{s}(d) = dbce$, $\hat{s}(b) = bce$, $\hat{s}(c) = ce$, and $\hat{s}(e) = e$.

B. When $i = 1$ ($len(s) = 5$): $s^{[1]} = a$ and $s^{[2]} = d$ violate $iRes(a,b)$. We replace $s^{[1]}$ by $a\hat{s}(b) = abce$; we have $s = abcedbce$.

B. When $i = 2$ ($len(s) = 8$): $s^{[2]} = b$ and $s^{[3]} = c$ violate $iRes(b,e)$. Replacing $s^{[2]}$ by $b\hat{s}(e) = be$, we have $s = abecedbce$.

B. When $i = 3$ through 6 ($len(s) = 9$): no immediate constraint is violated; do nothing.

B. When $i = 7$ ($len(s) = 9$): $s^{[7]} = b$ and $s^{[8]} = c$ violate $iRes(b,e)$. Replacing $s^{[7]}$ by $b\hat{s}(e) = be$; we have $s = abecedbece$.

B. When $i = 8, 9$ ($len(s) = 10$): no immediate constraint is violated; do nothing.

C. Finally, the algorithm returns $abecedbece$.

Note that $abecedbece$ conforms to the schema. ∎

Lemma 7.5.3 Let S be a schema containing only response constraints and satisfying both conditions in Theorem 7.5.1. Alg. 8 terminates on input S.

Proof: Let $S = (A, C)$ be the schema in the lemma statement and $\hat{s}$ the topological order of the causality graph $(A, E_{\blacktriangleright}^{\mathrm{or} \cup \mathrm{al} \cup \mathrm{im}})$ of S assigned to the vairable s at Step A of Alg. 8. Wlog, let $\hat{s} = a_1 ... a_\ell$ where $\ell = len(\hat{s})$.

Note that Steps A and C of the algorithm are executed exactly once. If s_{out} is the output string, i.e., held by variable s at step C, then Step B is executed once for each activity occurrence in s_{out} in the left-to-right order (small to large indices). We argue that $len(s_{\text{out}})$ is finite.

Observe that (1) the variable s initially holds the string $\hat{s}$ and (2) if Step B replaces an occurrence of a_i, the replacement string contains none of $a_1, ..., a_{i-1}$. Thus, the longest string to replace a_i is $a_i a_{i+1} ... a_\ell$.

Consider the worst case that each replacement at Step B is by the longest string in an execution of the algorithm. Let λ_i be the number of occurrences of a_i in s_{out}. It is easy to establish that $\lambda_1 = 1$ (a_1 is never in a replacement string of other activities), and $\lambda_i = \Sigma_1^{i-1} \lambda_i + 1$. It can be verified that $\lambda_i = 2^{i-1}$ is a solution. Thus, $len(s_{\text{out}}) \leqslant 2^{\ell-1}$. ∎

Lemma 7.5.4 Given a schema $S = (A, C)$ with only response constraints that satisfies both conditions in Theorem 7.5.1, the output string by Alg. 8 on input S satisfies each immediate constraint in C.

Proof: Let $s, \hat{s}(a)$, and i be as stated in Alg. 8. The lemma is easy to verify since for the i^{th} iteration, if pair $s^{[i]}$ and $s^{[i+1]}$ violates an immediate constraint in C, $s^{[i]}$ is replaced by $s^{[i]}\hat{s}(v)$ where $v \in A$ and $s^{[i]}\hat{s}(v)^{[1]}$ satisfies this constraint. Note that according to Condition (2) of Theorem 7.5.1, v is unique. Since Alg. 8 terminates according to Lemma 7.5.3, s satisfies each immediate constraint in C. ∎

Lemma 7.5.5 The "if" direction of Theorem 7.5.1 holds.

Proof: In this proof we only consider the case when a schema contains only response constraints (for precedence constraints, the proof is similar). Let S, A, C, $E_{\blacktriangleright}^{\text{or}}$, $E_{\blacktriangleright}^{\text{al}}$, and

$E_{\blacktriangleright}^{\text{im}}$ be as stated in Theorem 7.5.1. We analyze Alg. 8 step by step with S as input. Let s, $\hat{s}$, and i be as stated in Alg. 8. We prove by induction that after the i^{th} iteration of Step B, s satisfies each ordering and alternating constraint in C.

For reading convenience and proof consistency, denote s_0 as the string the constructed s before Step B; and for each $i > 0$, denote s_i as the constructed s after the i^{th} of Step B.

Basis: Before the first iteration of Step B, similar to the proof techniques used for Lemma 7.3.2, it is easy to show that s_0 satisfies each ordering constraint in C. Moreover, since each activity in A occurs in s_0 exactly once, s_0 satisfies each alternating constraint in C.

Suppose after the $(i-1)^{\text{th}}$ iteration for some $i > 0$, s_{i-1} satisfies each ordering and alternating constraint in C.

Induction: Then, during the i^{th} iteration, we will have three cases: (1) there does not exist $u \in A$, such that $iRes(s_{i-1}^{[i]}, u) \in C$, (2) there exists $u \in A$, such that $iRes(s_{i-1}^{[i]}, u) \in C$ and $s_{i-1}^{[i+1]} = u$, and (3) there exists $u \in A$, such that $iRes(s_{i-1}^{[i]}, u) \in C$ and $s_{i-1}^{[i+1]} \neq u$. If case (1) or (2) holds, $s_i = s_{i-1}$ satisfies each ordering and alternating constraint in C according to the hypothesis. Now we focus on case (3).

Let u and v be as stated above. Suppose there exists an ordering constraint $Res(a, b) \in C$, such that s_{i-1} satisfies $Res(a, b)$ and s_i does not. Since s_i is obtained from s_{i-1} by replacing $s_{i-1}^{[i]} = u$ by $u\hat{s}(v)$, $\hat{s}(v)$ must contain a and either (i) string $s_i^{[i+1]} s_i^{[i+2]} ... s_i^{[len(s_i)]}$ does not contain b, or (ii) b occurs before a in $\hat{s}(v)$ and string $s_{i-1}^{[i+1]} s_{i-1}^{[i+2]} ... s_{i-1}^{[len(s_{i-1})]}$ does not contain b. We now argue that (i) and (ii) cannot happen simultaneously. Since $(a, b) \in E_{\blacktriangleright}^{\text{or} \cup \text{al} \cup \text{im}}$, b occurs after a in $\hat{s}$ according to the topological order. Thus, if $\hat{s}(v)$ contains a, then $\hat{s}(v)$ contains b, and b occurs after a in $\hat{s}(v)$. For (i), $s_i^{[i+1]} s_i^{[i+2]} ... s_i^{[len(s_i)]}$ contains substring $\hat{s}(v)$, thus contains b; and for (ii), b occurs after a in $\hat{s}(v)$. Therefore, s_i satisfies each ordering constraint in C.

Suppose there exists an ordering constraint $aRes(a,b) \in C$, such that s_{i-1} satisfies $aRes(a,b)$ and s_i does not. Based on Lemma 7.3.4 and the above reasoning, s_i satisfies $Res(a,b)$. Then s_i does not satisfy $aRes(a,b)$ only because that (i) $\hat{s}(v)$ contains a, and (ii) in s_{i-1} there exists $j,k \in \mathbb{N}^+$, such that $j \leqslant i < k$, $s_{i-1}^{[j]} = a$, $s_{i-1}^{[k]} = b$, and for each $m \in [j..(k-1)]$, $s_{i-1}^{[m]} \neq a$ or b. Consider node $w = s_{i-1}^{[i]}$; note that w occurs before a in $\hat{s}$; otherwise a will not be contained in $\hat{s}(v)$. Thus wab is a subsequence of $\hat{s}$. Since $s_{i-1}^{[i]}$ is w and $s_{i-1}^{[k]}$ is b, there must exist $n \in [(i+1)..(k-1)]$, such that $s_{i-1}^{[n]}$ is a, which contradicts with for each $m \in [j..(k-1)]$, $s_{i-1}^{[m]} \neq a$. Therefore, s_i satisfies each alternating constraint in C.

According to the above induction, since Alg. 8 terminates by Lemma 7.5.3, s satisfies each ordering and alternating constraint in C. Together with Lemma 7.5.4, s satisfies each constraint in C. As each activity occurs in s at least once, s conforms S. ∎

7.6 Experimental Evaluations

In this section, several experiments are conducted to evaluate the performance of the syntactic-condition-based conformance checking approaches. Three main types of algorithms are implemented, including: (1) The naive algorithm to check DecSerFlow conformance using automata (denoted as Chk-A), (2) the syntactic-condition-based conformance checking algorithms for all four combinations of predicates (denoted as Chk-Or-Im for ordering and immediate constraints, Chk-Or-Al, Chk-Al-Im, and Chk-Sin for single direction constraints, i.e., either response or precedence), and (3) all four conforming string generation algorithms (denoted as Gen-Or-Im, Gen-Or-Al, Gen-Al-Im, and Gen-Sin). All algorithms are implemented in Java and executed on a computer with 8G RAM and dual 1.7 GHz Intel processors. The data sets (i.e., DecSerFlow schemas) used in experiments are randomly generated. Schema generation uses two parameters: number

of activities ($\#A$) and number of constraints ($\#C$), where each constraint is constructed by selecting a DecSerFlow predicate and two activities in a uniform distribution. Each experiment records the time needed for an algorithm to complete on an input schema. In order to collect more accurate results, each experiment is done for 1000 times to obtain an average time result with the same $\#A$ and same $\#C$ for schemas having $\#A < 200$, 100 times for schemas having $\#A \in [200, 400)$, and 10 times for $\#A \in [400, \infty)$. The reason to have less times of experiments for larger $\#A$ is that it takes minutes to hours for a single algorithm execution with large $\#A$, which makes it impractical to run 1000 times. We now report the findings.

The automata approach is exponentially more expensive than syntactic conditions

We compared the time needed for the automata and syntactic condition approaches on checking the same set of schemas that contain only ordering and alternating constraints. (For other three types of combinations of constraints, the results are similar). The input schemas have n activities and either n, $\frac{n}{2}$, or $\frac{2n}{3}$ constraints, where n ranges from 4 to 28. Fig. 8 shows the results (x-axis denotes the number of activities and y-axis denotes the time needed in the log scale). It can be observed that for the automata approach, the time needed is growing exponentially wrt the number of activities/constraints. For a schema with 28 activities and 28 constraints, it takes more than 3 hours to finish the checking. However, the syntactic condition approaches (whose complexity is polynomial) can finish the conformance checking almost instantly. As the times needed for either n, $\frac{n}{2}$, or $\frac{2n}{3}$ constraints are all too close around 1ms, we only use one curve (instead of three) in Fig. 8 to represent the result for the syntactic conditions approach.

The syntactic conditions approaches have at most a cubic growth rate in the size of the input schemas

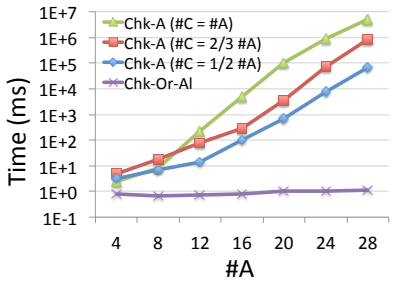

Figure 8: Automata vs Syn. Cond.

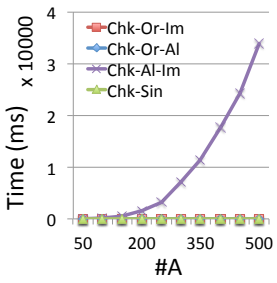

Figure 9: Scalability

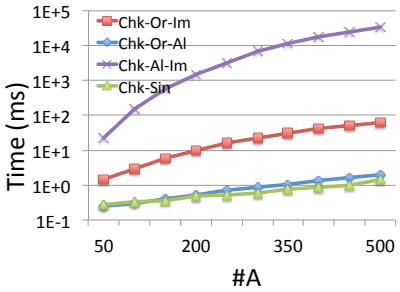

Figure 10: Scalability (log)

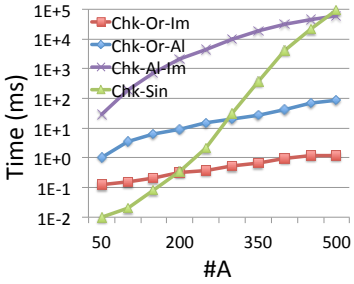

Figure 11: String Generation

We compute the times needed for the syntactic condition approaches for input schemas with n activities and n constraints, n between 50 and 500. Fig. 9 and 10 show the same result with normal and logarithm scales (resp.) of all four combinations of the constraints. From the result, the complexity of the syntactic condition approach for alternating and immediate constraints appears cubic due to the checking of Condition (4) of Definition 7.4.2 (collapsable); the complexity for ordering and immediate constraints is quadratic due to the pre-processing to form an im$^+$schema; the complexity for ordering and alternating constraints is linear as the pre-processing (to form an al$^+$schema by detecting strongly connected components) as well as the acyclicity check of the causality graphs are linear; finally, the complexity for the constraints of a single direction is also linear.

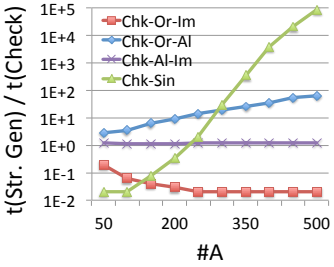

Figure 12: Str. Gen. / Checking

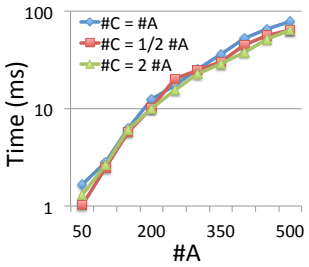

Figure 13: Changing #Constraints

Conforming string generation requires polynomial to exponential times

With the same experiment setting as above, Fig. 11 shows the time to generate a conforming string for a conformable schema. From the results, all string generating approaches are polynomial except for the single direction case (i.e, either response or precedence). According to Alg. 8, the length of a generated string can be as long as 2^n, where n is the number of activities in the given schema. Fig. 12 presents the ratios of the time to generate a conforming string over the time to check conformance of the same schema for conformable schemas. The results indicate that the complexity to generate a string can be polynomially lower (ordering and immediate case), the same (alternating and immediate case), polynomially higher (ordering and alternating case), and exponentially higher (single direction case) than the corresponding complexity to check conformance of the same schema. Note that the curves in Fig. 12 is lower or "smaller" than dividing "Fig. 11" by "Fig. 9" due to the reason that the data shown in Fig. 11 is only for the conformable schemas; while the one in Fig. 9 is for general schemas, where non-conformable schemas can be determined 5 - 15% faster than conformable ones due to the reason that a non-comformable schema fails the checking if it does not satisfy one of the conditions (e.g., in Theorem 7.3.9, there are three conditions to check); while a comformable schema can pass the check only after all conditions are checked.

Increasing the number of constraints increases more time for the automata

approach than syntactic condition approaches

We compute the time needed for the syntactic condition approaches with input schemas containing only ordering and immediate constraints with n activities and either n, $2n$, or $\frac{n}{2}$ constraints, where n ranges from 50 to 500. (For other three types of combinations of constraints, the results are similar). Fig. 13 shows the three curves for n, $2n$, and $\frac{n}{2}$ constraints respectively. Comparing the similar settings shown in Fig. 8, there does not exist an obvious growth in time when the number of constraints grow and the curves are almost the same. The reason is that the algorithms we used to check conformance and generate strings are graph-based approaches. As $\#C \in [\frac{\#A}{2}, 2\#A]$, we have $O(\#C) = O(\#A)$ that can provide the same complexity. Moreover, if $\#C < \frac{\#A}{2}$, there will be activities involving in no constraint, which leads to a non-practical setting; if $\#C > 2\#A$, almost all the randomly generated schemas will be non-confomable based on uniform distribution.

7.7 Summary

This chapter studied syntactic characterization of conformance for "core" DecSerFlow constraints that are reduced from general DecSerFlow constraints. We provided characterizations for (1) ordering and immediate constraints, (2) ordering and alternating constraints, (3) alternating and immediate constraints, and (4) ordering, alternating, and immediate constraints with precedence (or response) direction only. The general case for ordering, immediate, and alternating constraints with both precedence and response directions remains as an open problem; furthermore, it is unclear if the conformance problem for DecSerFlow constraints is in PTIME.

Chapter 8

Related Work

This chapter discusses the related works including business processes, schema mapping, and choreography.

8.1 Business Processes and Artifacts

Business process modeling has been studied variously in the last decade ([48, 66]). Traditional business process models are control-flow-centric, i.e., focusing extensively on activities and control flow that governs the ordering among activities. Examples of such models include BPMN [1], BPEL [50], and YAWL [36]. A promising trend is that the business process research and development communities are embracing a fundamental shift from control-flow-centric to data/artifact-centric process design and specification. The concept of business artifacts is introduced in [2]. In [67], the authors lay out the methodology in the context of Model Driven Business Transformation and describe the positive feedback received in real-world engagements. Nine patterns emerging in artifact-centric process models and develops a computational model based on Petri Nets were presented in [68]. Development of formal models was reported in [69, 12, 70]. Verifica-

tion of temporal properties concerning the workflow logic can be found in [69, 71, 72]. Static analysis of well-formedness was done in [12]. And finally, automated construction from non-temporal goals was shown possible for a restricted case [73].

The Guard-Stage-Milestone (GSM) paradigm is a declarative artifact-centric business process model based on the business artifact model originally introduced in [2, 74], but using a declarative basis [41, 15, 75].

There is a strong relationship between the GSM model and Case Management [76, 77]; both approaches focus on conceptual entities that evolve over time, and support *ad hoc* styles of managing activities. The GSM framework provides a formal operational semantics [41, 15, 75]. The core GSM constructs are being incorporated into the OMG Case Management Modeling Notation standard [78].

There is a loose correspondence between the artifact approach and proclets [79]; both approaches factor a BPM application into "bite-size" pieces that interact through time. Proclets do not emphasize the data aspect, and support only message-based proclet interaction. In addition to supporting messages, GSM permits interaction of artifact instances through condition testing and internal event triggering.

Concerning BPaaS, [80] proposes an anonymization-based approach to preserve the privacy of a public BP. Similarly, [81] focuses on how to hide the business logic of out-sourced GSM processes [15] while still providing the BP services to clients. In [39], the authors studied how auditing can be done for BPaaS, our SeGA framework can easily solve their problem.

Various techniques and formal models are proposed for BP verification [82, 83]. They are only performed at the model level. Their applications to runtime analysis therefore are very limited.

A generic solution for BP execution analysis with a process data warehouse model and ETL generation mechanism was presented in [84].

Providing process flexibility to support foreseen and unforeseen changes is a very active research area. [16] presented a novel and functional mechanism to handle ad hoc and just-in-time changes at runtime.

8.2 Schema Mapping and Relational Database

Schema mapping techniques are used in schema/data integration and data exchange between different schemas. The focus there is to reason about and query generated target instance(s) through mapping rules and source instance(s). A classic representative is Clio [6, 7]. In our entity-database mappings, a relational schema is associated with a hierarchical business entity. The purpose of the mappings is to facilitate data access in business processes, e.g., automated code generation. Updatability turns out to be crucial for maintain data consistency and connectivity between business processes and the enterprise database. Updatability was not studied in conjunction with schema mappings, and updatability is not always possible for schema mapping rules. Since our mapping rules corresponds to a special subset of Clio mappings, updatability results generalize to these Clio mappings.

[85] addresses a similar problem with update propagation given a mapping. However, the mapping languages used in [85] is far not expressive as tgds, considering [85] disallows joins.

The comparison of business entity updatability and view updates on relation database or XML ([3, 4, 5, 23]) has been addressed in Section 3.3. For database updatability, it is related to view self-maintenance for data warehouses with materialized views. Although not every data warehouse is self-maintainable in general [86], ED covers are always database updatable.

Schema/database integration is achieved through a global schema and mappings be-

tween the global schema and local schemas [87]. Earlier work used the relational setting and focuses on query answering [87]. Our ED rules differ from both GAV and LAV and focus on updates.

The techniques of recovering the source database given a schema mapping (specified by tgds) is similar to the ones used for inverting schema mapping [27, 28, 29]. The inverting schema mapping focuses on recovering the "most proper" source given a target, considering that the semantics in those papers always allow an arbitrary target to be some solution; while our target is only a solution when it is chased.

8.3 Choreography and Satisfiability

The choreography approach to modeling and analysis business processes or service interactions has been studied for a decade. A survey for formal models and results is provided in [62].

A number of standards have been proposed to address issues related to service-based process collaboration management. BPEL [50] together with WS-C [88] and WS-T (that includes Atomic Transaction [89] and Business Activity [90]) provides some basic support for coordination and exception handling. However none of these specifications is capable of or hints at querying runtime execution status and providing firsthand information for runtime monitoring and adjustment if necessary.

WS-CDL [57] is an XML based language for choreography, its choreography constraints message exchanges based on conditions that may involve information types, variables and tokens. Unfortunately, message contents need to be copied to variables to be used for choreography conditions. There is no data model for participant interface with a collaboration. Also, there is no direct support for multi-instances of a participant.

Recent work in [53, 59] extend BPEL to support choreographies with a bottom-

up approach to build a choreography from specified participant behaviors. BPEL4chor supports service interaction patterns, e.g., one-to-many send, one-from-many receive, one-to-many send/receive patterns through aggregation. Similar work on BPMN was in [91]. However, neither BPEL nor BPMN's extensions directly include data in their conceptual models and the instance level correlation support is much weaker than ours.

Let's Dance [43] provides a set of sequencing constraint primitives to allow a choreography to be specified in a graphical language. It lacks a clearly support for data or information models. Earlier work on conversations was reported in [51]. In the conversation model, BP systems collaborate with each other via generic asynchronous message exchanging. The information model in the conversation is limited.

Artifact-centric choreography [92] extends existing artifact-centric BP models with agents and locations. BPs can access artifacts from their locations with the help of agents. Petri-Net is used to specify artifact internal behaviors and external interactions. The model has no artifacts data attributes.

Process views have been used as an abstraction of BPs to support BP collaboration in [93, 94]. Various consistency rules are developed to make sure the derived process views are consistent with the BP models. Their approach supports design time BP coordination, but does not tackle the hard issue of run time management. [44] proposed a centralized artifact hub in coordinating business processes. Similarly [95] proposed a view mechanism for partners with artifact-centric BPs which support different views for different participants.

There have been work done on testing if a choreography is realizable. In [96, 97], a choreography is defined wrt a set of peers forming a collaboration. The notions of completed, partial and distributed realizability of choreography were defined and studied. It was shows that partial realizability is undecidable whereas distributed and complete realizability are decidable. [52, 58], focused on the realizability problem of global behavior

of interaction services. Sufficient conditions are given for realizability.

The constraint language studied for choreography is a part of DecSerFlow [54], and the constraints can be translated to Linear Temporal Logic (LTL) [9].

LTL employs infinite semantics, [98] first proved that LTL satisfiability checking is PSPACE-complete. A well-know result in [99] shows that LTL is equivalent to Büchi automata; and the LTL satisfiability checking can be translated to language emptiness checking. [100] shows the LTL with past operators is no more expressive than LTL.

Several complexity results upon satisfiability are given with respect to the different subsets of LTL. [101] shows that the restriction to Horn formulas will not decrease the complexity of LTL satisfiability checking. [102] investigates the complexity of cases restricted by the use of temporal operators, their nesting, and number of variables. [103] and [65] provide upper and lower bounds for different combinations of both temporal and propositional operators. [104] presents the tractability of LTL only with combination of "XOR" clauses.

For the finite semantics, [105] studies the semantics of LTL upon truncated (i.e., finite but not maximal) paths. [106] provides an exponential-time algorithm to check if a given LTL formula can be satisfied by a given finite-state model, but the execution is still infinite. Similarly, In [107], a linear-time algorithm is given to check if the given CTL formula can be satisfied by a given finite-state model. In [108], the authors studied the LTL over finte traces and proved that satisfiability, validity, and model checking under such semantics are all PSPACE-complete.

Chapter 9

Conclusions

This thesis studies the challenges in business process management (BPM). In details, we particularly focus on the data and collaboration management for BPM.

We initiate a study on data mappings between BPs and databases through formalizing the data models and formulating a mapping language. This idea of bridging BPs and databases has a potential to allow management issues to be dealt with separately for BPs and for databases while making sound design decisions. For BPM, it allows many interesting problems to be studied in the presence of data, e.g., process evolution. For databases, it brings a new dimension, i.e. BPs, into the database design, in particular, by including BPs' data needs, database design could avoid problems such as missing data or mismatched semantics.

On the technical front, there are several interesting problems to be addressed. A better understanding is needed for specifying entity-data mappings, alternative languages and relaxing the attribute-attribute mapping requirement are worth considering. As for one of the potential problems to solve, we adapt tuple generating dependency (tgd) as a mapping language to specify the mapping between BP data and enterprise database. tgd has been widely used in data exchange community, which also has a rather expressive

power. Unfortunately, the updatability checking falls into intractable cases for most of the tgd mappings specified, even with the present of keys and part of the source relations to be known. To further understand of how updatability can be checked efficiently under the tractable cases, immediate issues are to develop concrete algorithms for update propagation. Also, another angle to investigate updatability is to check the property incrementally, i.e., given both source and target instances, together with an update on target, to determine if there exists an update on source, s.t., the tgd mapping still holds.

The proposal of data mapping between BP data and enterprise database serves as a basis for separating BP data management and execution management. The separation enables business-processes-as-a-service (BPaaS). The demand for BPaaS is emerging while collaborative BPs remains a challenge. We have seen various vertical BPaaSs in for example HR and procurement. Clearly BPaaS is not just about providing APIs and interfaces for configuration and graphical analysis. The challenges lie in the capability to handle massive scaling, the service must be able to support multiple languages and execution environments, as well as massive customers and processes. We argue that the separation of the data from the execution engine is a good way to meet this demand. We demonstrate in the thesis that the SeGA framework provides a holistic approach in supporting this separation and result in a uniform way of facilitating different BP collaboration frameworks and supporting runtime analysis.

As a future work, the implementation of the SeGA framework can be challenging. The conceptual model of SeGA only addresses the problem of how to separate data and execution. However, in practice, how BP data and the related status information, schema, or correlation status can be wrapped into and unwrapped from a universal artifact efficiently needs to be further studied.

In terms of BP collaboration, we propose a declarative choreography language that can express correlations and choreographies for artifact-centric BPs in both type and

instance levels. It also incorporate data contents and cardinality on participant instances into choreography constraints. Furthermore, a subclass of the rule-based choreography is shown to be equivalent to a state-machine-based choreography.

As a follow-up study, we investigated syntactic characterization of conformance for the choreography language. we consider different combination of the constraint types and for each combination; syntactic conditions are provided to decide whether the given constraints are satisfiable. The syntactic conditions automatically lead to polynomial testing methods (comparing to PSPACE-complete complexity of general LTL satisfiability testing).

Bibliography

[1] "Business Process Model and Notation (BPMN), Version 2.0." `http://www.omg.org/spec/BPMN/2.0/PDF`, 2011.

[2] A. Nigam and N. S. Caswell, *Business Artifacts: An Approach to Operational Specification*, *IBM Systems Journal* **42** (2003), no. 3 428–445.

[3] F. Bancilhon and N. Spyratos, *Update Semantics of Relational Views*, *ACM Trans. Database Syst.* **6** (1981), no. 4 557–575.

[4] U. Dayal and P. A. Bernstein, *On the Correct Translation of Update Operations on Relational Views*, *ACM Trans. Database Syst.* **7** (1982), no. 3 381–416.

[5] J. Lechtenbörger, *The Impact of the Constant Complement Approach towards View Updating*, in *Proc. of the ACM Symposium on Principles of Database Systems*, 2003.

[6] R. Fagin et al, *Clio: Schema Mapping Creation and Data Exchange*, in *Conceptual Modeling: Foundations and Applications*, pp. 198–236, 2009.

[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, *Data exchange: Semantics and query answering*, in *ICDT*, pp. 207–224, 2003.

[8] C. Peltz, *Web Services Orchestration and Choreography*, *IEEE Computer* **36** (2003), no. 10 46–52.

[9] A. Pnueli, *The temporal logic of programs*, in *FOCS*, pp. 46–57, 1977.

[10] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, *CACM* **13** (1970), no. 6 377–387.

[11] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[12] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su, *Towards Formal Analysis of Artifact-Centric Business Process Models*, in *Proc. of the Int. Conf. on BPM*, 2007.

[13] P. P.-S. Chen, *The Entity-Relationship Model – Toward a Unified View of Data*, *ACM Trans. Database Syst.* **1** (Mar., 1976) 9–36.

[14] "Kingfore Corporation." `www.kingfore.net`.

[15] R. Hull et al, *Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events*, in *Proc. ACM Int. Conf. on DEBS*, 2011.

[16] W. Xu, J. Su, Z. Yan, J. Yang, and L. Zhang, *An Artifact-Centric Approach to Dynamic Modification of Workflow Execution*, in *Proc. Int. Conf. on CoopIS*, 2011.

[17] W. M. P. van der Aalst and A. H. M. ter Hofstede, *YAWL: yet another workflow language*, *Inf. Syst.* **30** (2005), no. 4 245–275.

[18] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu, *A Flexible, Object-Centric Approach for Business Process Modelling*, *Service Oriented Computing and Applications* **4** (2010), no. 3 191–201.

[19] V. Künzle and M. Reichert, *PHILharmonicFlows: Towards a Framework for Object-Aware Process Management*, *Journal of Software Maintenance* **23** (2011), no. 4 205–244.

[20] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, *Translating web data*, in *VLDB*, pp. 598–609, 2002.

[21] S. Abiteboul and N. Bidoit, *Non first normal form relations: An algebra allowing data restructuring*, *J. Comput. Syst. Sci.* **33** (1986), no. 3 361–393.

[22] A. K. Chandra and D. Harel, *Computable Queries for Relational Data Bases*, *JCSS* **21** (1980), no. 2 156–78.

[23] L. Wang, E. A. Rundensteiner, and M. Mani, *Updating xml views published over relational databases: Towards the existence of a correct update mapping*, *Data Knowl. Eng.* **58** (2006), no. 3 263–298.

[24] Y. Sun, W. Xu, J. Su, and J. Yang, *SeGA: A Mediator for Artifact-Centric Business Processes*, in *CoopIS*, pp. 658–661, 2012.

[25] R. Fagin, P. G. Kolaitis, and L. Popa, *Data exchange: Getting to the core*, in *Proceedings of the 22nd ACM Symposium on Principles of Database Systems (PODS)*, pp. 90–101, 2003.

[26] Y. Sun, J. Su, B. Wu, and J. Yang, *Modeling data for business processes*, in *IEEE 30th Intl. Conf. on Data Engineering (ICDE)*, pp. 1048–1059, 2014.

[27] R. Fagin, *Inverting schema mappings*, in *Proc. of the 25th ACM Symposium on Principles of Database Systems PODS*, pp. 50–59, 2006.

[28] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan, *Quasi-inverses of schema mappings*, in *Proc. of the 26th ACM Symposium on Principles of Database Systems PODS*, pp. 123–132, 2007.

[29] M. Arenas, J. Pérez, and C. Riveros, *The recovery of a schema mapping: Bringing exchanged data back*, in *Proc. of the 27th ACM Symposium on Principles of Database Systems, PODS*, pp. 13–22, 2008.

[30] G. Grahne, A. Moallemi, and A. Onet, *Recovering exchanged data*, in *Proc. of the 34th ACM Symposium on Principles of Database Systems, PODS*, 2015.

[31] S. Abiteboul and O. M. Duschka, *Complexity of answering queries using materialized views*, in *Proc. of the 17th ACM Symposium on Principles of Database Systems (PODS)*, pp. 254–263, 1998.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[33] A. C. Klug, *Calculating constraints on relational expressions*, *ACM Trans. Database Syst.* **5** (1980), no. 3 260–290.

[34] M. Kutz, *The complexity of boolean matrix root computation*, *Theor. Comput. Sci.* **325** (2004), no. 3 373–390.

[35] W. van der Aalst and K. van Hee, *Workflow Management: Models, methods and systems*. The MIT Press, 2004.

[36] W. van der Aalst and A. ter Hofstede, *YAWL: yet another workflow language*, *Information Systems* **30** (2005), no. 4 245–275.

[37] Z. Wu, S. Deng, and J. Wu, *Service Computing and Service Technology*. Zhejiang University Press, 2009.

[38] G. Group, "Gartner Newsroom." `http://www.gartner.com/it/page.jsp?id=1740414`, 2011.

[39] R. Accorsi, *Business Process as a Service: Chances for Remote Auditing*, in *COMPSAC Workshops*, pp. 398–403, 2011.

[40] T. Heath, D. Boaz, M. Gupta, R. Vaculín, Y. Sun, R. Hull, and L. Limonad, *Barcelona: A Design and Runtime Environment for Declarative Artifact-Centric BPM*, in *Proc. of the 11th Int. Conf. on Service-Oriented Computing*, ICSOC, pp. 705–709, 2013.

[41] E. Damaggio, R. Hull, and R. Vaculín, *On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard-Stage-Milestone Lifecycles*, in *Proc. of the 9th Int. Conf. on Business Process Management, BPM*, pp. 396–412, 2011.

[42] G. Liu, X. Liu, H. Qin, J. Su, Z. Yan, and L. Zhang, *Automated Realization of Business Workflow Specification*, in *Proc. of the ICSOC/ServiceWave Workshops*, pp. 96–108, 2009.

[43] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede, *Let's Dance: A Language for Service Behavior Modeling*, in *Proc. of the 14th Int. Conf. on Cooperative Information Systems, CoopIS*, pp. 145–162, 2006.

[44] R. Hull, N. C. Narendra, and A. Nigam, *Facilitating Workflow Interoperation Using Artifact-Centric Hubs*, in *Proc. of the 7th Int. Conf. on Service-Oriented Computing, ICSOC*, pp. 1–18, 2009.

[45] Y. Sun, W. Xu, and J. Su, *Declarative choreographies for artifacts*, in *ICSOC*, pp. 420–434, 2012.

[46] L. Limonad, D. Boaz, R. Hull, R. Vaculín, and F. T. Heath, *A Generic Business Artifacts Based Authorization Framework for Cross-Enterprise Collaboration*, in *SRII Global Conference*, pp. 70–79, 2012.

[47] R. Cattell and D. Barry, *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[48] R. Hull, J. Su, and R. Vaculín, *Data management perspectives on business process management: tutorial overview*, in *SIGMOD Conference*, pp. 943–948, 2013.

[49] R. Hull and J. Su, *Tools for composite web services: a short overview*, *SIGMOD Record* **34** (2005), no. 2 86–95.

[50] "Web Services Business Process Execution Language (BPEL), Version 2.0." `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, 2007.

[51] J. Hanson, P. Nandi, and S. Kumaran, *Conversation support for business process integration*, in *Enterprise Distributed Object Computing Conference, 2002. EDOC '02. Proceedings. Sixth International*, pp. 65 – 74, 2002.

[52] T. Bultan, X. Fu, R. Hull, and J. Su, *Conversation specification: a new approach to design and analysis of e-service composition*, in *Proc. of the 12th Int. Conf on World Wide Web, WWW*, pp. 403–410, 2003.

[53] G. Decker, O. Kopp, F. Leymann, and M. Weske, *BPEL4Chor: Extending BPEL for Modeling Choreographies*, in *Proc. of the 5th Int. Conf. on Web Services, ICWS*, 2007.

[54] W. M. P. van der Aalst and M. Pesic, *DecSerFlow: Towards a Truly Declarative Service Flow Language*, in *Proc. of the 3rd Int. Workshop on Web Services and Formal Methods, WS-FM*, pp. 1–23, 2006.

[55] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, *Choreography and orchestration conformance for system design*, in *Proc. 8th Int. Conf. on Coordination Models and Languages (COORDINATION)*, vol. 4038, (Bologna, Italy), pp. 63–81, Springer, June, 2006.

[56] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot, *A theoretical basis of communication-centred concurrent programming*, 2006.

[57] "Web Services Choreography Description Language (WS-CDL), Version 1.0." `http://www.w3.org/TR/ws-cdl-10/`, 2005.

[58] X. Fu, T. Bultan, and J. Su, *Conversation protocols: a formalism for specification and verification of reactive electronic services*, *Theor. Comput. Sci.* **328** (2004), no. 1-2.

[59] O. Kopp, L. Engler, T. Lessen, F. Leymann, and J. Nitzsche, *Interaction choreography models in bpel: Choreographies on the enterprise service bus*, in *Subject-Oriented Business Process Management* (A. Fleischmann, W. Schmidt, R. Singer, and D. Seese, eds.), vol. 138 of *Communications in Computer and Information Science*, pp. 36–53. Springer Berlin Heidelberg, 2011.

[60] X. Fu, T. Bultan, and J. Su, *Analysis of interacting bpel web services*, in *Proceedings of the 13th international conference on World Wide Web*, WWW '04, (New York, NY, USA), pp. 621–630, 2004.

[61] M. Vardi and P. Wolper, *Reasoning About Infinite Computations*, *Inf. Comput.* **115** (1994), no. 1.

[62] J. Su, T. Bultan, X. Fu, and X. Zhao, *Towards a Theory of Web Service Choreographies*, in *Proc. of the 4th Int. Workshop on Web Services and Formal Methods, WS-FM*, 2007.

[63] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers, *Imperative versus declarative process modeling languages: An empirical investigation*, in *Business Process Management Workshops (1)*, pp. 383–394, 2011.

[64] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, *Declare: Full support for loosely-structured processes*, in *EDOC*, pp. 287–300, 2007.

[65] A. Artale, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev, *The complexity of clausal fragments of ltl*, in *LPAR*, pp. 35–52, 2013.

[66] W. M. P. van der Aalst, *Business process management demystified: A tutorial on models, systems and standards for workflow management*, in *Lectures on Concurrency and Petri Nets*, pp. 1–65, 2003.

[67] K. Bhattacharya, R. Guttman, K. Lymann, F. Heath III, S. Kumaran, P. Nandi, F. Wu, P. Athma, C. Freiberg, L. Johannsen, and A. Staudt, *A model-driven approach to industrializing discovery processes in pharmaceutical research*, *IBM Systems Journal* **44** (2005), no. 1 145–162.

[68] R. Liu, K. Bhattacharya, and F. Y. Wu, *Modeling business contexture and behavior using business artifacts*, in *CAiSE*, vol. 4495 of *LNCS*, 2007.

[69] C. E. Gerede, K. Bhattacharya, and J. Su, *Static analysis of business artifact-centric operational models*, in *IEEE Int. Conf. on Service-Oriented Computing and Applications*, 2007.

[70] S. Abiteboul, L. Segoufin, and V. Vianu, *Modeling and verifying active xml artifacts*, *Data Engineering Bulletin* **32** (15, 2009) 10.

[71] C. E. Gerede and J. Su, *Specification and verification of artifact behaviors in business process models*, in *Proceedings of 5th International Conference on Service-Oriented Computing (ICSOC)*, (Vienna, Austria), September, 2007.

[72] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu, *Automatic verification of data-centric business processes*, in *Proc. Int. Conf. on Database Theory (ICDT)*, pp. 252–267, 2009.

[73] C. Fritz, R. Hull, and J. Su, *Automatic construction of simple artifact-based business processes*, in *Proc. Int. Conf. on Database Theory (ICDT)*, 2009.

[74] S. Kumaran, P. Nandi, F. F. T. H. III, K. Bhaskaran, and R. Das, *Adoc-oriented programming*, in *SAINT*, pp. 334–343, 2003.

[75] Y. Sun, R. Hull, and R. Vaculín, *Parallel processing for business artifacts with declarative lifecycles*, in *On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I*, pp. 433–443, 2012.

[76] W.-D. Zhu and et al, "Advanced Case Management with IBM Case Manager." `http://www.redbooks.ibm.com/redpieces/abstracts/sg247929.html?Open`.

[77] W. M. P. van der Aalst and M. Weske, *Case handling: a new paradigm for business process support*, *Data Knowl. Eng.* **53** (May, 2005) 129–162.

[78] "Case Management Model and Notation 1.0." `http://www.omg.org/spec/CMMN/1.0/`, 2014.

[79] W. M. P. van der Aalst *et. al.*, *Proclets: A framework for lightweight interacting workflow processes*, *Int. J. Cooperative Inf. Syst.* (2001) 443–481.

[80] M. Bentounsi, S. Benbernou, C. S. Deme, and M. J. Atallah, *Anonyfrag: an anonymization-based approach for privacy-preserving BPaaS*, in *Cloud-I*, p. 9, 2012.

[81] R. Eshuis, R. Hull, Y. Sun, and R. Vaculín, *Splitting GSM Schemas: A Framework for Outsourcing of Declarative Artifact Systems*, in *BPM*, pp. 259–274, 2013.

[82] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg, *Analyzing interacting ws-bpel processes using flexible model generation*, *Data Knowl. Eng.* **64** (2008), no. 1 38–54.

[83] S. Nakajima, *Model-checking behavioral specification of bpel applications*, *Electr. Notes Theor. Comput. Sci.* **151** (2006), no. 2 89–105.

[84] F. Casati, M. Castellanos, U. Dayal, and N. Salazar, *A generic solution for warehousing business process data*, in *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pp. 1128–1137, 2007.

[85] S. Melnik, A. Adya, and P. A. Bernstein, *Compiling mappings to bridge applications and databases*, *ACM Trans. Database Syst.* **33** (2008), no. 4.

[86] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, *Maintaining views incrementally*, in *Proc. ACM SIGMOD Management of Data*, pp. 157–166, 1993.

[87] M. Lenzerini, *Data integration: A theoretical perspective*, in *PODS*, pp. 233–246, 2002.

[88] "Web services coordination 1.1 (WS-coordination)." `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-pr-01.pdf`, 2006.

[89] "Web services atomic transaction (WS-atomic transaction)." `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-pr-01.pdf`, 2006.

[90] "Web services business activity (WS-business activity)." `http://docs.oasis-open.org/ws-tx/wstxwsba-1.1-spec-pr-01.pdf`, 2006.

[91] K. Pfitzner, G. Decker, O. Kopp, and F. Leymann, *Web service choreography configurations for bpmn*, in *Service-Oriented Computing - ICSOC 2007 Workshops*, vol. 4907, pp. 401–412. 2009.

[92] N. Lohmann and K. Wolf, *Artifact-centric choreographies*, in *Service-Oriented Computing*, vol. 6470, pp. 32–46. 2010.

[93] D.-R. Liu and M. Shen, *Business-to-business workflow interoperation based on process-views*, *Decision Support Systems* **38** (2004), no. 3 399–419.

[94] R. Eshuis and P. Grefen, *Constructing customized process views*, *Data Knowl. Eng.* **64** (Feb., 2008) 419–438.

[95] S. Yongchareon and C. Liu, *A process view framework for artifact-centric business processes*, in *Proc. of the 19th International Conference on Cooperative Information Systems*, CoopIS '11, pp. 26–43, 2010.

[96] N. Lohmann and K. Wolf, *Realizability is controllability*, in *Web Services and Formal Methods*, vol. 6194 of *Lecture Notes in Computer Science*, pp. 110–127. 2010.

[97] N. Lohmann and K. Wolf, *Decidability results for choreography realization*, in *Service-Oriented Computing*, vol. 7084 of *Lecture Notes in Computer Science*, pp. 92–107. 2011.

[98] A. P. Sistla and E. M. Clarke, *The complexity of propositional linear temporal logics*, *J. ACM* **32** (1985), no. 3 733–749.

[99] M. Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification (preliminary report)*, in *LICS*, pp. 332–344, 1986.

[100] N. Markey, *Past is for free: on the complexity of verifying linear temporal properties with past*, *Acta Inf.* **40** (2004), no. 6-7 431–458.

[101] C.-C. Chen and I.-P. Lin, *The computational complexity of satisfiability of temporal horn formulas in propositional linear-time temporal logic*, *Inf. Process. Lett.* **45** (1993), no. 3 131–136.

[102] S. Demri and P. Schnoebelen, *The complexity of propositional linear temporal logics in simple cases*, *Information and Computation* **174** (1998) 61–72.

[103] M. Bauland, T. Schneider, H. Schnoor, I. Schnoor, and H. Vollmer, *The complexity of generalized satisfiability for linear temporal logic*, in *FoSSaCS*, pp. 48–62, 2007.

[104] C. Dixon, M. Fisher, and B. Konev, *Tractable temporal reasoning*, in *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, AAAI Press, 2007.

[105] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout, *Reasoning with temporal logic on truncated paths*, in *CAV*, pp. 27–39, 2003.

[106] O. Lichtenstein and A. Pnueli, *Checking that finite state concurrent programs satisfy their linear specification*, in *POPL*, pp. 97–107, 1985.

[107] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, *ACM Transactions on Programming Languages and Systems* **8** (1986) 244–263.

[108] G. De Giacomo and M. Y. Vardi, *Linear temporal logic and linear dynamic logic on finite traces*, in *Proc. of the 23rd Intl. Joint Conf. on Artificial Intelligence IJCAI*, 2013.