Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 233–249. http://www.scpe.org
ISSN 1895-1767

APPROACHES TO AGGREGATE PRICE MODELS TO ENABLE COMPOSITE SERVICES ON ELECTRONIC MARKETPLACES *

FREDERIC JUNKER† JÜRGEN VOGEL‡ AND KATARINA STANOEVSKA§

Abstract. Marketplaces for cloud services, referred to as electronic marketplaces (eMPs), have been introduced for a number of purposes, including comparability and transparency of service offerings. Since the complexity of services and their provisioning is increasing, eMPs need to handle the dynamic and automated creation of composite services, i.e. services which are yielded in part by using third-party services available on the marketplace. To meet this requirement, eMPs need a standardized machine readable description of service offerings. This paper presents an approach to define price models for services traded on eMPs, using a subset of the Unified Service Description Language (USDL). To enable composite service pricing, this paper introduces an approach on the aggregation of several price models into one price model and analyses the time complexity of the price aggregation algorithm presented.

Key words: price model, USDL, composite service, price aggregation

AMS subject classifications. 68-04 68U35

1. Introduction. An eMP is a place on the Internet where services are offered by *service providers* and consumed by *service consumers*. A number of eMPs with varying functionalities and target audiences exist today, including Microsoft Windows Azure Marketplace, Google Apps Marketplace, AppExchange (salesforce.com), Android Marketplace, SuiteApp.com and Zoho Marketplace [1].

While most applications and services traded on those eMPs are small, monolithic and gadget-style, deploying applications and services on a large scale will lead to an increasing complexity of the underlying software and infrastructure. Accordingly, furthering the pervasion of eMPs requires research on the following topics:

1. Composite services, i.e. services which are provisioned in part by employing third-party services available on the marketplace.
2. Complex price models allowing the service provider of a composite service to cover the cost incurred by using third-party services.
3. Integrated service pricing, i.e. storing technical and pricing aspects together in a machine readable format.

Each of the third-party services on which a composite product relies has its own price model. Still, the composite product shall be analyzed, offered, used and charged like an elementary product from the user's perspective. So far, the structural design and the quantification of price models for services are chosen manually, without support from the marketplace the product is traded on. However, in case of a complex composite product using hundreds of third-party services, this approach does not scale well (1.) due to the large number of combinations of price models to be evaluated, and (2.) due to the tedious integration of price models with heterogeneous structural design.

1.1. Example: Composite Service. One example of a composite service is a cell phone service, which is visually displayed in cf. Fig. 1.1. When providing the service S, the cell phone service provider has to pay for using $S_1, ..., S_5$, each of which has its own independent price model:

1. Price model of S_1: \$3.000.000 per month for the right to erect cell towers on 3rd party land property.
2. Price model of S_2: \$0.1 per call minute for transmitting calls via the 3rd-party cable network operator's resources (connection-oriented service).
3. Price model of S_3: \$0.05 per text message for transmitting text messages via the 3rd-party cable network operator's resources (connectionless service).
4. Price model of S_4: (i.) \$0.05 per text message for saving the log of text messages in the 3rd-party database. Applies only for the first 5.000.000 text messages per month. (ii.) \$1.000.000 per month for using the database in the first place.

*This paper bases on the paper 'Aggregating Price Models for Composite Services in Cloud Service Marketplaces' presented at the C4BB4C Workshop at the 10th IEEE ISPA, Madrid, July 2012, http://www.dsa-research.org/c4bb4c/

†University of St. Gallen and SAP Research St. Gallen, Blumenbergplatz 9, CH-9000 St. Gallen, Switzerland, frederic.junker@unisg.ch.

‡SAP Research Zürich, Regensdorf/Switzerland, juergen.vogel@sap.com.

§University of St. Gallen, St. Gallen/Switzerland, katarina.stanoevska@unisg.ch.

5. Price model of S_5: \$4 per month staff costs.

Before offering the cell phone service on the marketplace, the provider needs to determine the service's price model. The price model shall be defined such that the cell phone service provider can pay for using the services $S_1, ..., S_5$. Accordingly, the service provider needs to join the price models into one single price model which constitutes the cost incurred by using 3rd-party services[1].

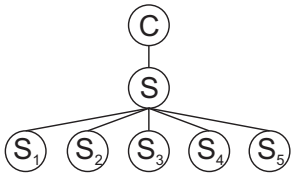

Fig. 1.1: The *product graph* of the example cell phone service.

1.2. Machine Readable Price Models. The Web Services Description Language (WSDL) "only target[s] the description of technical characteristics of services" [2]. In contrast, the Unified Service Description Language (USDL) is an effort to complement WSDL as a "specification language to describe services from a business, operational and technical perspective. USDL plays a major role in the Internet of Services to describe tradable services which are advertised in [eMPs]" [2].

1.3. Research Contribution. This paper presents an approach to define price models using a subset of USDL, including the computation of the *payment*, i.e. the actual amount of money charged to the service consumer, given the service's price model. To support composite service pricing, this paper presents two algorithms for the aggregation of several price models into one price model: aggressive and gentle deinterleaving. We prove both aggregation algorithms have a complexity of $O(n \cdot log(n))$. Aggressive and gentle deinterleaving have individual advantages and shortcomings, and the choice needs to be made based on application requirements.

The remainder of this paper is organized as follows: Sect. 2 presents related work and outlines the gap this paper intends to fill. Sect. 3 defines terms used in this paper as well as the structure of a price model. Sect. 4 describes how to compute the *payment*. Sect. 5 presents and analyses approaches to aggregate the price models of several services.

2. Related Work. Aggregating services is well understood from both technological and business perspectives. [4] analyses the aggregation of web services on a technical level. Research also exists on aggregating Service Level Agreements (SLAs) across several services [5] and aggregating services in business networks [6].

From a business perspective, [7] and [8] examine "determining the optimal composition [of business services] in terms of its short- and long-term profitability" and pricing models for composite mobile services. Beyond, [9] analyses usage metering for e-services, leading to an e-service taxonomy.

Eurich et al [3] research on revenue streams and business model innovation of cloud-based platforms, as well as "the design of the commercialization of PaaS solutions". The revenue streams of several real-world cloud-based platforms are discussed. However, approaches to quantified aggregation of price models and/or revenue streams are not presented. Wang [12] presents an approach towards profit-optimizing selection of third-party services in service value networks. The approach bases on brokers, i.e. instances mediating between customers and providers, and is therefore to be distinguished from our approach. However, the broker has some common features with electronic marketplaces mentioned in this paper, such as charging customers and awarding payments to providers. Another approach towards cost-based selection of third-party services is presented by [16]. An auction system is used to select third-party services to minimize the cost a service provider incurs, while keeping the computational complexity of identifying optimal third-party services low. In contrast, our approach assumes the third-party services have already been selected, and intends to minimize the complexity of the resulting aggregate price model that denotes the cost incurred by using all third-party services. Therefore, [16] and this paper are different in objective, and the two approaches may well be combined to realize efficient automation of service composition, pricing and billing on electronic marketplaces.

[1] Then, the service provider uses marketing approaches to define the price model of service S. This step is covered by existing marketing research.

Kantere et al [13] present an approach to optimally price 'data management services' offered via the cloud. Their approach relies on an economic model that computes the optimal price for cloud cache/data management services based on demand. In contrast, our research work 1) attempts to structure and handle price models for any kind of service offered in the cloud, and 2) deals with service value networks in which composite services rely on third-party services. The problem statement is therefore merely distantly related.

Literature from microeconomics, marketing and pricing, such as Nagle et al [10] and Homburg [11], deal with determining prices and price models under the supply and demand conditions of a free market. Such literature generally aims to determine prices and price models to maximize the profit the provider of a product or service makes. In contrast, we intend to determine a minimum price or price model, which is able to cover the cost a service provider incurs by using third-party services to offer the service.

Approaches from cost-earnings accounting compute indexes and ratios based on costs and earnings realized in the past. In contrast, our approach is based in investment and strategy, i.e. which decision needs to be made to generate certain costs and earnings. SLAs generally determine penalties the provider of a service pays to the customer if a given service level is not met. However, our approach merely determines a price model for a service, i.e. the payment the customer makes to the provider in return for using the service. Service levels and penalties are not considered in this paper, and may be added in future work. Due to the fundamental differences between those fields and our approach, no specific literature from those fields has been included in this overview of related work.

Existing work outlines requirements for the machine readable definition of price models of electronic services and introduces USDL (Unified Service Description Language) [2]. However, the automated aggregation of such machine readable price models has not been examined yet.

3. Price Models. The *price model* defines how much the consumer is charged for consuming a service. The service provider determines the price model before offering the service on the marketplace.

In contrast to the price model, the *payment* is the actual amount of money charged to the service consumer. The *payment* is computed by the payment calculation method described in Sect. 4.

To illustrate the definition of price models presented in this section, we pick the example cell phone service provider who defines the following price model for its service S, as seen in cf. Fig. 1.1: The consumer pays \$10 per month as a basic fee, \$0.1 per call minute and \$0.1 per text message. In case of more than 50 text messages a month, the price drops to \$0.05 per text message. Additionally, the actual cost charged to the consumer shall not be higher than \$30 each month. In the following sections, we use a subset of USDL (presented in [2]) to transform this textual representation of the price model into a machine readable definition. Also, the price model can be modelled as an ontology.

3.1. Billing Unit. The *billing unit* is the unit per which the consumer is charged ("per what"). For example, the consumer can subscribe to a service and can be charged per the billing units *Day*, *Week*, *Month*, *Quarter* or *Year*.

3.2. Payment Assessment Metric (PAM). A PAM determines the basis by which the consumer gets charged for using a service (i.e. "for what"). Each PAM possesses one or several billing units per which the consumer can be charged. Table 3.1 presents several PAMs and their associated billing units. This list is not exhaustive and can be amended in future work.

3.3. Price Model Component. A price model component (PMC) is a linear function which maps a number of consumed billing units to a *payment*. A PMC specifies a *price* which applies per billing unit, as well as the billing unit itself and a PAM. The term *price* is not to be confused with *price model* and *payment*. Beyond, a PMC applies only during the validity time frame, determined by two dates named t_{VFrom} and t_{VTo}, and within the price fence, which is a range of the number of billing units consumed, represented by the integers u_{Min} and u_{Max}.

Formal definition: A PMC is a 4-tuple *c = (pam, bu, price, condition)*:

- *pam*: A payment assessment metric.
- *bu*: One of the billing units associated with the PAM *pam*.
- *price*: The price charged by the PMC per billing unit.
- *condition* $= (t_{VFrom} \leq t \leq t_{VTo}) \wedge (u_{Min} \leq u \leq u_{Max})$: A logical expression determining whether the PMC is valid for given values of the variables t and u. t denotes the time, u denotes the consumed units, i.e. the number of billing units the consumer has consumed. $t_{VFrom}, t_{VTo} \in \mathbb{N}_0 \cup \{\infty\} =$

PAM	Description	Billing Units
Subscription	The consumer pays for the time frame during which the product can be used. The subscription fee applies independently from the quantitative consumption of the service within the subscribed time frame.	Day Week Month Quarter Year
Pay Per Use Event	The consumer pays for each event of interaction with the service.	Invocation Notification Transaction Session
Pay Per Use Time	The consumer pays for the time of actual interaction with the service.	Millisecond Second Hour Day Week
Pay Per Use Quantity	The consumer pays for the quantity of resources consumed by interacting with the service.	Kilobyte Megabyte Gigabyte
Licence	The consumer makes a one-time payment, which entitles to consumption of the service without limitation in time.	Licence
Admission	A one-time payment the service consumer makes before using the service for the first time. In contrast to the PAM *licence*, the consumer will have to make further payments over time.	Admission

Table 3.1: PAMs and their associated billing units.

$\{0, 1, 2, ..., \infty\}$ specify the PMC's validity time frame. These integers can be interpreted as points of time. $u_{Min}, u_{Max} \in \mathbb{N} \cup \{\infty\} = \{1, 2, ..., \infty\}$ determine the upper and lower bound of the price fence; u_{Max} and t_{VTo} can take the value ∞ to denote the absence of an upper bound.

3.4. Price Model. A price model contains:

- A set of *price model components*. The example price model mentioned above has 4 price model components: One for the \$10 monthly basic fee, one for the \$0.1 per call minute, one for the \$0.1 per text message for the first 50 text messages, and one for the \$0.05 per text message if above 50 text messages. The payment generated by the price model equals the sum of the payments generated by each of its PMCs.
- A *payment limit*. If the *payment* is higher than the *payment limit*, only the amount equal to the *payment limit* is charged to the consumer. The semantic function of the payment limit is to allow the service provider to equip the price model with a cost control function. The example price model mentioned above has the payment limit \$30, assuming that the payment calculation (see Sect. 4) and billing is run on a monthly basis.

Formal definition: A price model is a couple $P = (C, paymentLimit)$:

- A set of price model components $C = \{c_1, ..., c_n\}$. $C = \emptyset$ means the service is offered for free.
- A payment limit, referred to as $paymentLimit$.

The cell phone price model presented above is translated into the following structured machine-readable price model in USDL format: $P = (\$30, \{A, B, C, D\})$, with the PMCs A, B, C, D presented in cf. Table 3.2.

4. Payment Calculation. This section demonstrates how to compute the *payment*, i.e. the concrete amount of money charged to the consumer of a service, given the service's price model and the consumed units, i.e. the number of billing units consumed by the service consumer. The payment calculation method is required by eMPs, e.g. for billing and price-based product search.

PMC	A	B	C	D
PAM	Subscription	PPUTime	PPUEvent	PPUEvent
billing unit	month	minute	transaction	transaction
price	\$10	\$0.1	\$0.1	\$0.05
t_{VFrom}	0	0	0	0
t_{VTo}	∞	∞	∞	∞
u_{Min}	1	1	1	51
u_{Max}	∞	∞	50	∞

Table 3.2: Cell phone price model translated into structured USDL format.

The *payment* generated by the price model is the sum of the payments generated by each of the price model's PMCs. The payment generated by a PMC is the PMC's *price* multiplied by the PMC's *applying units*. Sect. 4.1 demonstrates how to compute the applying units, given the PMC and the number of consumed units. Additionally, as mentioned in the previous section, the price model contains a *payment limit*. Then, the *payment* is computed as follows:

$$payment = min\left(\sum_{i=1}^{n}(c_i.price * a_i), paymentLimit\right)$$

$c_1, ..., c_n$: PMCs of price model
a_i: applying units of PMC c_i
$paymentLimit$: payment limit of price model.

4.1. Applying Units. The concept of *applying units* allows the service provider to define fine-grained price models, including discounts and time-based changes in pricing. $t_{UFrom}, t_{UTo} \in \mathbb{N}_0$ denote the time frame during which the service was consumed. $v \in \mathbb{N}$ denotes the number of consumed units. The number of applying units a of a PMC is determined by checking the fulfilment of the PMC's condition for all possible value combinations of the variables t and u. The number of applying units equals the number of consumed units multiplied by the ratio between (see cf. Fig. 4.1):

1. the number of value combinations for which the PMC's condition evaluates to *true*, and
2. the total number of value combinations of $t \in D_t = [t_{UFrom}, t_{UTo}]$ and $u \in D_u = [1, v]$.

This definition yields the following formula:

$$a = v * \frac{|(t,u)|}{|D_t \times D_u|} : \quad condition = true$$

where

$$condition = (t_{VFrom} \leq t \leq t_{VTo}) \wedge (u_{Min} \leq u \leq u_{Max}).$$

4.1.1. PAM *Subscription*. In case of the PAM Subscription, the quantitative consumption of the service is specified by the two dates t_{UFrom} and t_{UTo}. The number of consumed units v is the length of the time span $(max(t_{VFrom}, t_{UFrom}), min(t_{VTo}, t_{UTo}))$ measured in billing units. If v is not an integer, it is rounded to the smallest larger integer. Accordingly, the definition presented above decomposes to[2]:

$$a = v \cdot \frac{|[t_{UFrom}, t_{UTo}] \cap [t_{VFrom}, t_{VTo}]|}{|[t_{UFrom}, t_{UTo}]|} \cdot \frac{|[1, v] \cap [u_{Min}, u_{Max}]|}{|[1, v]|}$$

[2] $[x, y]$ denotes the set of numbers between x and y, so $|[x, y]| = y - x + 1$

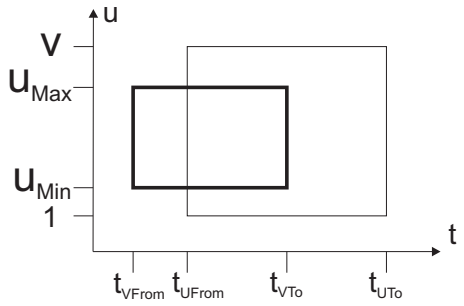

Fig. 4.1: A PMC depicted as a rectangle (with thick lines) in Euclidean space, with the axes being time and quantity of service consumption. The rectangle with thin lines represents the actual consumption by the service consumer. The intersection of the two rectangles equals the set of all combinations for the variables t and u for which the PMC's condition evaluates to *true*.

$$= v \cdot \frac{|[t_{UFrom}, t_{UTo}] \cap [t_{VFrom}, t_{VTo}]|}{(t_{UTo} - t_{UFrom} + 1)} \cdot \frac{|[1, v] \cap [u_{Min}, u_{Max}]|}{v - 1 + 1}$$

$$= \frac{max(0, min(t_{UTo}, t_{VTo}) - max(t_{UFrom}, t_{VFrom}) + 1)}{t_{UTo} - t_{UFrom} + 1} \cdot$$

$$max(0, min(u_{Max} - u_{Min} + 1, v - u_{Min} + 1))$$

4.1.2. PAM other than *Subscription*. In the case of any other PAM than *Subscription*, v is directly specified as a float number which determines the number of billing units consumed within the validity time frame. In this case, $t_{UFrom} = t_{VFrom}$ and $t_{UTo} = t_{VTo}$. Accordingly, the formula for computing a reduces to:

$$a = max(0, min(u_{Max} - u_{Min} + 1, v - u_{Min} + 1))$$

As an example, cf. Fig. 4.2 shows the payment generated by the two PMCs representing the text messages.

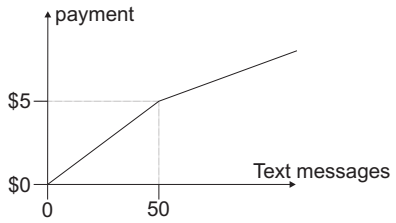

Fig. 4.2: Payment generated by the cell phone price model's PMCs C and D, depending on the number of text messages sent by the service consumer. The quantity discount curve needs to be completed by the customer each time he/she is billed.

4.2. Example. In our cell phone example, the consumer is billed each month. Assume he/she had subscribed to the cell phone service for the entire month 0, and during this time he/she made calls with a total duration of 100 minutes and sent 200 text messages. The *payments* generated by each PMC are presented in cf. Table 4.1[3]. According to the formula presented in Sect. 4, the payment for the entire price model is computed as follows: $payment = min(\$10 + \$10 + \$5 + \$7.5, \$30) = \mathbf{\$30}$.

[3]The numbers v, u_{Min}, u_{Max} and a are measured in billing units. The *price* applies per billing unit (see Sect. 3.3)

PMC	A	B	C	D
PAM	Subscription	PPUTime	PPUEvent	PPUEvent
billing unit	month	minute	transaction	transaction
t_{VFrom}	0	0	0	0
t_{VTo}	∞	∞	∞	∞
t_{UFrom}	0	n.a.	n.a.	n.a.
t_{UTo}	0	n.a.	n.a.	n.a.
v	1	100	200	200
u_{Min}	1	1	1	51
u_{Max}	∞	∞	50	∞
a	1	100	50	150
price	\$10	\$0.1	\$0.1	\$0.05
payment	**\$10**	**\$10**	**\$5**	**\$7.5**

Table 4.1: Payments generated on behalf of each PMC of the example price model.

5. Price Aggregation. Innovative electronic application marketplaces feature the trading of composite services, which are services yielded in part by using third-party services available on the marketplace. For example, the cell phone service S described in Sect. 1.1 is provided using the 3rd-party services $S_1, ..., S_5$, resulting in the *product graph* shown in cf. Fig. 1.1.

According to the requirements of modern eMPs outlined in Sect. 1, this section presents an approach to aggregate several price models which are defined according to Sect. 3. The straightforward way is to add each PMC of each price model as-is into the resulting aggregate price model. However, a composite product in a real cloud service scenario can consist of hundreds of services, yielding thousands or more PMCs to be aggregated. To accelerate the processes within the marketplace, e.g. cost calculations during product selection, this section presents algorithms minimizing the number of PMCs during price aggregation.

5.1. Terminology and Elementary Aggregation Operations. We refer to the aggregation of a set of PMCs $C_X = \{c_1, ..., c_n\}$ as the computation of a set of PMCs C_Y such that C_X and C_Y are *equivalent*, i.e. C_X and C_Y generate the same *payment*, regardless of the number and temporal occurrence of the consumed units[4]. Aggregation is accomplished by the following procedure:

1. Separate remaining PMCs into groups of same PAM and billing unit.
2. For each group, perform *deinterleaving*, either aggressive or gentle (Sections 5.5 and 5.6).
3. For each group, perform *merging* (Section 5.4).

We refer to *deinterleaving* of a set of PMCs C_X as the process of converting C_X into another set of PMCs C_Y such that C_Y shall contain as few PMCs as possible, and shall contain no overlapping PMCs if possible. We will see that these goals can conflict, and we will present different operations appropriate for different goal priorities.

5.1.1. Merging Horizontally Adjacent PMCs. Two PMCs c_1 and c_2 are *horizontally adjacent* if $c_1.t_{VTo} = c_2.t_{VFrom}$ or vice versa. Merging replaces two horizontally adjacent PMCs by one PMC whose validity time frame is the union of the validity time frames of the two adjacent PMCs (see cf. Fig. 5.1). Horizontally adjacent PMCs can be merged if and only if (1.) they have the same PAM, billing unit and price, and (2.) their price fence has the lower bound zero and no upper bound, i.e. $u_{min} = 0$ and $u_{max} = \infty$.

If the second condition does not hold (i.e. at least one of the PMCs' price fence is not $u_{min} = 0$ and $u_{max} = \infty$), then merging is not possible, even if the first condition holds and all PMCs have the same price fence (see cf. Fig. 5.2). A simple calculation shows that the two sets of PMCs in cf. Fig. 5.2 are not *equivalent*, i.e. they can generate different *payments* for some numbers of consumed units within the time frames (x_1, x_2) and (x_2, x_3).

[4]Thereby, it is ensured the service provider can pay for the 3rd-party services used.

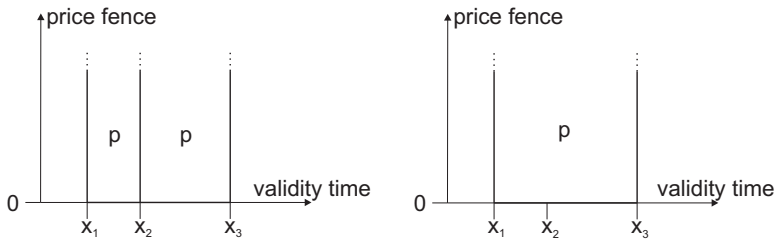

Fig. 5.1: Merging of horizontally adjacent PMCs (before and after).

Fig. 5.2: **Not possible**: Merging of horizontally adjacent PMCs.

5.1.2. Deinterleaving Horizontally Overlapping PMCs. Two PMCs c_1 and c_2 *overlap horizontally* if $c_2.t_{VFrom} < c_1.t_{VTo}$ and $c_2.t_{VTo} > c_1.t_{VFrom}$ or vice versa. Deinterleaving replaces two horizontally overlapping PMCs by three PMCs (see cf. Fig. 5.3). Horizontally overlapping PMCs can be deinterleaved if and only if (1.) they have the same PAM and billing unit, and (2.) their price fence has the lower bound zero and no upper bound, i.e. $u_{min} = 0$ and $u_{max} = \infty$. As in horizontal merging, if the second condition does not hold, then deinterleaving is not possible, even if the first condition holds and all input PMCs have the same price fence.

Fig. 5.3: Deinterleaving of horizontally overlapping PMCs (before and after).

5.1.3. Merging Vertically Adjacent PMCs. Two PMCs c_1 and c_2 are *vertically adjacent* if $c_1.u_{min} = c_2.u_{max}$ or vice versa. Vertical merging replaces two vertically adjacent PMCs by one PMC whose price fence is the union of the their price fences (see cf. Fig. 5.4). Adjacent PMCs can be merged if (1.) they have the same PAM, billing unit and price, and (2.) they have the same validity time frame.

5.1.4. Deinterleaving Vertically Adjacent PMCs. Two PMCs c_1 and c_2 *overlap vertically* if $c_2.u_{min} < c_1.u_{max}$ and $c_2.u_{max} > c_1.u_{min}$ or vice versa. Vertically overlapping PMCs can be deinterleaved if and only if (1.) they have the same PAM and billing unit. Due to the analogy to the previous operations, this section does not contain a separate illustrative figure.

5.2. Aggressive Deinterleaving. Aggressive deinterleaving eliminates all overlapping PMCs. The number of output PMCs can be smaller, yet also larger than the number of input PMCs.

5.2.1. Example. To demonstrate the process of aggressive deinterleaving, we take the set of PMCs listed in cf. Table 5.1 as an example. Aggressive deinterleaving transforms the set $C_X = \{A, B, C, D, E, F, G\}$ into the

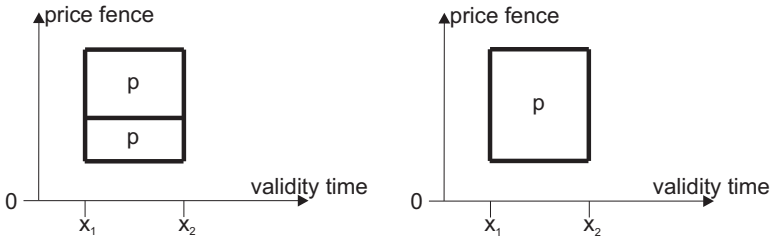

Fig. 5.4: Merging of vertically adjacent PMCs (before and after).

set $C_Y = \{A', B', C', D', E', F', G', H'\}$ are presented in cf. Table 5.2. The PMCs before and after aggressive deinterleaving are visually presented in cf. Fig. 5.5. As outlined in Sect. 5.1, cf. Fig. 5.3, all PMCs in C_X must have (1.) the same PAM and billing unit, and (2.) have the price fence $u_{min} = 0$ and $u_{max} = \infty$.

PMC	t_{VFrom}	t_{VTo}	**price**
A	9	12	1
B	2	9	3
C	7	9	1
D	0	7	1
E	7	11	4
F	6	15	1
G	16	18	2

Table 5.1: Example PMCs before aggressive deinterleaving.

PMC	t_{VFrom}	t_{VTo}	**price**
A'	0	2	1
B'	2	6	4
C'	6	7	5
D'	7	9	9
E'	9	11	6
F'	11	12	2
G'	12	15	1
H'	16	18	2

Table 5.2: Example PMCs after aggressive deinterleaving.

5.2.2. Deinterleaving Algorithm. Algorithm 5.2.1 is the main deinterleaving algorithm. In lines 3 and 4, it sorts the given PMCs by their t_{VFrom} and t_{VTo} dates. The sorted PMCs are stored in two queues, Q_{VFrom} and Q_{VTo}. *lastcut* is initialized with the lowest t_{VFrom} date of any PMC. In the iterations of the **while** loop, algorithm 5.2.1 determines all *cuts* in a sorted order. A PMC *c starts* (respectively *stops*) at a given point $p \in \mathbb{N}$ if $c.t_{VFrom} = p$, respectively $c.t_{VTo} = p$. A *cut* is a point in time where at least one PMC in C_X starts or stops. First, algorithm 5.2.1 computes the price of the next PMC to be added to its output C_Y by (i.) adding the prices of all PMCs in A_{start} to and (ii.) subtracting the prices of all PMCs in A_{end} from the price of the previous PMC added to C_Y. In the first iteration, A_{start} contains all PMCs starting at the smallest t_{VFrom} date of any PMC in C_X and $A_{end} = \emptyset$. In line 17, algorithm 5.2.1 calls algorithm 5.2.2 to determine the next cut, i.e. the smallest t_{VFrom} or t_{VTo} date of all PMCs which have not yet been polled from Q_{VFrom} and Q_{VTo} (see cf. Table 5.5). Algorithm 5.2.2 also stores the PMCs starting and stopping at the next cut in the sets A_{start} and A_{end}. cf. Table 5.3 presents the content of the variables *lastcut*, *cut* and *currentprice* when line 17 of

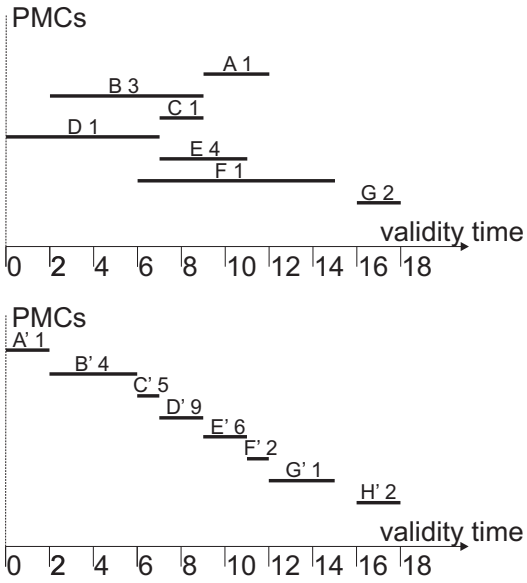

Fig. 5.5: Example PMCs before and after aggressive deinterleaving. The numbers indicate the PMCs' price.

algorithm 5.2.1 has been executed. *Iteration* refers to the iteration of the **while** loop during which the variables take the presented state. In lines 18-21, algorithm 5.2.1 adds a new PMC to C_Y; cf. Table 5.4 shows the PMCs created in each iteration of the *while* loop.

Algorithm 5.2.2 is used by the main deinterleaving algorithm to determine the next *cut*, i.e. the smallest t_{VFrom} or t_{VTo} date where at least one PMC remaining in Q_{VFrom} and Q_{VTo} starts or stops. The queues are sorted, so:

$$cut = min(Q_{VFrom}.peek().t_{VFrom}, Q_{VTo}.peek().t_{VTo})$$

Since several PMCs can start/stop at *cut*, algorithm 5.2.2 polls PMCs from Q_{VFrom} and Q_{VTo} in two *while* loops.

Iteration	1	2	3	4	5	6	7	8	9
lastcut	0	2	6	7	9	11	12	15	16
cut	2	6	7	9	11	12	15	16	18
currentprice	1	4	5	9	6	2	1	0	2

Table 5.3: Example PMCs: Content of variables *lastcut*, *cut* and *currentprice* after execution of line 17 of algorithm 5.2.1.

Iteration	1	2	3	4	5	6	7	8	9
t_{VFrom}	0	2	6	7	9	11	12	n.a.	16
t_{VTo}	2	6	7	9	11	12	15	n.a.	18
price	1	4	5	9	6	2	1	n.a.	2

Table 5.4: Example PMCs: New PMCs created in lines 18-21 of algorithm 5.2.1.

Q_{VFrom}	D	B	F	C	E	A	G
Iteration	0	1	2	3	3	4	8
Q_{VTo}	D	B	C	E	A	F	G
Iteration	3	4	4	5	6	7	9

Table 5.5: Example PMCs: Iterations of *while* loop in algorithm 5.2.1 during which PMCs are polled from the two queues.

Algorithm 5.2.1 Deinterleave

Require: $C_X = \{c_1, ..., c_n\}$: PMCs to be deinterleaved
Require: $C_Y = \emptyset$: holder for algorithm output
1: $A_{start} \leftarrow \emptyset$
2: $A_{end} \leftarrow \emptyset$
3: $Q_{VFrom} \leftarrow sortByValidFrom(C_X)$
4: $Q_{VTo} \leftarrow sortByValidTo(C_X)$
5: $cut \leftarrow 0$
6: $lastcut \leftarrow GetNext(Q_{VFrom}, Q_{VTo}, A_{start}, A_{end})$
7: $currentprice \leftarrow 0$
8: **while** $|Q_{VFrom}| + |Q_{VTo}| > 0$ **do**
9: **for all** $c \in A_{start}$ **do**
10: $currentprice \leftarrow currentprice + c.price$
11: **end for**
12: **for all** $c \in A_{end}$ **do**
13: $currentprice \leftarrow currentprice - c.price$
14: **end for**
15: $A_{start} \leftarrow \emptyset$
16: $A_{end} \leftarrow \emptyset$
17: $cut \leftarrow GetNext(Q_{VFrom}, Q_{VTo}, A_{start}, A_{end})$
18: **if** $currentprice > 0$ **then**
19: $c :=$ new PMC
20: $c.price = currentprice$
21: $c.t_{VFrom} = lastcut$
22: $c.t_{VTo} = cut$
23: $C_Y = C_Y \oplus c$
24: **end if**
25: $lastcut \leftarrow cut$
26: **end while**

5.2.3. Upper Bound of Number of PMCs After Deinterleaving. As seen in cf. Fig. 5.5, deinterleaving can increase the number of PMCs. This section proves that the number of PMCs will, however, always be less than doubled: $|C_Y| \leq 2|C_X| - 1$.

Proof.

Induction basis: $|C_X| = 1$. Then: $|C_Y| = 1 \leq 2 * 1 - 1$

Induction hypothesis: $\forall |C_X| < n : \quad |C_Y| \leq 2|C_X| - 1$

Induction step: Prove that $|C_X| = n : \quad |C_Y| \leq 2|C_X| - 1$:

Let $C_{Y,n-1}$ be the resulting set of PMCs of deinterleaving $C_X \setminus \{c\}$ for some $c \in C_X$.

$\Rightarrow |C_{Y,n-1}| \leq 2|C_X \setminus \{c\}| - 1$ according to induction hypothesis.

Prove that $|C_Y| \leq |C_{Y,n-1}| + 2$, i.e. adding c to $C_X \setminus \{c\}$ will increase the number of PMCs after deinterleaving by at most 2.

1. If and only if $\forall c_o \in C_X \setminus \{c\} : \quad c.t_{VFrom} \neq c_o.t_{VFrom} \wedge c.t_{VFrom} \neq c_o.t_{VTo}$, then $|C_Y| \geq |C_{Y,n-1}| + 1$. Illustration: If and only if c starts where no other PMC in C_X starts or stops, then one PMC in $C_{Y,n-1}$ is split in two by adding c to $C_X \setminus \{c\}$, so the number of PMCs after deinterleaving is increased by 1.

Algorithm 5.2.2 GetNext

Require: Q_{VFrom}: Queue containing PMCs sorted by t_{VFrom}
Require: Q_{VTo}: Queue containing PMCs sorted by t_{VTo}
Require: $A_{start} = \emptyset, A_{end} = \emptyset$
1: $min \leftarrow \infty$
2: **if** $|Q_{VFrom}| > 0$ **then**
3: $\quad min \leftarrow Q_{VFrom}.poll().t_{VFrom}$
4: **end if**
5: **if** $|Q_{VTo}| > 0 \wedge min > Q_{VTo}.peek().t_{VTo}$ **then**
6: $\quad min \leftarrow Q_{VTo}.poll().t_{VFrom}$
7: **end if**
8: **while** $|Q_{VFrom}| > 0 \wedge Q_{VFrom}.peek().t_{VFrom} = min$ **do**
9: $\quad A_{start} \leftarrow A_{start} \oplus Q_{VFrom}.poll()$
10: **end while**
11: **while** $|Q_{VTo}| > 0 \wedge Q_{VTo}.peek().t_{VTo} = min$ **do**
12: $\quad A_{end} \leftarrow A_{end} \oplus Q_{VTo}.poll()$
13: **end while**
14: **return** min

Otherwise, the number of PMCs after deinterleaving remains unchanged.

2. Analogously: If and only if $\forall c_o \in C_X \setminus \{c\}: \quad c.t_{VTo} \neq c_o.t_{VTo} \wedge c.t_{VFrom} \neq c_o.t_{VTo}$, then $|C_Y| \geq |C_{Y,n-1}| + 1$.
3. In total, $|C_Y| \leq |C_{Y,n-1}| + 2$.

$\Rightarrow |C_Y| \leq |C_{Y,n-1}| + 2 \leq 2|C_X \setminus c| - 1 + 2 \leq 2|C_X| - 1$. □

5.2.4. Time Complexity of Deinterleaving Algorithm. The deinterleaving algorithm has a time complexity of $O(n \cdot log(n))$, where $n = |C_X|$.

Proof. Lines 1-2, 5 and 7 of Algorithm 5.2.1 take $O(1)$. Lines 3-4 take $O(n \cdot log(n))$ due to sorting. Line 6 of Algorithm 5.2.1 calls Algorithm 5.2.2, one call of which has complexity $O(n)$: Lines 1-7 of Algorithm 5.2.2 have complexity $O(1)$. Lines 8-13 take $O(n)$, since initially the queues Q_{VFrom} and Q_{VTo} are both filled with n PMCs, and at most all of these n PMCs can be polled by lines 9 and 12. Lines 15-16 and 18-25 of Algorithm 5.2.1 take $O(1)$ over one iteration of the **while** loop, and therefore take $O(n)$ over all $O(n)$ iterations of the **while** loop.

We use aggregate analysis to show that line 17 of Algorithm 5.2.1 takes $O(n)$ over all iterations of the **while** loop, despite the call of Algorithm 5.2.2, which can take $\omega(1)$. As mentioned before, lines 1-7 of Algorithm 5.2.2 take O(1). Initially, the queues Q_{VFrom} and Q_{VTo} are both filled with n PMCs, so $|Q_{VFrom}| = |Q_{VFrom}| = n$. Since both algorithms never push any PMCs onto the queues, lines 8-13 Algorithm 5.2.2 can pop at most those n PMCs from the queues over all iterations of the **while** loop in Algorithm 5.2.1. In fact, since line 6 of Algorithm 5.2.1 has already popped one or more PMCs from each queue, the **while** loop actually causes less than n pop operations.

Analogously, we use aggregate analysis to show that lines 9-14 of Algorithm 5.2.1 take $O(n)$. Since A_{start} and A_{end} are filled with the elements popped from the queues by Algorithm 5.2.2 and cleared by every iteration of the **while** loop of Algorithm 5.2.1, each of the $O(n)$ PMCs initially in the queues is exactly once an element of A_{start} and A_{end}. Therefore, both **for** loops in lines 9-14 execute $O(n)$ times over all iterations of the **while** loop. Each execution of any of the **for** loops takes $O(1)$.

In total, the **while** loop of Algorithm 5.2.1 takes $O(n)$, the sorting operations of lines 3-4 take $O(n \cdot log(n))$ and all other operations of Algorithm 5.2.1 take $O(1)$. Therefore, Algorithm 5.2.1 has a complexity of $O(n \cdot log(n))$. □

5.3. Gentle Deinterleaving. Gentle deinterleaving reduces the number of PMCs as far as actually possible, and never increases the number of PMCs. This feature is achieved at the expense that some overlapping PMCs may remain in the deinterleaving output. An example for gentle deinterleaving can be seen in cf. Fig. 5.6. In contrast, aggressive deinterleaving eliminates all overlapping PMCs, at the expense of possibly increasing the number of output PMCs. The advantage of having no overlapping PMCs is that for marketplace operations

such as payment calculation, PMCs are more easily organizable, e.g. in tree structures. The choice between aggressive and gentle deinterleaving needs to be made based on the requirements of the marketplace application.

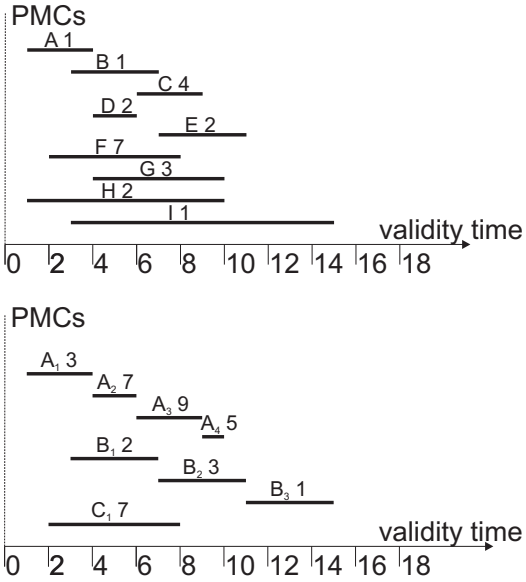

Fig. 5.6: Example PMCs before and after gentle deinterleaving. The numbers indicate the PMCs' price.

5.3.1. Gentle Deinterleaving Algorithm. A PMC is *lonely* if both its t_{VFrom} and t_{VTo} dates are not identical with the t_{VFrom} or t_{VTo} date of any other PMC. The gentle deinterleaving algorithm follows this procedure:

1. Generate PMC graph from the input set of PMCs, each of whose nodes represents one PMC. Two nodes are connected if and only if the represented PMCs start or stop at the same point, or both. This procedure can be performed e.g. by hashing the validFrom/validTo dates of the input PMCs or other duplicate identification approaches. Figure 5.7 shows the PMC graph generated according to the example PMCs shown in cf. Fig 5.6.
2. Identify sets of connected PMCs using standard algorithms for detecting strongly connected components. Our example yields three groups: $G_1 = \{A, D, C, G, H\}, G_2 = \{B, E, I\}, G_3 = \{F\}$.
3. For each set of connected PMCs, perform aggressive deinterleaving (Algorithm 5.2.1), and add the resulting PMCs to output.

The resulting set of PMCs is minimal, i.e. the number of PMCs cannot be reduced any further (except by merging, see Sect. 5.4). Deinterleaving any two PMCs of the resulting set will either leave the number of PMCs unchanged, or add a PMC. The time complexity is $O(n \cdot log(n))$, identically to aggressive deinterleaving. The optimality question is (of course) different, and is left to future work at this point.

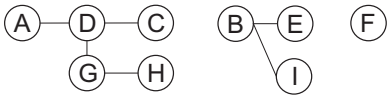

Fig. 5.7: Example PMC graph. PMC F forms a set of its own.

5.4. Merging. Merging is the second part of the aggregation procedure, and is intended to further reduce the number of PMCs based on the operation depicted in cf. Fig. 5.1. The deinterleaving procedure may output adjacent PMCs with the same PAM, billing unit and price. We refer to *merging* as the replacement of adjacent PMCs with the same PAM, billing unit and price by one single PMC. To demonstrate the process of merging,

we take the PMCs listed in cf. Table 5.6 as an example. Merging then transforms these PMCs into the PMCs listed in cf. Table 5.7.

PMC	t_{VFrom}	t_{VTo}	**price**
A	2	5	1
B	5	7	3
C	7	9	3
D	9	11	3
E	12	14	2
F	14	15	2

Table 5.6: Example PMCs before merging.

PMC	t_{VFrom}	t_{VTo}	**price**
A'	2	5	1
B'	5	11	3
E'	12	15	2

Table 5.7: Example PMCs after merging.

Assuming the deinterleaving algorithm has been performed before, the PMCs are sorted by t_{VFrom} (and t_{VTo}). Algorithm 5.4.1 demonstrates merging in time complexity $O(n)$.

Algorithm 5.4.1 Merge

Require: $C_Y = \{c_1, ..., c_n\}$: queue containing PMCs sorted by t_{VFrom} property by Algorithm 5.2.1
Require: $C_Z = \emptyset$: holder for algorithm output
1: $lastPMC = nil$
2: **for all** $c \in C_Y$ **do**
3: **if** $lastPMC \neq nil \wedge lastPMC.t_{VTo} == c.t_{VFrom}$ **then**
4: $lastPMC.t_{VTo} = c.t_{VTo}$
5: **else**
6: $C_Z = C_Z \oplus c$
7: $lastPMC = c$
8: **end if**
9: **end for**

5.5. Deinterleaving and merging based on price fence. The three previous sections demonstrated deinterleaving and merging based on validity time frame. However, as depicted in cf. Fig. 5.4, PMCs can also be deinterleaved and merged based on their price fence, given the conditions mentioned in Sect. 5.1. The procedure is identical, yet the properties t_{UFrom} and t_{UTo} are replaced with u_{min} and u_{max} in the algorithms 5.2.1 and 5.4.1. The operation of adding a PMC to the output set of PMCs in algorithm 5.2.1 can be enhanced with tree structures that produce a set of linked lists of PMCs, where each linked list contains PMCs with identical t_{UFrom} and t_{UTo} dates, which are sorted by price fence. This will increase the time complexity of the aggregation procedure (though not the asymptotic complexity of $O(n \cdot log(n))$), however decreases the effort for payment calculation. This approach is appropriate since aggregation of price models is typically performed only once for one composite product, yet the payment calculation for the resulting aggregate price model is performed multiple times.

5.6. Aggregation Bases. The aggregation procedures presented in the previous sections aggregate PMCs based on their validity time frame, i.e. the part-condition $t_{VFrom} \leq t \leq t_{VTo}$ of a PMC's *condition*. Since the subset of USDL used to define price models will be extended in the future, the *condition* of a PMC may be amended by further part-conditions of the form $a_{low} \leq a \leq a_{high}$. Such part-conditions exhibit the same

properties as the validity time frame: Both have a lower and an upper value referred to as $c.lower(A)$ and $c.upper(A)$. The applying units are limited to the units consumed between the lower and the upper values. As a result, the aggregation procedures can be adapted, so they can also be applied to these part-conditions.

To facilitate such an adaptation, the this section presents the formal concept of the *aggregation basis*, which generalizes the aggregation procedures presented in previous sections from the validity time frame $t_{VFrom} \leq t \leq t_{VTo}$ to any part-condition of the form $a_{low} \leq a \leq a_{high}$. The *aggregation basis* determines by which of their properties PMCs are aggregated. For each aggregation basis A, the PMC possesses a lower and an upper bound, referred to as $c.lower(A)$ and $c.upper(A)$. The lower and upper bounds take values from a totally ordered set, referred to as the *aggregation set* S_A. The condition $c.lower(A) < c.upper(A)$ must hold for the PMC to be properly defined. In this paper, we presented the aggregation basis *validity time frame* A_{VT}. Let $\mathcal{A}$ be the set of all aggregation bases. Accordingly, $\mathcal{A} = \{A_{VT}\}$. $\mathcal{A}$ will be amended once further aggregation bases are added to the structural definition of a PMC. Table 5.8 shows the lower bound, upper bound and aggregation set for the aggregation basis. The price fence is currently not considered an aggregation basis, since cf. Fig. 5.2 shows that aggregation procedures are not necessarily applicable in all cases of price fences.

A	$c.lower(A)$	$c.upper(A)$	S_A
A_{VT}	$c.t_{VFrom}$	$c.t_{VTo}$	$\mathbb{N}_0 \cup \{\infty\}$

Table 5.8: Lower bound, upper bound and aggregation set for a PMC c.

5.7. Multi-Dimensional Deinterleaving and Merging Procedures. Aggregating PMCs by several aggregation variables is significantly more complex than by one single aggregation variable, since PMCs may have different lower/upper values for several aggregation variables. Due to the high complexity of the facts and circumstances, this section merely gives a short sketch of multi-dimensional aggregation of PMCs. Future work shall investigate more deeply on the issue.

5.7.1. Approaches. The trivial way to aggregate PMCs by several aggregation bases is to iterate over all aggregation bases and apply the one-dimensional aggregation presented in previous sections each time. The limitation of this approach is that the number of PMCs is not reduced as far as possible, as one can exemplary see in cf. Fig. 5.8. Aggregation by aggregation basis 1 only merges the PMCs in the upper right corner, and aggregation by aggregation basis 2 introduces no changes at all. However, it is possible to aggregate all PMCs into one 'big' PMC by alternatingly aggregating by each aggregation basis. Multi-dimensional aggregation algorithms can therefore trace which PMCs have been deinterleaved and merged by aggregating by one given aggregation basis, resulting in one or more new PMCs with new lower/upper values to be considered when aggregating over another aggregation basis. As a result, aggregation over one aggregation basis may have to be performed multiple times to minimize the number of PMCs, as seen in cf. Fig. 5.8.

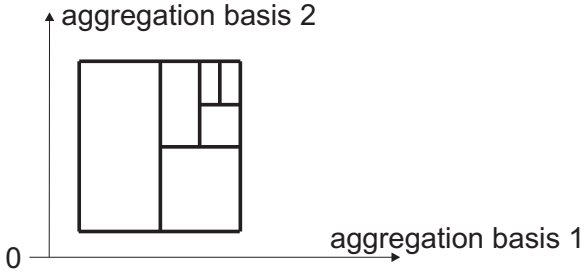

Fig. 5.8: Multi-Dimensional Merging Example: The components need to be merged alternatingly by the two aggregation bases to reduce the number of PMCs as far as possible.

5.7.2. Number of Output PMCs. The problem of multi-dimensional aggregation is that PMCs may have different lower and upper boundaries for several aggregation bases. In the one-dimensional case, deinterleaving two PMCs to eliminate overlapping introduces at most one additional PMC (see Sect. 5.2.3). In the

multi-dimensional case, the number of PMCs may increase much more. One simple solution is to aggregate PMCs only if their lower and upper values are identical in all but at most one aggregation bases. Similarly to the gentle deinterleaving algorithm, this may leave multidimensionally overlapping PMCs in return for keeping the number of PMCs low. Future work shall include detailed analyses on the upper boundary of the number of output PMCs in the multi-dimensional case, analogously to the proof in Sect. 5.2.3.

6. Conclusions and Outlook. This paper presented an approach to define price models using a subset of USDL, as well as an approach to aggregate several price models in order to support composite services in a cloud computing application environment. From a business perspective, future work should investigate how our approach fits into existing approaches towards pricing strategy in electronic services, such as [14] and [15]. Technically, the theory behind aggregating price models yields interesting algorithmic and mathematical questions. While the price model structure presented in this paper is still comparably simple, introducing extensions of the subset of USDL for more fine-grained price models will result in an array of new problems to be solved. These extensions include:

- Different prices for different consumer groups.
- SLAs specifying prices depending on service level chosen by consumer, as well as penalties paid by provider to consumer if service level is not met.

These changes will require the payment calculation and aggregation approaches to be adapted, yet the concept behind the algorithms presented in this paper remains valid.

Future work shall include multi-dimensional aggregation algorithms to handle several aggregation bases. The optimality of the gentle deinterleaving algorithm still requires investigation. Also, we shall present investigations on the effects of adding more aggregation bases on the algorithmic realization of price aggregation. Furthermore, we strive to define PMCs such that price fences can be considered an aggregation variable, avoiding the problem depicted in cf. Fig. 5.2. Finally, the current approach towards price aggregation is that the aggregate price model is simplified as much as possible while keeping the generated payments exactly the same for any number and temporal occurrence of the consumed units. We shall investigate on approaches towards price aggregation that give up this goal to an acceptable extent, in return for being able to offer a simple price model to the service consumer even when the service relies on third-party services with highly heterogeneous price models.

Acknowledgement. The research leading to these results has partially received funding from the 4CaaSt project (http://www.4caast.eu/) from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 258862. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

REFERENCES

[1] F. Noyer, *Characteristics of Electronic Marketplaces for Cloud Computing Services*, Master Thesis, University of Fribourg, Switzerland

[2] J. Cardoso, A. Barros, N. May, U. Kylau, *Towards a Unified Service Description Language for the Internet of Services: Requirements and First Developments*, IEEE International Conference on Services Computing, IEEE Computer Society Press, 2010.

[3] M. Eurich, A. Giessmann, T. Mettler, and K. Stanoevska-Slabeva, *Revenue Streams of Cloud-based Platforms: Current State and Future Directions*, in Proceedings of the Seventeenth Americas Conference on Information Systems (AMCIS), 2011, Paper 302.

[4] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Sheng, *Quality Driven Web Services Composition*, Proceeding WWW '03 Proceedings of the 12th International Conference on World Wide Web.

[5] T. Unger, F. Leymann, S. Mauchart, T. Scheibler, *Aggregation of Service Level Agreements in the Context of Business Processes*, 12th International IEEE Enterprise Distributed Object Computing Conference, 2008. EDOC '08.

[6] T. Kohlborn, A. Korthaus, C. Riedl, H. Krcmar, *Service aggregators in business networks*, 13th Enterprise Distributed Object Computing Conference Workshops, 2009. EDOCW 2009.

[7] H. Elshaafi, J. McGibney, D. BotvichODO, *Business Driven Optimisation of Service Compositions*, 7th International Conference on Next Generation Web Services Practices (NWeSP), 2011

[8] F. Faria, J. M. Nogueira, *Context-Based Application-Aware Pricing for Composite Mobile Services in Wireless Networks*, Wireless Days (WD), 2010 IFIP

[9] V. Agarwal, N. Karnik, A. Kumar, *Metering and accounting for composite e-services*, IEEE International Conference on E-Commerce, 2003. CEC 2003.

[10] T. T. Nagle, R. K. Holden, G. M. Larsen, *Pricing, Praxis der optimalen Preisfindung*, Springer-Verlag, Berlin/Heidelberg, 1998.

[11] C. Homburg, *Marketingmanagement, 4. Auflage*, Gabler-Verlag, Springer Fachmedien, Wiesbaden, 2011.

[12] X. Wang, *Price Heuristics for Highly Efficient Profit Optimization of Service Composition*, 2011 IEEE International Conference on Services Computing (SCC).
[13] V. Kantere, D. Dash, G. Francois, S. Kyriakopoulou, A. Ailamaki, *Optimal service pricing for a cloud cache*, IEEE Transactions on Knowledge and Data Engineering, 2011
[14] A. Dixit, T. W. Whipple, G. M. Zinkhan, E. Gailey, *A taxonomy of information technology-enhanced pricing strategies*, Journal of Business Research, Volume 61, Issue 4, 2008, pp. 275-283
[15] E. Iveroth, A. Westelius, C.-J. Petri, N.-G. Olve, M. Cöster, F. Nilsson, *How to differentiate by price: Proposal for a five-dimensional model*, European Management Journal, Available online 1 August 2012
[16] A. Watanabe, *Web Service Selection Algorithm Using Vickrey Auction*, 2012 IEEE 19th International Conference on Web Services (ICWS)

Edited by: José Luis Vásquez-Poletti, Dana Petcu and Francesco Lelli
Received: Sep 20, 2012
Accepted: Oct. 15, 2012