Klaim-DB: A Modeling Language for Distributed Database Applications

Xi Wu[2], Ximeng Li[1], Alberto Lluch Lafuente[1(✉)],
Flemming Nielson[1], and Hanne Riis Nielson[1]

[1] DTU Compute, Technical University of Denmark, Kgs. Lyngby, Denmark
{ximl,albl,fnie,hrni}@dtu.dk
[2] Shanghai Key Laboratory of Trustworthy Computing,
Software Engineering Institute, East China Normal University, Shanghai, China
xiwu@sei.ecnu.edu.cn

Abstract. We present the modelling language, Klaim-DB, for distributed database applications. Klaim-DB borrows the distributed nets of the coordination language Klaim but essentially re-incarnates the tuple spaces of Klaim as databases, and provides high-level language abstractions for the access and manipulation of structured data, with integrity and atomicity considerations. We present the formal semantics of Klaim-DB and illustrate the use of the language in a scenario where the sales from different branches of a chain of department stores are aggregated from their local databases. It can be seen that raising the abstraction level and encapsulating integrity checks (concerning the schema of tables, etc.) in the language primitives for database operations benefit the modelling task considerably.

1 Introduction

Today's data-intensive applications are becoming increasingly distributed. Multi-national collaborations on science, economics, military etc., require the communication and aggregation of data extracted from databases that are geographically dispersed. Distributed applications including websites frequently adopt the Model-View-Controller (MVC) design pattern in which the "model" layer is a database. Fault tolerance and recovery in databases also favors the employment of distribution and replication. The programmers of distributed database applications are faced with not only the challenge of writing good queries, but also that of dealing with the coordination of widely distributed databases. It is commonly accepted in the formal methods community that the *modelling* of complex systems in design can reduce implementation errors considerably [1,11].

Klaim [2] is a kernel language for specifying distributed and coordinated processes. In Klaim, processes and information repositories exist at different localities. The information repositories are tuple spaces, that can hold data and

X. Wu – Part of the work was done when Xi Wu was a visiting researcher at DTU compute.

T. Holvoet and M. Viroli (Eds.): COORDINATION 2015, LNCS 9037, pp. 197–212, 2015.
DOI: 10.1007/978-3-319-19282-6_13

code. Processes can read tuples from (resp. write tuples to) local or remote tuple spaces, or spawn other processes to be executed at certain localities. Many distributed programming paradigms can be modelled in Klaim.

While Klaim provides an ideal ground for the modelling of networked applications in general, the unstructured tuple spaces and low-level operations mostly targeting individual tuples create difficulty in the description of the data-manipulation tasks usually performed using a high-level language such as SQL. A considerable amount of meta-data needed by databases has to be maintained as particular tuples or components of tuples, the sanity checks associated with database operations have to be borne in mind and performed manually by the programmer, difficulties arise when batch operations are performed and atomicity guarantees are needed, and so on.

To support the modelling of applications operating on distributed, structured data, we propose the language Klaim-DB, which is inspired by Klaim in that it allows the distribution of *nodes*, and remote operations on data. Its succinct syntax eases the precise formulation of an operational semantics, giving rigor to high-level formal specification and reasoning of distributed database applications. The language also borrows from SQL, in that it provides structured data organized as databases and tables, and high-level actions that accomplish the data-definition and data-maniputation tasks ubiquitous in these applications.

We use a running example of database operations in the management of a large-scale chain of department stores. Each individual store in the chain has its local database containing information about the current stock and sales of each kind of product. The semantic rules for the core database operations will be illustrated by the local query and maintenance of these individual databases, and our final case study will be concerned with data aggregation across multiple local databases needed to generate statistics on the overall product sales.

This paper is structured as follows. In Section 2, the syntax of Klaim-DB is presented, which is followed by the structural operational semantics specified in Section 3, with illustration of the semantic rules for the main database operations. Our case study is then presented in Section 4. Extensions of Klaim-DB and a discussion of alternative language design choices are covered in Section 5. We conclude in Section 6 with a discussion of related works.

2 Syntax

The syntax of Klaim-DB is presented in Table 1. A net N models a database system that may contain several databases situated at different localities. As in standard Klaim [2], we distinguish between physical localities, also called sites ($s \in \mathcal{S}$), and logical localities ($\ell \in \mathcal{L}$) that are symbolic names used to reference sites. At each site s, there is an "allocation environment" $\rho : \mathcal{L} \hookrightarrow \mathcal{S}$ mapping the logical localities known at s to the sites referred to by them.

We assume for simplicity that there is only one database at each site. The syntax $s ::_\rho C$ for a node of the net captures the ensemble C of processes and tables of the database at the site s, where the allocation environment is ρ.

Table 1. The Syntax of Klaim-DB

$$\begin{aligned}
N &::= \mathsf{nil} \mid N_1 || N_2 \mid (\nu s)N \mid s ::_{\rho} C \\
C &::= P \mid (I, R) \mid C_1 | C_2 \\
P &::= \mathsf{nil} \mid a.P \mid A(\tilde{e}) \mid \mathsf{foreach_s}\ T\ \mathsf{in}\ R : P \mid \mathsf{foreach_p}\ T\ \mathsf{in}\ R : P \mid P_1; P_2 \\
a &::= \mathsf{insert}(t, tb)@\ell \mid \mathsf{insert\_tb}(TBV, tb)@\ell \mid \mathsf{delete}(T, \psi, tb, !TBV)@\ell \mid \\
&\quad\ \ \mathsf{sel\_ext}(T, \psi, tb, t, !TBV)@\ell \mid \mathsf{sel\_int}(T, \psi, TBV, t, !TBV') \mid \\
&\quad\ \ \mathsf{update}(T, \psi, t, tb)@\ell \mid \mathsf{aggr}(T, \psi, tb, f, T')@\ell \mid \mathsf{create}(I)@\ell \mid \mathsf{drop}(tb)@\ell \\
t &::= e \mid t_1, t_2 \\
T &::= e \mid \ !x \mid T_1, T_2
\end{aligned}$$

The parallel composition of different nodes is represented using the $||$ operator. With the restriction operator $(\nu s)N$, the scope of s is restricted to N. Each table is represented by a pair (I, R) where I is an interface that publishes certain structural information about the table as attributes (for example, $I.id$ stands for the table identifier and $I.sk$ is a schema describing the data format of the table), and R is a multiset of tuples, in which each tuple corresponds to one row of data. The construct $C_1|C_2$ is the parallel composition of different components.

We distinguish between tuples t and templates T. A template T can contain not only actually fields that are expressions e, but also formal fields $!x$ where x is a variable that can be bound to values.

A process P can be an inert process nil, an action prefixing $a.P$, a parameterized process invocation $A(\tilde{e})$, a looping process $\mathsf{foreach_s}\ T\ \mathsf{in}\ R : P$, or $\mathsf{foreach_p}\ T\ \mathsf{in}\ R : P$, or a sequential composition $P_1; P_2$, for which we require that $bv(P_1) \cap fv(P_2) = \emptyset$. Looping is introduced in addition to recursion via process invocation, to ease the task of traversing tables or data selection results in a natural way. We also allow both prefixing and sequential composition in our language, as in CSP [4]. Sequential composition is needed to facilitate the specification of possible continuation after the completion of a loop, whereas prefixing is retained and used in situations where substitutions need to be applied after receiving an input. The difference between the two variants of looping process is that the sequential loop $\mathsf{foreach_s}\ T\ \mathsf{in}\ R : P$ goes through the different rounds sequentially, while the parallel loop $\mathsf{foreach_p}\ T\ \mathsf{in}\ R : P$ forks one parallel process for each round.

A process can perform nine different kinds of actions. Actions $\mathsf{insert}(t, tb)@\ell$, $\mathsf{insert\_tb}(TBV, tb)@\ell$, $\mathsf{delete}(T, \psi, tb, !TBV)@\ell$, $\mathsf{sel\_ext}(T, \psi, tb, t, !TBV)@\ell$, $\mathsf{sel\_int}\ (T, \psi, TBV, t, !TBV')$, $\mathsf{update}(\ T, \psi, t, tb)@\ell$ and $\mathsf{aggr}(T, \psi, tb, f, T')@\ell$ are used to access/manipulate the data inside a table; they resemble the operations performed by the "Data-Manipulation Language" in SQL. On the other hand, actions $\mathsf{create}(I)@\ell$, and $\mathsf{drop}(tb)@\ell$ are used for the creation and deletion of a table — they correspond to the operations performed by the "Data-Definition Language" in SQL.

The actions $\mathsf{insert}(t, tb)@\ell$ and $\mathsf{insert\_tb}(TBV, tb)@\ell$ are used to insert a new row t, or all the rows of a table bound to the table variable TBV, into a table

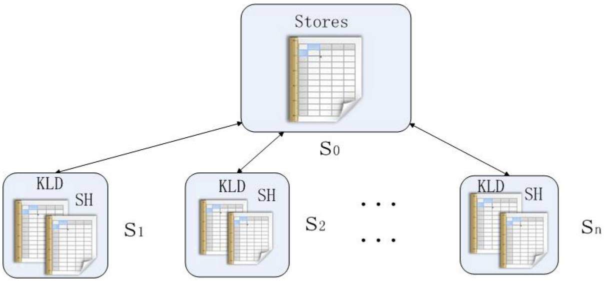

Fig. 1. Running Example

named tb inside the database at ℓ, respectively. On the other hand, the action $\mathsf{delete}(T, \psi, tb, !TBV)@\ell$ deletes all rows matching the pattern T and the predicate ψ from table tb in the database located at ℓ, and binds the deleted rows to the table variable TBV.

The language has two variants of "selection": an "external" one that selects data from tables actually existing in databases, identified by their table identifiers, and an "internal" one that selects data from temporary tables bound to table variables. The action $\mathsf{sel\_ext}(T, \psi, tb, t, !TBV)@\ell$ performs the "external" selection. It picks all rows matching the pattern T as well as satisfying the predicate ψ, from the table identified by tb of the database located at ℓ, and binds the resulting table into the table variable TBV. On the other hand, $\mathsf{sel\_int}(T, \psi, TBV, t, !TBV')$ performs the "internal" selection, i.e., from the content of the table variable TBV, and binds the resulting table further into the table variable TBV'. In $\mathsf{sel\_int}(T, \psi, TBV, t, !TBV')$, the meanings of T, ψ, and t are the same as those in $\mathsf{sel\_ext}(T, \psi, tb, t, !TBV)@\ell$. In both variants, we require that each component of t should be a value or a bound variable of T.

The action $\mathsf{update}(T, \psi, t, tb)@\ell$ replaces each row matching T yet satisfying ψ in table tb (at ℓ) with a new row t, while leaving the rest of the rows unchanged. It is required that $fv(t) \subseteq bv(T)$.

The action $\mathsf{aggr}(T, \psi, tb, f, T')@\ell$ applies the aggregation function f on the multiset of all rows meeting T and ψ in table tb (at ℓ) and binds the aggregation result to the pattern T'.

The action $\mathsf{create}(I)@\ell$ (resp. $\mathsf{drop}(tb)@\ell$) creates a table identified by I (resp. drops the table identified by tb) in the database at ℓ. An item is a row in a table containing sequences of values, which can be obtained from the evaluation of an expression e. Pattern-matching is used to manage the data inside a table by means of a given template T, which is a sequence of values and variables.

Setting the Scene for the Running Example. Consider the management of a large-scale chain of department stores, in which the head office can manage the sales of different imaginative brands (e.g., KLD, SH,...) in its individual branches, as shown in Figure 1.

We will use underlined symbols for logical and physical localities, as well as allocation environments, to distinguish between their uses in the example and elsewhere (e.g., in the semantics). Suppose the database of the head office is maintained on machine $\underline{s_0}$ and the databases of its branches are maintained on machines $\underline{s_1}$ to $\underline{s_n}$. The site $\underline{s_0}$ has the local environment $\underline{\rho_0} = [self \mapsto \underline{s_0}][\underline{\ell_1} \mapsto \underline{s_1}]...[\underline{\ell_n} \mapsto \underline{s_n}]$. We use $\underline{\rho_j} = [self \mapsto \underline{s_j}]$ as the local environment for each site $\underline{s_j}$ — this restricts database accesses across different local databases and corresponds to a centralized architecture.

City	*Address*	*Shop_Name*	*Brand*	*Logical_Locality*
CPH	ABC DEF 2, 1050	Shop1	$\{KLD, SH, ...\}$	$\underline{\ell_1}$
CPH	DEF HIJ 13, 2800	Shop2	$\{KLD, SH, ...\}$	$\underline{\ell_2}$
CPH	HIJ KLM 26, 1750	Shop3	$\{KLD, SH, ...\}$	$\underline{\ell_3}$
AAL	KLM NOP 3, 3570	Shop4	$\{LAM, IMK, ...\}$	$\underline{\ell_4}$
AAL	NOP QUW 18, 4500	Shop5	$\{LAM, IMK, ...\}$	$\underline{\ell_5}$
...	...	...	...	...

Fig. 2. The Table Stores

A table with identifier Stores exists in the database of the head office, and records the information of its branches, as shown in Figure 2. The header partially describes the schema I_0 of the table, and the subsequent rows, constituting the multiset R_0, contain the information of the different branches. Formally, we have $I_0.id$ = Stores, $I_0.sk =<$ “*City*” : *String*, “*Address*” : *String*, “*Shop_name*” : *String*, “*Brand*” : *Set*, “*Logical_Locality*” : *String* $>$.

Each database of a branch has several tables identified by the name of the brand, which record the information of the stock and sales of the corresponding brand. The table of one shoe brand, KLD, in one branch, is shown in Figure 3. The identifier of this table is KLD and the schema is $<$ “*Shoe_ID*” : *String*, “*Shoe_name*” : *String*, “*Year*” : *String*, “*Color*” : *String*, “*Size*” : *Int*, “*In-stock*” : *Int*, “*Sales*” : *Int* $>$.

Shoe_ID	*Shoe_name*	*Year*	*Color*	*Size*	*In-stock*	*Sales*
001	HighBoot	2015	red	38	5	2
001	HighBoot	2015	red	37	8	5
001	HighBoot	2015	red	36	3	1
001	HighBoot	2015	black	38	3	2
001	HighBoot	2015	black	37	5	2
002	ShortBoot	2015	green	38	2	0
002	ShortBoot	2015	brown	37	4	3
...	...	...	...	...	...	...

Fig. 3. The Table KLD in one branch

To sum up, the network of databases and operating processes can be represented by

$$\underline{s_0} ::_{\underline{\rho_0}} ((I_0, R_0)|C'_0) \mid\mid \underline{s_1} ::_{\underline{\rho_1}} ((I_1, R_1)|C'_1) \mid\mid ... \mid\mid \underline{s_n} ::_{\underline{\rho_n}} ((I_n, R_n)|C'_n),$$

where for $j \in \{1, ..., n\}$, (I_j, R_j) describes the local table for the brand KLD inside its database at $\underline{s_j}$, and for each $k \in \{0, ..., n\}$, C'_k stands for the remaining processes and tables at the site $\underline{s_k}$.

3 Semantics

We devise a structural operational semantics [8] for Klaim-DB, as shown in Table 3 and Table 4. The semantics is defined with the help of a structural congruence — the smallest congruence relation satisfying the rules in Table 2, where the α-equivalence of N and N' is denoted by $N \equiv_\alpha N'$.

Table 2. The Structural Congruence

$N_1 \vert\vert N_2 \equiv N_2 \vert\vert N_1$	$(\nu s_1)(\nu s_2)N \equiv (\nu s_2)(\nu s_1)N$
$(N_1 \vert\vert N_2) \vert\vert N_3 \equiv N_1 \vert\vert (N_2 \vert\vert N_3)$	$N_1 \vert\vert (\nu s)N_2 \equiv (\nu s)N_1 \vert\vert N_2$ (if $s \notin \mathsf{fn}(N_1)$)
$N \vert\vert \mathsf{nil} \equiv N$	$s ::_\rho C \equiv s ::_\rho C \vert \mathsf{nil}$
$N \equiv N'$ (if $N \equiv_\alpha N'$)	$s ::_\rho (C_1 \vert C_2) \equiv s ::_\rho C_1 \vert\vert s ::_\rho C_2$

We start by explaining our notation used in the semantic rules. The evaluation function $\mathcal{E}[\![\cdot]\!]_\rho$ evaluates tuples and templates under the allocation environment ρ. The evaluation is performed pointwise on all the components of tuples t and templates T, much in the manner of standard Klaim [2]. We denote by $[a/b]$ the substitution of a for b; in particular, when a and b are tuples of the same length, point-wise substitution is represented. Pattern matching is performed with the help of the *predicate* $match(eT, et)$ where eT is an evaluated template and et is an evaluated tuple. This $match(eT, et)$ can be defined in a manner much like that of [2]. In more detail, $match(eT, et)$ gives a boolean value indicating whether the pattern matching succeeded. In case it did, the substitution $\sigma = [et/eT]$ can be resulted. For example, $match((3, !x, !y), (3, 5, 7)) = \mathsf{true}$ and $[(3, 5, 7)/(3, !x, !y)] = [5/x, 7/y]$. We denote by $\psi\sigma$ the fact that the boolean formula ψ is satisfied after the substitution σ is applied to it. We will use $\uplus$, $\cap$ and $\setminus$ to represent the union, intersection and substraction, of multisets and the detailed definitions are given in Appendix A.

The notation $I[a \mapsto b][...]$ represents the update of the interface I that maps its attribute a to b; multiple updates are allowed. In addition, we will use $I.sk \downarrow_t^T$ to represent the projection of the schema $I.sk$ according to the template T (matching the format requirements imposed by $I.sk$) and the tuple t. When t only contains constants or the bound variables of T as its components, $I.sk \downarrow_t^T$ is a new schema that describes only the columns referred to in the variable components of t. Since we have left the schema under-specified, this projection operation is illustrated in Example 1, rather than formally defined.

Example 1. Suppose $I.sk$ $=<$ "*Shoe_ID*" : *String*, "*Shoe_name*" : *String*, "*Year*" : *String*, "*Color*" : *String*, "*Size*" : *Int*, "*In-stock*" : *Int*, "*Sales*" : *Int* $>$, which specifies that the table having I as its interface has seven columns: "*Shoe_ID*$''$ of type *String*, "Shoe_name" of type *String*, "Year" of type *String*, "Color" of type *String*, "Size" of type *Int*, "In-stock" of type *Int*, "Sales" of type *Int*. Suppose in addition that $T = ($"001", "*HighBoot*"$, !x, !y, !z, !w, !p)$ and $t = (y, z, p)$. Then $I.sk \downarrow_t^T =<$ "*Color*" : *String*, "*Size*" : *Int*, "*Sales*" : *Int* $>$. □

Table 3. The Semantics for Actions

$$\text{(INS)}\ \frac{\rho_1(l) = s_2 \quad I.id = tb \quad t \models I.sk \quad R' = R \uplus \{\mathcal{E}[\![t]\!]_{\rho_1}\}}{s_1 ::_{\rho_1} \mathsf{insert}(t, tb)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow s_1 ::_{\rho_1} P \mid\mid s_2 ::_{\rho_2} (I,R')}$$

$$\text{(INS\_TB)}\ \frac{\rho_1(l) = s_2 \quad I.id = tb \quad I'.sk = I.sk \quad R'' = R \uplus R'}{s_1 ::_{\rho_1} \mathsf{insert\_tb}((I',R'), tb)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow s_1 ::_{\rho_1} P \mid\mid s_2 ::_{\rho_2} (I,R'')}$$

$$\text{(DEL)}\ \frac{\begin{array}{c} R' = \{t \mid t \in R \wedge \neg(match(\mathcal{E}[\![T]\!]_{\rho_1}, t) \wedge \psi[t/\mathcal{E}[\![T]\!]_{\rho_1}])\} \\ R'' = \{t \mid t \in R \wedge match(\mathcal{E}[\![T]\!]_{\rho_1}, t) \wedge \psi[t/\mathcal{E}[\![T]\!]_{\rho_1}]\} \\ \rho_1(l) = s_2 \quad I.id = tb \quad \sigma' = [(I,R'')/TBV] \end{array}}{s_1 ::_{\rho_1} \mathsf{delete}(T, \psi, tb, !TBV)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow s_1 ::_{\rho_1} P\sigma' \mid\mid s_2 ::_{\rho_2} (I,R')}$$

$$\text{(SEL\_EXT)}\ \frac{\begin{array}{c} \rho_1(l) = s_2 \quad I.id = tb \quad I' = I[id \mapsto \bot][sk \mapsto I.sk{\downarrow}_t^T] \quad \sigma' = [(I',R')/TBV] \\ R' = \{\mathcal{E}[\![t\sigma]\!]_{\rho_1} \mid \exists t' : t' \in R \wedge match(\mathcal{E}[\![T]\!]_{\rho_1}, t') \wedge \sigma = [t'/\mathcal{E}[\![T]\!]_{\rho_1}] \wedge \psi\sigma\} \end{array}}{s_1 ::_{\rho_1} \mathsf{sel\_ext}(T, \psi, tb, t, !TBV)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \ \rightarrow s_1 ::_{\rho_1} P\sigma' \mid\mid s_2 ::_{\rho_2} (I,R)}$$

$$\text{(SEL\_INT)}\ \frac{\begin{array}{c} I' = I[id \mapsto \bot][sk \mapsto I.sk{\downarrow}_t^T] \quad \sigma' = [(I',R')/TBV] \\ R' = \{\mathcal{E}[\![t\sigma]\!]_{\rho_1} \mid \exists t' : t' \in R \wedge match(\mathcal{E}[\![T]\!]_{\rho_1}, t') \wedge \sigma = [t'/\mathcal{E}[\![T]\!]_{\rho_1}] \wedge \psi\sigma\} \end{array}}{s_1 ::_{\rho_1} \mathsf{sel\_int}(T, \psi, (I,R), t, !TBV).P \ \rightarrow s_1 ::_{\rho_1} P\sigma'}$$

$$\text{(UPD)}\ \frac{\begin{array}{c} \rho_1(l) = s_2 \quad I.id = tb \\ R'_1 = \{t' \mid t' \in R \wedge \neg(\exists \sigma : match(\mathcal{E}[\![T]\!]_{\rho_1}, t') \wedge \sigma = [t'/\mathcal{E}[\![T]\!]_{\rho_1}] \wedge \psi\sigma \wedge \mathcal{E}[\![t\sigma]\!]_{\rho_1} \models I.sk)\} \\ R'_2 = \{\mathcal{E}[\![t\sigma]\!]_{\rho_1} \mid \exists t' : t' \in R \wedge match(\mathcal{E}[\![T]\!]_{\rho_1}, t') \wedge \sigma = [t'/\mathcal{E}[\![T]\!]_{\rho_1}] \wedge \psi\sigma \wedge \mathcal{E}[\![t\sigma]\!]_{\rho_1} \models I.sk\} \end{array}}{s_1 ::_{\rho_1} \mathsf{update}(T, \psi, t, tb)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow s_1 ::_{\rho_1} P \mid\mid s_2 ::_{\rho_2} (I, R'_1 \uplus R'_2)}$$

$$\text{(AGGR)}\ \frac{\begin{array}{c} t' = f(\{t \mid \exists \sigma' : t \in R \wedge match(\mathcal{E}[\![T]\!]_{\rho_1}, t) \wedge \sigma' = [t/\mathcal{E}[\![T]\!]_{\rho_1}] \wedge \psi\sigma') \\ \rho(l) = s_2 \quad I.id = tb \quad match(\mathcal{E}[\![T']\!]_{\rho_1}, t') \end{array}}{\begin{array}{c} s_1 ::_{\rho_1} \mathsf{aggr}(T, \psi, tb, f, T')@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow \\ s_1 ::_{\rho_1} P[t'/\mathcal{E}[\![T']\!]_{\rho_1}] \mid\mid s_2 ::_{\rho_2} (I,R) \end{array}}$$

$$\text{(CREATE)}\ \frac{\rho_1(l) = s_2}{s_1 ::_{\rho_1} \mathsf{create}(I)@\ell.P \mid\mid s_2 ::_{\rho_2} \mathsf{nil} \rightarrow s_1 ::_{\rho_1} P \mid\mid s_2 ::_{\rho_2} (I, \emptyset)}$$

$$\text{(DROP)}\ \frac{\rho_1(l) = s_2 \quad I.id = tb}{s_1 ::_{\rho_1} \mathsf{drop}(tb)@\ell.P \mid\mid s_2 ::_{\rho_2} (I,R) \rightarrow s_1 ::_{\rho_1} P \mid\mid s_2 ::_{\rho_2} \mathsf{nil}}$$

We proceed with a detailed explanation of the semantic rules in Table 3 that account for the execution of Klaim-DB actions and the ones in Table 4 that mainly describe the execution of the control flow constructs. In the explanation, we will avoid reiterating that each table resides in a database located at some ℓ, but directly state "table ... located at ℓ".

3.1 Semantics for Actions

Insertion and Deletion. The rule (INS) of Table 3 says that to insert a row t into a table tb at s_2, the logical locality ℓ needs to be evaluated to s_2 under the local environment ρ_1, the table identifier tb must agree with that of a destination table (I, R) already existing at s_2, and the tuple t needs to satisfy the requirements imposed by the schema $I.sk$. If these conditions are met, then the evaluated tuple is added into the data set R.

Example 2 (Adding new Shoes). The local action at $\underline{s_1}$ that inserts an entry for KLD high boots of a new color, white, sized "37", produced in 2015, with 6 in stock, is $\mathsf{insert}(T_0, \mathsf{KLD})@\mathit{self}$, where $T_0 = (\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{white}\text{"}, \text{"37"}, 6, 0)$. By (INS), we have

$$\underline{s_1} ::_{\underline{\rho_1}} \mathsf{insert}(T_0, \mathsf{KLD})@\mathit{self}.\mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1) \rightarrow\ \underline{s_1} ::_{\underline{\rho_1}} \mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1'),$$

where $R_1' = R_1 \uplus \{T_0\}$, since $\underline{\rho_1}(\mathit{self}) = \underline{s_1}$, $\mathsf{KLD} = I_1.\mathit{id}$ and $T_0 \models I_1.sk$. □

Deletion operations are performed according to the rule (DEL). A deletion can be performed if the logical locality ℓ refers to the physical locality s_2 under the local environment ρ_1, and the specified table identifier tb agrees with that of the table (I, R) targeted. The rows that do not match the pattern T or do not satisfy the condition ψ will constitute the resulting data set of the target.

Example 3 (Deleting Existing Shoes). The local action at $\underline{\ell_1}$, deleting all entries for white KLD high boots sized "37" produced in 2015, from the resulting table of Example 2, is $\mathsf{delete}(T_0, \mathsf{true}, \mathsf{KLD})@\mathit{self}$, where $T_0 = (\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{White}\text{"}, \text{"37"}, !x, !y)$.

We have the transition:

$$\underline{s_1} ::_{\underline{\rho_1}} \mathsf{delete}(T_0, \mathsf{true}, \mathsf{KLD})@\mathit{self}.\mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1') \rightarrow\ \underline{s_1} ::_{\underline{\rho_1}} \mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1).$$

This reflects that the original table is recovered after the deletion. □

Selection, Update and Aggregation. The rule (SEL_EXT) describes the way the "external" selection operations are performed. It needs to be guaranteed that the logical locality ℓ is evaluated to s_2 (under the local environment ρ_1), and the identifier tb is identical to that of the table (I, R) existing at s_2. If the conditions are met, all the rows that match the pattern T and satisfy the predicate ψ, will be put into the result data set R'. The schema of the resulting table is also updated according to the pattern T and the tuple t. The resulting table (I', R') is substituted into each occurrence of the table variable TBV used in the continuation P.

Example 4 (Selection of Shoes in a Certain Color). The Klaim-DB action performed from the head office, selecting the color, size, and sales of the types of high boots that are not red, at the local database at $\underline{s_1}$, is

$$\mathsf{sel\_ext}((\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, !x, !y, !z, !w, !p), y \neq \text{"}\mathit{red}\text{"}, \mathsf{KLD}, (y, z, p), !\mathit{TBV})@\underline{\ell_1}.$$

According to the rule (SEL_EXT), we have the transition:

$$\begin{aligned}&\underline{s_0} ::_{\underline{\rho_0}} \mathsf{sel\_ext}((\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, !x, !y, !z, !w, !p), y \neq \text{"}\mathit{red}\text{"}, \mathsf{KLD}, (y, z, p), !\mathit{TBV})@\underline{\ell_1}.\mathsf{nil}\\ &||\ \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1)\\ &\rightarrow\ \underline{s_0} ::_{\underline{\rho_0}} \mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1).\end{aligned}$$

The conditions $\rho_0(\underline{\ell_1}) = \underline{s_1}$ and $\mathsf{KLD} = I.id$ in the premises of the rule are satisfied. The I' in (SEL_EXT) is such that $I'.id = \perp$, $I'.sk =<$ "*Color*" : *String*, "*Size*" : *Int*, "*Sales*" : *Int* $>$, and I' agrees with I on the other attributes. The table variable TBV is replaced by (I', R'), for some $R' = \{($"*black*", "38", "2"$)$, $($"*black*", "37", "2"$)\}$. □

"Internal" selections are performed according to the rule (SEL_INT). No constraints concerning localities/identifiers are needed. All the rows that match the pattern T and satisfy the predicate ψ are selected into the resulting data set R'. The schema of the result table is produced by using the projection operation introduced in the beginning of this section. All occurrences of the table variable TBV in the continuation P will be replaced by the resulting table (I', R') before continuing with further transitions. We use $\perp$ for $I'.id$ to indicate that (I', R') is a "temporary" table that can be thought of as existing in the query engine, rather than in the database.

The updates of data stored in tables are executed according to the rule (UPD). Again, it is checked that the specified logical locality of the table (I, R) to be updated corresponds to the site where the table actually resides, and that the specified table identifier matches the actual one. An update goes through all the elements t' of R. This t' is updated only if it matches the template T, resulting in the substitution σ that makes the predicate ψ satisfied, and $t\sigma$ is evaluated to a tuple that satisfies the schema $I.sk$. The evaluation result of $t\sigma$ will then replace t' in the original table, which is captured by R'_2 . If at least one of the above mentioned conditions is not met, then t' will be left intact (described by R'_1).

Example 5 (Update of Shoes Information). Suppose two more red KLD high boots sized 37 are sold. The following local action informs the database at $\underline{s_1}$ of this.

$$\begin{array}{l}\mathsf{update}(\ (\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{red}\text{"}, \text{"37"}, !x, !y), \mathsf{true}, \\ \qquad\qquad (\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{red}\text{"}, \text{"37"}, x-2, y+2), \mathsf{KLD})@\mathit{self}.\end{array}$$

We have the transition:

$$\begin{array}{l}\underline{s_1} ::_{\underline{\rho_1}} \mathsf{update}((\text{"001"}, \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{red}\text{"}, \text{"37"}, !x, !y), \mathsf{true}, (\text{"001"}, \\ \text{"}\mathit{HighBoot}\text{"}, \text{"2015"}, \text{"}\mathit{red}\text{"}, \text{"37"}, x-2, y+2), \mathsf{KLD})@\mathit{self}.\mathsf{nil} \parallel \underline{s_1} ::_{\underline{\rho_1}} (I, R) \rightarrow \\ \underline{s_1} ::_{\underline{\rho_1}} \mathsf{nil} \parallel \underline{s_1} ::_{\underline{\rho_1}} (I, R'_1 \uplus R'_2).\end{array}$$

The multiset R'_1 consists of all the entries that are intact — shoes that are not red high boots sized 37, while R'_2 contains all the updated items. □

The rule (AGGR) describes the way aggregation operations are performed. The matching of localities and table identifiers is still required. The aggregation function f is applied to the multiset of all tuples matching the template T, as well as satisfying the predicate ψ. The aggregation result t' obtained by the application of f is bound to the specified template T' only if the evaluation of T' matches t'. In that case the substitution reflecting the binding is applied to the continuation P and the aggregation is completed.

Example 6 (Aggregation of Sales Figures).

The local aggregation returning the total sales of the shoes with ID "001", can be modelled by the Klaim-DB action $\mathsf{aggr}(T_0, \mathsf{true}, \mathsf{KLD}, \mathit{sum}_7, (!\mathit{res}))@\mathit{self}$, where $T_0 = (\text{“001”}, !x, !y, !z, !w, !q, !o)$, and $\mathit{sum}_7 = \lambda R.(\mathit{sum}(\{v_7 | (v_1, ..., v_7) \in R\}))$, i.e., sum_7 is a function from multisets R to unary tuples containing the summation results of the 7-th components of the tuples in R.

It is not difficult to derive:

$$\begin{array}{l} \underline{s_1} ::_{\underline{\rho_1}} \mathsf{aggr}(T_0, \mathsf{true}, \mathsf{KLD}, \mathit{sum}_7, (!\mathit{res}))@\underline{\ell_1}.\mathsf{nil} \;||\; \underline{s_1} ::_{\underline{\rho_1}} (I, R) \\ \rightarrow\; \underline{s_1} ::_{\underline{\rho_1}} \mathsf{nil} \,||\, \underline{s_1} ::_{\underline{\rho_1}} (I, R). \end{array}$$

The variable $!\mathit{res}$ is then bound to the integer value 12 $(2 + 5 + 1 + 2 + 2)$. □

Example 7 (Selection using Aggregation Results). Consider the query from $\underline{s_0}$ that selects all the colors, sizes and sales of the types of high boots produced in the year 2015, whose ID is "001" and whose sales figures are above average. This query can be modelled as a sequence of actions at $\underline{s_0}$, as follows.

$$\begin{array}{l} \underline{s_0} ::_{\underline{\rho_0}} \mathsf{aggr}(T_0, \mathsf{true}, \mathsf{KLD}, \mathit{avg}_7, !\mathit{res})@\underline{\ell_1}.\mathsf{sel\_ext}(T_0, w \geq \mathit{res}, \mathsf{KLD}, (x, y, w), !\mathit{TBV})@\underline{\ell_1}.\mathsf{nil} \\ ||\;\; \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1), \end{array}$$

where $T_0 = (\text{“001}'', \text{“}\mathit{HighBoot}\text{”}, \text{“2015”}, !x, !y, !z, !w)$ and $\mathit{avg}_7 = \lambda R.(\frac{\mathit{sum}(\{v_7 | (v_1, ..., v_7) \in R\})}{|R|})$ is a function from multisets R to unary tuples containing the average value of the 7-th components of the tuples in R. □

Adding and Dropping Tables. To create a new table, the rule (CREATE) ensures that the physical site s_2 corresponding to the logical locality ℓ mentioned in the action $\mathsf{create}(I)@\ell$ does exist. Then a table with the interface I and an empty data set is created at the specified site. It remains an issue to ascertain that there are no existing tables having the same identifier $I.id$ at the target site. This is achieved with the help of the rule (PAR) in Table 4.

To drop an existing table, the rule (DROP) checks that a table with the specified identifier tb does exist at the specified site $\rho_1(l) = s_2$. Then the table is dropped by replacement of it with nil.

3.2 Semantics for Processes and Networks

Directing our attention to Table 4, the rules $(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{tt}})$ and $(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{ff}})$ specify when and how a loop is to be executed one more time, and has finished, respectively. The rule $(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{tt}})$ says that if there are still more tuples (e.g., t_0) in the multiset R matching the pattern specified by the template T, then we first execute one round of the loop with the instantiation of variables in T by corresponding values in t_0, and then continue with the remaining rounds of the loop by removing t_0 from R. The rule $(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{ff}})$ says that if there are no more tuples in R matching T, then the loop is completed. The rules $(\mathrm{FOR}_{\mathrm{p}}^{\mathrm{tt}})$ and $(\mathrm{FOR}_{\mathrm{p}}^{\mathrm{ff}})$ can be understood similarly.

Table 4. The Semantics for Processes and Nets (Continued)

$$(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{tt}})\ \frac{t_0 \in R \quad \mathit{match}(\mathcal{E}[\![T]\!]_\rho, t_0)}{s ::_\rho \mathsf{foreach}_{\mathsf{s}}\ T\ \mathsf{in}\ R{:}P \to s ::_\rho P[t_0/\mathcal{E}[\![T]\!]_\rho]; \mathsf{foreach}_{\mathsf{s}}\ T\ \mathsf{in}\ R \setminus \{t_0\} : P}$$

$$(\mathrm{FOR}_{\mathrm{p}}^{\mathrm{tt}})\ \frac{t_0 \in R \quad \mathit{match}(\mathcal{E}[\![T]\!]_\rho, t_0)}{s ::_\rho \mathsf{foreach}_{\mathsf{p}}\ T\ \mathsf{in}\ R{:}P \to s ::_\rho P[t_0/\mathcal{E}[\![T]\!]_\rho] \mid \mathsf{foreach}_{\mathsf{p}}\ T\ \mathsf{in}\ R \setminus \{t_0\} : P}$$

$$(\mathrm{FOR}_{\mathrm{s}}^{\mathrm{ff}})\ \frac{\neg(\exists t_0 \in R : \mathit{match}(\mathcal{E}[\![T]\!]_\rho, t_0))}{s ::_\rho \mathsf{foreach}_{\mathsf{s}}\ T\ \mathsf{in}\ R : P \to s ::_\rho \mathsf{nil}} \qquad (\mathrm{FOR}_{\mathrm{p}}^{\mathrm{ff}})\ \frac{\neg(\exists t_0 \in R : \mathit{match}(\mathcal{E}[\![T]\!]_\rho, t_0))}{s ::_\rho \mathsf{foreach}_{\mathsf{p}}\ T\ \mathsf{in}\ R : P \to s ::_\rho \mathsf{nil}}$$

$$(\mathrm{SEQ}^{\mathrm{tt}})\ \frac{s ::_\rho P_1 \mid\mid N \to s ::_\rho P_1' \mid\mid N'}{s ::_\rho P_1; P_2 \mid\mid N \to s ::_\rho P_1'; P_2 \mid\mid N'} \qquad (\mathrm{SEQ}^{\mathrm{ff}})\ \frac{s ::_\rho P_1 \mid\mid N \to s ::_\rho \mathsf{nil} \mid\mid N'}{s ::_\rho P_1; P_2 \mid\mid N \to s ::_\rho P_2 \mid\mid N'}$$

$$(\mathrm{CALL})\ s ::_\rho A(\tilde{t}) \to s ::_\rho P[\tilde{v}/\tilde{x}] \quad \text{if } A(\tilde{x}) \triangleq P \wedge \mathcal{E}[\![\tilde{t}]\!]_\rho = \tilde{v}$$

$$(\mathrm{PAR})\ \frac{N_1 \to N_1'}{N_1 || N_2 \to N_1' || N_2} \quad \text{if } \mathit{Lid}(N_1') \cap \mathit{Lid}(N_2) = \emptyset$$

$$(\mathrm{RES})\ \frac{N \to N'}{(\nu s)N \to (\nu s)N'} \qquad (\mathrm{EQUIV})\ \frac{N_1 \equiv N_2 \quad N_2 \to N_3 \quad N_3 \equiv N_4}{N_1 \to N_4}$$

The rules ($\mathrm{SEQ}^{\mathrm{tt}}$) and ($\mathrm{SEQ}^{\mathrm{ff}}$) describe the transitions that can be performed by sequential compositions $P_1; P_2$. In particular, the rule ($\mathrm{SEQ}^{\mathrm{tt}}$) accounts for the case where P_1 cannot finish after one more step; whereas the rule ($\mathrm{SEQ}^{\mathrm{ff}}$) accounts for the opposite case. We require that no variable bound in P_1 is used in P_2, thereby avoiding the need for recording the substitutions resulting from the next step of P_1, that need to be applied on P_2.

The rule (PAR) says that if a net N_1 can make a transition, then its parallel composition with another net N_2 can also make a transition, as long as no clashes of table identifiers local to each site in $N_1||N_2$ are introduced. It makes use of the function $\mathit{Lid}(...)$ that gives the multiset of pairs of physical localities and table identifiers in networks and components. This function is overloaded on networks and components, and is defined inductively as follows.

$$\begin{array}{ll}
\mathit{Lid}(\mathsf{nil}) = \emptyset & \mathit{Lid}(s, P) = \emptyset \\
\mathit{Lid}(N_1||N_2) = \mathit{Lid}(N_1) \uplus \mathit{Lid}(N_2) & \mathit{Lid}(s, (I, R)) = \{(s, I.\mathit{id})\} \\
\mathit{Lid}((\nu s)N) = \mathit{Lid}(N) & \mathit{Lid}(s, C_1|C_2) = \mathit{Lid}(s, C_1) \uplus \mathit{Lid}(s, C_2) \\
\mathit{Lid}(s ::_\rho C) = \mathit{Lid}(s, C) &
\end{array}$$

The rules (CALL), (RES) and (EQUIV) are self-explanatory.

It is a property of our semantics that no local repetition of table identifiers can be caused by a transition (including the creation of a new table). Define $\mathit{no\_rep}(A) = (A = \mathsf{set}(A))$, which expresses that there is no repetition in the multiset A, thus A coincides with its underlying set. This property is formalized in Lemma 1 whose proof is straightforward by induction on the derivation of the transition.

Lemma 1. *For all nets N and N', if $no\_rep(Lid(N))$ and $N \to N'$, then it holds that $no\_rep(Lid(N'))$.*

Thus, imposing "non-existence of local repetition of table identifiers" as an integrity condition for the initial network will guarantee the satisfaction of this condition in all possible derivatives of the network.

Example 8 (Transition of Networks). Continuing with Example 2, we illustrate how our semantics help establish global transitions.

Suppose there is no other table at $\underline{s_1}$ with the identifier KLD. By (PAR), and the structural congruence, we have

$$\begin{aligned} &\underline{s_0} ::_{\underline{\rho_0}} (I_0, R_0)|C'_0 \mid\mid \underline{s_1} ::_{\underline{\rho_1}} (I_1, R_1)|\mathsf{insert}(t_1, \mathsf{KLD})@\mathit{self}.\mathsf{nil}|C''_1 \mid\mid \dots \mid\mid \underline{s_n} ::_{\underline{\rho_n}} (I_n, R_n)|C'_n \\ &\to \underline{s_0} ::_{\underline{\rho_0}} (I_0, R_0)|C'_0 \mid\mid \underline{s_1} ::_{\underline{\rho_1}} (I_1, R'_1)|C''_1 \mid\mid \dots \mid\mid \underline{s_n} ::_{\underline{\rho_n}} (I_n, R_n)|C'_n, \end{aligned}$$

where $t_1 = (\text{“001”}, \text{“}\mathit{HighBoot}\text{”}, \text{“2015”}, \text{“}\mathit{white}\text{”}, \text{“37”}, \text{“6”}, \text{“0”})$. □

4 Case Study

Continuing with our running example, we illustrate the modelling of data aggregation over multiple databases local to different branches of the department store chain in Klaim-DB. In more detail, a manager of the head office wants statistics on the total sales of KLD high boots from the year 2015, in each branch operating in Copenhagen.

We will think of a procedure *stat* at the site s_0 of the head office, carrying out the aggregation needed. Thus the net for the database systems of the department store chain, as considered in Section 2, specializes to the following, where C''_0 is the remaining tables and processes at s_0 apart from Stores and *stat*.

$$\underline{s_0} ::_{\underline{\rho_0}} (\mathit{stat}|(I, R)|C''_0) \mid\mid \underline{s_1} ::_{\underline{\rho_1}} ((I_1, R_1)|C'_1) \mid\mid \dots \mid\mid \underline{s_n} ::_{\underline{\rho_n}} ((I_n, R_n)|C'_n)$$

A detailed specification of the procedure *stat* is then given in Figure 4.

```
$stat \triangleq$ $\mathsf{create}(I_{\text{res}})@\mathit{self}.$
        $\mathsf{sel\_ext}((!x, !y, !z, !w, !p), \mathsf{KLD} \in w \wedge x = \text{“}\mathit{CPH}\text{”}, \mathit{Stores}, (z, p), !\mathit{TBV})@\mathit{self}.$
        $\mathsf{foreach_p}\ (!q, !u)\ \mathsf{in}\ \mathit{TBV}.R :$
            $\mathsf{aggr}((\text{“001”}, \text{“}\mathit{HighBoot}\text{”}, \text{“2015”}, !x, !y, !z, !w), \mathsf{true}, \mathsf{KLD}, \mathit{sum}_7, (!\mathit{res}))@u.$
            $\mathsf{insert}((q, \text{“}\mathit{HighBoot}\text{”}, \mathit{res}), \mathsf{result})@\mathit{self};$
        ...
        $\mathsf{drop}(\mathsf{result})@\mathit{self}$
```

Fig. 4. The Procedure for Distributed Data Aggregation

First of all, a result table with the interface I_{res} is created, where $I_{\text{res}}.id = \mathsf{result}$ and $I.sk =< \text{“}\mathit{Brand}\text{”} : \mathit{String}, \text{“}\mathit{City}\text{”} : \mathit{String}, \text{“}\mathit{Shop\_name}\text{”} : \mathit{String},$

"*Sales*" : *Int* >. Then all the logical localities of the local databases used by the branches in Copenhagen that actually sell KLD shoes are selected, together with the shop names of such branches. This result set is then processed by a parallel loop. The number of KLD high boots from 2015 that are sold is counted at each of these localities (branches), and is inserted into the resulting table together with the corresponding shop name and the information "High Boot" describing the shoe type concerned. The resulting table, displayed in Figure 5, can still be queried/manipulated before being dropped.

In the end of this case study, we would like to remark that the use of the parallel loop in carrying out all the individual remote aggregations has made the overall query more efficient. Hence some performance issues can be captured by modelling in Klaim-DB.

Shop_name	*Shoe_name*	*Sales*
Shop1	HighBoot	12
Shop2	HighBoot	53
Shop3	HighBoot	3
...	...	...

Fig. 5. The Table result

5 Extension and Discussion

Joining Tables. Our modelling language can be extended to support querying from the "join" of tables. For this extension, we make use of under-specified join expressions *je*, that can have table identifiers or table variables as parameters. The syntax of the new selection actions is shown below.

$$\begin{aligned} a ::= ... &\mid \mathsf{sel\_ext}(T, \psi, je(tb_1, ..., tb_n), t, !TBV)@\ell_1, ..., \ell_n \\ &\mid \mathsf{sel\_int}(T, \psi, je(TBV_1, ..., TBV_n), t, !TBV') \mid ... \end{aligned}$$

For an external selection, we allow to use a list $\ell_1, ..., \ell_n$ of localities (abbreviated as $\overline{\ell}$), to join tables located in multiple databases. In more detail, for $i \in \{1, ..., n\}$, tb_i is supposed to be the identifier of a table located at ℓ_i.

We use $[\![je]\!]^{\mathrm{R}}$ and $[\![je]\!]^{\mathrm{I}}$ to represent the interpretation of *je* in terms of how the datasets and the schemas of the joined tables should be combined. As an example, a *plain list* of table identifiers or tables (substitution of table variables) corresponds to taking the concatenation of all the schemas and selecting from the cartesian product of all the data sets.

$$[\![je]\!]^{\mathrm{I}}(I_1, ..., I_n) = \bigoplus_{j \in \{1,...,n\}} I_j.sk \qquad [\![je]\!]^{\mathrm{R}}(R_1, ..., R_n) = R_1 \times ... \times R_n$$

The pattern matching against the template T and the satisfaction of the predicate ψ will be examined on tuples from $[\![je]\!]^{\mathrm{R}}(R_1, ..., R_n)$, and the predicate ψ can now impose constraints on the fields from different tables.

The adaption needed for the semantics is fairly straightforward. The semantic rules (SELJ_EXT) and (SELJ_INT) that describe selection operations from joined tables are presented in Table 5. In the rule (SELJ_EXT), although it is stipulated that the j-th table identifier specified in the list $\overline{tb}$ must be identical to the j-th table (I_j, R_j) listed as parallel components, no undesired stuck configurations are caused because of the structrual congruence.

Table 5. The Semantic Rules for Selection from Joins

$$\text{(SELJ\_EXT)}\ \frac{\begin{array}{l} n = |\overline{tb}| = |\overline{\ell}| \wedge \forall j \in \{1, ..., n\} : \rho_0(l_j) = s_j \wedge tb_j = I_j.id \\ I' = [id \mapsto \bot][sk \mapsto (\llbracket je \rrbracket^{\mathrm{I}}(\overline{I}))\downarrow_t^T] \qquad \sigma' = [(I', R')/TBV] \\ R' = \{\mathcal{E}\llbracket t\sigma \rrbracket_{\rho_1} \mid \exists t' : t' \in \llbracket je \rrbracket^{\mathrm{R}}(\overline{R}) \wedge match(\mathcal{E}\llbracket T \rrbracket_{\rho_1}, t') \wedge \sigma = (t'/\mathcal{E}\llbracket T \rrbracket_{\rho_1}) \wedge \psi\sigma\} \end{array}}{\begin{array}{l} s_0 ::_{\rho_0} \mathsf{sel\_ext}(T, \psi, je(\overline{tb}), t, !TBV)@\overline{\ell}.P \mid\mid s_1 ::_{\rho_1} (I_1, R_1) \mid\mid ... \mid\mid s_n ::_{\rho_n} (I_n, R_n) \\ \quad \to s_0 ::_{\rho_0} P\sigma' \mid\mid s_1 ::_{\rho_1} (I_1, R_1) \mid\mid ... \mid\mid s_n ::_{\rho_n} (I_n, R_n) \end{array}}$$

$$\text{(SELJ\_INT)}\ \frac{\begin{array}{l} I' = [id \mapsto \bot][sk \mapsto (\llbracket je \rrbracket^{\mathrm{I}}(\overline{I}))\downarrow_t^T] \ \ \sigma' = [(I', R')/TBV] \\ R' = \{\mathcal{E}\llbracket t\sigma \rrbracket_{\rho_1} \mid \exists t' : t' \in \llbracket je \rrbracket^{\mathrm{R}}(\overline{R}) \wedge match(\mathcal{E}\llbracket T \rrbracket_{\rho_1}, t') \wedge \sigma = (t'/\mathcal{E}\llbracket T \rrbracket_{\rho_1}) \wedge \psi\sigma\} \end{array}}{s_1 ::_{\rho_1} \mathsf{sel\_int}(T, \psi, je(\overline{(I, R)}), t, !TBV).P \ \to s_1 ::_{\rho_1} P\sigma'}$$

This extension paves the way for the general ability to operate on multiple databases by a single action, which is in line with the design philosophy of multi-database systems (e.g., [5]).

Discussion. We could have required whole tables to be received into local variables by using the standard Klaim actions $\mathsf{out}(t)@\ell$ and $\mathsf{in}(t)@\ell$, and made the selection and aggregation operations work only on table variables. In this way we could have gotten rid of "external selection". However, "external selection" on remote localities can potentially reduce the communication cost considerably, since only one tuple (for aggregation) or a part of a table (for selection) need to be returned. For selection the reduction is particularly meaningful when the resulting data set is small.

Concerning the result of selection operations, an alternative that we have not adopted is the direct placement of the result in a separate table. This table is either created automatically by the selection operation itself, with an identifier specified in the selection action, or a designated "result table" at each site. However, a problem is that the removal of the automatically created tables will need to be taken care of by the system designer making the specification, using $\mathsf{drop}(I)@\ell$ actions. And similar problems arise with the maintenance of the designated "result table" local to each site (e.g., the alteration of its schema, the cleaning of old results, etc.). To abstain from these low-level considerations, table variables are introduced and binding is used for the selection results.

The *interoperability* between database systems and ordinary applications can also be realized by bringing back the primitive Klaim actions ($\mathsf{out}(t)@\ell$, $\mathsf{in}(T)@\ell$, and $\mathsf{eval}(P)@\ell$) and allowing the co-existence of tables and plain tuples at different localities. Via the re-introduction of the eval action, we would also be able to send out mobile processes to perform complex data processing on-site.

6 Conclusion

We have proposed Klaim-DB — a modelling language that borrows insights from both the coordination language Klaim and database query languages like SQL, to support the high-level modelling and reasoning of distributed database applications. The semantics is illustrated with a running example on the query and management of the databases used by a department store chain. Data aggregation across the geographically scattered databases at the individual stores, as performed by a coordinator, is then modelled in the language. In the model, the local aggregations at the store-owned databases are performed in parallel with each other, benefiting the performance.

Our use of templates in the query actions of Klaim-DB is in line with the spirit of the QBE language (Query by Example [12]). The choice of using multiset operations for the semantics of these actions, on the other hand, has the flavor of the Domain Relational Calculus underlying QBE. The work presented in [9] discusses typical coordination issues in distributed databases in Linda, from an informal, architectural viewpoint. This work is marginally related to ours.

The specification and enforcement of security policies is an important concern in distributed database systems. A simple example is: when inserting data into a remote table, it is important to know whether the current site trusts the remote site with respect to confidentiality, and whether the remote site trusts the current site with respect to integrity. Recently, [3] and [7] address privacy and information flow issues in database applications. Another work, [10], provides an information flow analysis for locality-based security policies in Klaim. By elaborating on the language design, we provide in this paper a solid ground for any future work aiming to support security policies and mechanisms.

Another interesting line of future work is the specification of transactions (e.g., [6]). This is needed for the modelling of finer-grained coordination between database accesses.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable feedback. Xi Wu is partly supported by the IDEA4CPS project, ECNU Project of Funding Overseas Short-term Studies, Domestic Academic Visit and NSFC Project (No.61361136002 and No.61321064). Ximeng Li is partly funded by SESAMO, a European Artemis project.

References

1. Abrial, J.-R.: Formal methods: Theory becoming practice. J. UCS 13(5), 619–628 (2007)
2. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. (1998)
3. Dwork, C.: Differential privacy. In: Encyclopedia of Cryptography and Security, 2nd edn., pp. 338–340 (2011)
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
5. Kuhn, E., Ludwig, T.: Vip-mdbs: A logic multidatabase system. In: Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems, DPDS 1988, pp. 190–201. IEEE Computer Society Press (1988)
6. Kühn, E., Elmagarmid, A.K., Leu, Y., Boudriga, N.: A parallel logic language for transaction specification in multidatabase systems. Journal of Systems Integration 5(3), 219–252 (1995)

7. Lourenço, L., Caires, L.: Information flow analysis for valued-indexed data security compartments. In: Trustworthy Global Computing - 8th International Symposium, TGC 2013, pp. 180–198 (2013)
8. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61:17–139 (2004)
9. Thirukonda, M.M., Menezes, R.: On the use of linda as a framework for distributed database systems (2002)
10. Tolstrup, T.K., Nielson, F., Hansen, R.R.: Locality-based security policies. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2006. LNCS, vol. 4691, pp. 185–201. Springer, Heidelberg (2007)
11. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)
12. Zloof, M.M.: Query by example. In: Proceedings of the National Computer Conference and Exposition, AFIPS 1975, pp. 431–438. ACM, New York (1975)

A Multiset Notation

We use $\uplus$, $\cap$ and $\setminus$ to represent the union, intersection and substraction, respectively, of multisets. For a multiset S, and an element s, the application $M(S,s)$ of the multiplicity function M gives the number of repetitions of s in S. Note that for $s \notin S$, $M(S,s) = 0$. Then our notions of union, intersection and subtraction are such that

$$\begin{aligned} M(S_1 \uplus S_2, s) &= M(S_1, s) + M(S_2, s) \\ M(S_1 \cap S_2, s) &= min(M(S_1, s), M(S_2, s)) \\ M(S_1 \setminus S_2, s) &= abs(M(S_1, s) - M(S_2, s)) \end{aligned}$$

Here $abs(v)$ gives the absolute value of the integer v.