Process Algebra-Based Query Workflows

Thomas Hornung[1], Wolfgang May[2], and Georg Lausen[1]

[1] Institut für Informatik, Universität Freiburg
{hornungt,lausen}@informatik.uni-freiburg.de
[2] Institut für Informatik, Universität Göttingen
may@informatik.uni-goettingen.de

Abstract. In this paper we combine ideas from workflow processing and database query answering. Tailoring process algebras like Milner's *Calculus of Communicating Systems (CCS)* to relational dataflow makes them a natural candidate for specifying data-oriented workflows in a declarative way. In addition to the classical evaluation of relational operator trees, the combination with the CCS control structures provides (guarded) alternatives and test-based iterations using recursive process fragment definitions. For the actual atomic constituents of the process, language concepts from the relational world, like queries, but also the use of abstract datatypes, e.g., graphs, can be embedded.

We illustrate the advantages of the approach by an application scenario with remote, heterogeneous sources and Web Services that return their results asynchronously. The presented approach has been implemented in a prototype.

1 Introduction

Most of the information that is needed for daily tasks is available on the Web. The main problem is often not to *get* the information, but to process it efficiently and appropriately in an automatic way. Efficiency does not necessarily mean millions of data items, but often a relatively small number of items, scattered over multiple data sources, and to organize the process of combining, evaluating, making decisions, interacting. Consider for example travel planning: not only the nearest airport to a certain destination has to be found, but depending on the airlines, different airports must be considered, and availability of the flights has to be checked. Then, transportation from/to the airports, possibly provided by local railway companies, has to be arranged. Even employees of travel agencies usually process such enquiries manually, which requires a lot of time and is potentially incomplete and suboptimal. Although the manual process follows a small number of common patterns (e.g., searching for paths in a transitive relationship distributed over several sources, like flight schedules and train schedules, with heuristics for bridging long distances vs. shorter distances, making prereservations, doing backtracking) it is hard to automatize it since the sources are not integrated, and the underlying formalism has to cover both procedural tasks and data manipulation tasks. Often it is easier to design the process *how* to solve

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 440–454, 2009.
© Springer-Verlag Berlin Heidelberg 2009

such a problem than stating a single query. Furthermore, most of the data is not immediately available for querying via e.g. query languages like SQL or XQuery, but kept in the *Deep Web*, which consists of dynamically generated result pages, which can only be queried interactively via Web forms.

This technical environment together with the intrinsic complexity of the tasks requires for *flexible* data workflows using a *generic* data model and an extensible set of functional modules, including the ability to interact *actively* with remote services. Important basic functionality includes appropriate mechanisms to deal with *information acquisition* and target-driven *information processing* on a high level, like using design patterns for acting on graph-structured domains.

In the following, we present and discuss an approach that attempts to satisfy the above requirements. The core aspects are the intertwined description of the control flow of the process (by a process algebra, e.g., CCS [14]) and the handling of the dataflow (based on the relational model), and the use of heterogeneous atomic constituents like queries and actions in the workflow: CCS is extended to relational dataflow, called RelCCS, and realized as a language in the *MARS (Modular Active Rules for the Semantic Web)* framework [13] for embedding heterogeneous component languages. RelCCS is complementary to the original rule-based MARS paradigm, and employs the functionality of the MARS framework as an infrastructure.

The focus is not on performance, but on the qualitative ability to express and execute complex workflows and decision processes in a reasonable time – i.e., to replace hours of interactive human Web search by an unsupervised automated process that also may take hours but finally results in one or more proposals, including the optimal one.

The process design/programming in the RelCCS language is not expected to be done by casual users, but by skilled process designers in cooperation with domain experts – analogously to application database design.

Application Scenario. Consider the scenario to find either the cheapest or shortest (in terms of total time spent travelling) route to a given location (e.g., for a conference travel) or a combination of both. Human, manual search usually employs some kind of intuitive search strategy. Roughly, the strategy is to try to cover as much distance as possible by plane (assuming the distance is above a certain threshold), and then bridge the remaining distance by train or bus; if this fails, do backtracking. One usually starts with considering a known set of airports near the hometown. This shows that human problem solving, although always considering one possibility (=tuple) at a time, is inherently based on a set-oriented model.

With the means of the presented approach, such tasks can be formulated as data workflows. The backtracking is here replaced by breadth-first search, where the search space is explored stepwise and pruned based on intermediate results.

The actual process can thus be described as (i) determining a set of local airports, e.g., the ten nearest airport to a place, (ii) computing all connections from the starting point to each of these airports, (iii, in parallel) trying to find connections from each of the airports to the destination, and *joining* the results

from (ii) and (iii); always under consideration of arrival and departure times and required time for changing. While for train connections, sources usually are able to return transitive connections, flight portals only return transitive connections over the flights of the same airline. Thus, here an actual graph exploration and search is to be applied.[1]

The expected answer is the *set* of k best alternatives (wrt. a weighted function of price and duration), where each solution contains the actual connection data (flight and train numbers, departure/arrival times). Furthermore, it should in general be possible to extend the process specification in such a way that the best available one is actually booked automatically.

Structure of the paper. Section 2 introduces the RelCCS language. In Section 3, we illustrate the use of the approach by implementing the above example workflow. Related work is discussed in Section 4 before we draw a short conclusion.

2 RelCCS: The Relational Dataflow Process Language

RelCCS is a variant tailored to relational dataflow of the well-known *Calculus of Communicating Systems (CCS)* process algebra [14]. It has been designed as a part of the *MARS (Modular Active Rules for the Semantic Web)* framework [13] whose central metaphor is a model and an architecture for active rules and processes that use *heterogeneous* event, query, and action languages. This distinctive feature of MARS proves useful in the present paper, too: it allows to embed sublanguages for queries and even supplementary generic data structures via APIs expressed as actions and queries into the workflows to be specified.

Here we present MARS only as far as it is necessary to get the ideas that are relevant for the realization of RelCCS. The MARS meta-model distinguishes *rules* (not relevant in this paper), *events* (that may also occur in CCS workflows as described in this paper), *queries*, *tests*, and *actions/processes* (cf. Section 2.1); the dataflow is based on sets of tuples of variable bindings (like in Datalog; cf. Section 2.2). The MARS meta-language concept relies on an XML markup for nested expressions of *different* languages throughout whole MARS.

2.1 The Process Model: Processes and Their Constituents

The CCS Process Algebra. Processes can formally be described by process algebras; the most prominent ones are *CCS – Calculus of Communicating Systems*

[1] Experiences with conference travels showed that real travel agencies are often challenged with finding the potential nearest airports to rather unknown destinations (e.g., St.Malo/France), and are rather weak in finding non-direct flight connections using different airlines (e.g., Lufthansa + AirFrance) or via non-expected intermediate airports (via Stansted to reach Dinard/France), or surprising connections (fly to Jersey Island and take the ferry to St.Malo) – actually, ferries are contained in the railway portals, so it is not necessary to find out about individual ferry lines. The latter shows also that it would not be advantageous to try to save time by predefining the set of destination airports by the user, but to use a fully algorithmic search that is not biased in any way.

[14] and *CSP – Communicating Sequential Processes* [10]; we chose CCS as the base to develop RelCCS. A CCS algebra with a carrier set $\mathcal{A}$ (its atomic constituents) is defined as follows (we consider here the asynchronous variant of CCS that allows for implicit delays), using a set of process variables:

With $a \in \mathcal{A}$, X a process variable, P and Q process expressions, $X := P$ is a process definition, a, X, $a.P$ (prefixing; sequential composition), (P, Q) (sequential composition), $P|Q$ (concurrent composition), and P_1+P_2 (alternative composition; generally written as $\sum_{i \in I} P_i$ for a set I of indexes) are process expressions. The semantics is defined in [14] by transition rules that immediately induce an implementation strategy. By carrying out an action, a process changes into another process:

$$a.P \xrightarrow{a} P \;,\; \frac{P \xrightarrow{a} P'}{(P,Q) \xrightarrow{a} (P',Q)} \;,\; \frac{P_i \xrightarrow{a} P}{\sum_{i \in I} P_i \xrightarrow{a} P} \text{ (for } i \in I) \;,$$
$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q} \;,\; \frac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'} \;,\; \frac{X := P \quad P \xrightarrow{a} P'}{X \xrightarrow{a} P'} \;.$$

Note that prefixing $a.Q$ is actually a special case of sequence (P, Q) where P is atomic. While in CCS, the state of a process is encoded in its behavior (via the possible actions), we generalize the definition to processes with an explicit state described by sets of tuples of variable bindings in Sections 2.2 and 2.3.

Atomic Constituents. While in the basic formalism of CCS, all atomic constituents are considered to be actions, in our approach, atomic constituents are event specifications, queries, tests, and atomic actions:

- atomic actions: these are actually executed as actions, e.g., by Web Services;
- event specifications as atomic constituents: executing an event specification means to wait for an occurrence of the specified event, incorporate the results in the state of the process, and then continue;
- executing a query means to evaluate the query, incorporate the results in the state of the process, and continue the process;
- executing a test means to evaluate it, and incorporate the results in the state of the process, and continue appropriately.

The approach is parametric in the languages used for expressing the constituents. Users write their processes in RelCCS, embedding atomic constituents in sublanguages of their choice. While the semantics of RelCCS provides the global semantics, the constituents are handled by specific services that implement the respective languages.

2.2 State, Communication, and Data Flow via Variable Bindings

The state of a process, and the dataflow through the process and to/from the processors of the constituents is provided by *logical variables* in the style of deductive rules, production rules etc.: The state of the computation of a process

is represented by a set of tuples of variable bindings, i.e., every tuple is of the form $t = \{v_1/x_1, \ldots, v_n/x_n\}$ with $v_1, \ldots, v_n$ variables and $x_1, \ldots, x_n$ elements of the underlying domain (which is in our case the set of strings, numbers, and XML literals). Thus, for given active variables $v_1, \ldots, v_m$, such a state can be seen as a relation whose attributes are the names of the variables. We denote a process expression P to be executed in a current state R by $P[R]$.

By that, the approach does only minimally constrain the embedded languages. For instance, all paradigms of query languages, following a functional style (such as XPath/XQuery), a logic style (such as Datalog or SPARQL [17]), or both (F-Logic [11]) can be used. The semantics of the event part (that is actually a "query" against an event stream that is evaluated incrementally) is –from that aspect– very similar, and the action part takes a set of tuples of variable bindings as input.

2.3 Syntax and Semantics of RelCCS

RelCCS combines the constructs of CCS with relational data flow. Syntactically, it uses mnemonic names (which are also used in its XML markup) instead of the CCS symbol operators.

Let $\mathcal{P}$ denote the set of process expressions, let $\mathbb{V}$ denote the set of variable names. For a given finite set $\mathsf{Var} \subseteq \mathbb{V}$, $\mathsf{Tuples(Var)}$ denotes the set of possible tuples over Var. As usual, $2^{\mathsf{Tuples(Var)}}$ denotes the set of sets of tuples over Var. A given set R of tuples of variable bindings is thus an element $R \in 2^{\mathsf{Tuples(Var)}}$.

The mapping $[\![\ ]\!] : \mathcal{P} \times 2^{\mathsf{Tuples(Var)}} \rightarrow 2^{\mathsf{Tuples(Var)}}$ specifies the formal semantics by mapping a process expression $P \in \mathcal{P}$ and a set R of tuples of variable bindings to the set $[\![P[R]]\!]$ of tuples of variable bindings that result from execution of a process P for an initial state R. The definition of this denotational semantics $[\![P[R]]\!]$ by structural induction over $\mathcal{P}$ is given below.

Example 1. *Consider a simple query q whose answers are all pairs (c, b) such that c is a country and b is a city in c with more than one million inhabitants:*

$$[\![q(\{\{c/\text{"}\mathsf{Germany}\text{"}\}, \{c/\text{"}\mathsf{Austria}\text{"}\}, \{c/\text{"}\mathsf{Switzerland}\text{"}\}, \{c/\text{"}\mathsf{Joe}\text{"}\}\})]\!] = \{\{c/\text{"}\mathsf{Germany}\text{"}, b/\text{"}\mathsf{Berlin}\text{"}\}, \{c/\text{"}\mathsf{Germany}\text{"}, b/\text{"}\mathsf{Hamburg}\text{"}\}, \{c/\text{"}\mathsf{Germany}\text{"}, b/\text{"}\mathsf{Munich}\text{"}\}, \{c/\text{"}\mathsf{Austria}\text{"}, b/\text{"}\mathsf{Vienna}\text{"}\}\}$$

There is no resulting tuple for "Switzerland", because there are no cities with more than one million inhabitants in Switzerland, and there is no resulting tuple for "Joe" since "Joe" is not a country at all. On the other hand, answer tuples to q like $\{c/\text{"}\mathsf{France}\text{"}, b/\text{"}\mathsf{Paris}\text{"}\}$ do not belong to the result because their value for c does not match any value of c of the initial tuples.

Note that $[\![\ ]\!]$ is just the declarative semantics that does neither depend on, nor prescribe the operational details of actual evaluation: $q[R]$ may be answered by computing $R \bowtie \sigma[\mathsf{population} > 1000000](\mathsf{City})$ for a suitable database relation City, or iteratively a Deep Web query q' can be stated for every country, yielding e.g. $q'(\text{"}\mathsf{Germany}\text{"}) = \{\text{"}\mathsf{Berlin}\text{"}, \text{"}\mathsf{Hamburg}\text{"}, \text{"}\mathsf{Munich}\text{"}\}$ and generating the result set incrementally from the answers.

The situation is similar to the definition of the formal semantics of the relational algebra, and actual query optimization and evaluation.

Atomic Constituents. For atomic constituents p, $[\![p[R]]\!]$ extends R (Queries, Events), restricts R (Tests), or (Actions) just uses R and leaves it unchanged:

- Actions: executing $\mathsf{Action}(a)[R]$ means to execute a for each tuple in R without changing the state R. $[\![\mathsf{Action}(a)[R]]\!] := R$, plus external side effects of a.
- $\mathsf{Query}(q)[R]$: R is used to provide the *input parameters* to the query q. A query q can be seen as a predicate q_0 (its *characteristic predicate*, which contains all input/output mappings) over variables $\overline{qv} = \{qv_1, \ldots, qv_n\}$, from which some variables $\overline{qin} = \{qin_1, \ldots, qin_k\} \subseteq \{qv_1, \ldots, qv_n\}$ act as input variables, the others $\overline{qout} = \{qout_1, \ldots, qout_m\} = \overline{qv} \setminus \overline{qin}$ act as output variables. Given a tuple $t \in R$, the input tuple for q is $t_q := \pi[\overline{qin}](t)$[2] and $[\![\mathsf{Query}(q)[t_q]]\!] := \{t' \in q_0 : t_q \subseteq t'\}$. With this, let $[\![\mathsf{Query}(q)[t]]\!] := \{t\} \bowtie [\![\mathsf{Query}(q)[t_q]]\!]$, and analogously, $[\![\mathsf{Query}(q)[R]]\!] := \bigcup_{t \in R} [\![\mathsf{Query}(q)[t]]\!] = R \bowtie\ q_0$.
- $\mathsf{Test}(c)[R]$: the tuples $r \in R$ that satisfy the test survive: $[\![\mathsf{Test}(c)[R]]\!] = \sigma[c](R)$, like SQL's `SELECT * FROM R WHERE` c.
  Optionally, the test can be parameterized with a quantifier $(\mathsf{exists}|\mathsf{notExists}|\mathsf{all})$ where the whole set R of tuples is taken and if one, none, or all tuples $t \in R$ satisfy the test, the result is R, otherwise $\emptyset$. E.g., $[\![\mathsf{Test[exists]}(x = 3)[R]]\!] = R$ if for some tuple t in R, the value of the variable x is 3, otherwise $[\![\mathsf{Test[exists]}(x = 3)[R]]\!] = \emptyset$.
- $\mathsf{Event}(ev)[R]$ is analogous to queries: for each tuple of R, events matching the given event specification are caught and the variable bindings are appropriately extended. For the present application for query answering, events actually play a minor role; they can be used for designing complex workflows manually. Here we just give the semantics for the sake of completeness:
  Given an event occurrence ev_0 that matches the event specification ev for a certain tuple $t \in R$ resulting in a set of tuples $ev_0(t)$, $[\![\mathsf{Event}(ev)[R]]\!]$ contains $R \bowtie ev_0(t)$ (the actual semantics of "matching" depends on the embedded event specification language). Thus, at a given timepoint τ, $[\![\mathsf{Event}(ev)[R]]\!] = R \bowtie \{ev_0(t) : ev_0$ occurred between "starting" $ev[R]$ and $\tau\}$.

Operators. For every evaluation $P[R]$, the set R of initial tuples is modified by executing the process P, resulting in a new relation $[\![P[R]]\!]$ as "outcome" that is returned to the superior process.

- Prefixing, Sequence: execute $\mathsf{Seq}(P, Q)[R]$ by executing $P[R]$, yielding R', and then execute $Q[R']$. This "common" interpretation of sequence builds actually upon the inner join: $[\![\mathsf{Seq}(P, Q)[R]]\!] := [\![Q[[\![P[R]]\!]]]\!]$. More explicitly, $[\![\mathsf{Seq}(P, Q)[R]]\!] = [\![P[R]]\!] \bowtie [\![Q[[\![P[R]]\!]]]\!]$ (analogously, $[\![\mathsf{Seq}(P_1, \ldots, P_n)[R]]\!]$ is defined inductively).

[2] As usual, π, σ, ρ denote relational projection, selection, and renaming.

As a more general idea, tailored to (more accidentally sequential) evaluation of queries, instead of $\bowtie$, also left/right/full outer joins make sense, and even a modified form of relational difference as negation: For that, we parameterize Seq as Seq[join] (default), Seq[(left|right|full)-outer-join], and Seq[minus]. E.g., the semantics of $\mathsf{Seq[minus]}(P_1, \ldots, P_n)$ is defined as follows: Assume $R_i := [\![\mathsf{Seq[minus]}(P_1, \ldots, P_i)[R]]\!]$ after step i (for $i = 0$: $R_0 := R$) and $S := [\![P_{i+1}[R_i]]\!]$ of step i+1, let $[\![\mathsf{Seq[minus]}(P_1, \ldots, P_{i+1})[R]]\!] := R_i \setminus (R_i \ltimes S)$. For example, the query $q_1(A, B, X) \wedge \neg\exists C, Y : q_2(B, C, Y)$ can be evaluated as $\mathsf{Seq[minus]}(\mathsf{Query}(q_1(A, B, X)), \mathsf{Query}(q_2(B, C, Y)))$.

- $\mathsf{Alternative}(P_1, \ldots, P_n)[R]$ and $\mathsf{Union}(P_1, \ldots, P_n)[R]$: each branch is started with R.

  For the (full) union, the result tuples of an alternative or union are the union $R_1 \cup \ldots \cup R_n$ of the results of its branches, $[\![\mathsf{Union}(P_1, \ldots, P_n)[R]]\!] = [\![P_1[R]]\!] \cup \ldots \cup [\![P_n[R]]\!]$.

  For the alternative, the following operational restriction holds: All branches have to be guarded, i.e., before the first *action* is executed, a test must be executed (optionally preceded by queries to obtain additional information). For instance, in $\mathsf{Alternative}(\mathsf{Seq}(\mathsf{Test}(c), P_1), \mathsf{Seq}(\mathsf{Test}(\neg c), P_2))$, all tuples that satisfy c will actually run through the first branch, and the others run through the second branch.

  If the guards of the branches are exclusive, the alternative is equivalent to the union. If the guards are non-exclusive, the actual outcome is nondeterministic: for each tuple t, the quicker branch will preempt the others, and exclusively contribute $[\![P_i[t]]\!]$ to the result.
- $\mathsf{Concurrent}(P_1, \ldots, P_n)[R]$: each branch is started with R. The result is $[\![\mathsf{Concurrent}(P_1, \ldots, P_n)[R]]\!] := [\![P_1[R]]\!] \bowtie \ldots \bowtie [\![P_n[R]]\!]$, i.e., *each* tuple runs through *all* branches (possibly being extended with further variables), and the results are joined. Note that if a tuple is removed in some branch, it will not occur at all in the result.

  Like for sequences, the operator is also parameterized: in addition to $\bowtie$, left/right/full outer join and relational difference are also allowed.

Complete vs. Partial Answers. An intuitive and simple model is that the whole set of tuples proceeds synchronously through the process, like the view on relational algebra when taught in courses. The actual execution also covers asynchronous remote services and even *partial* answers, where services return tuples that can be computed quickly, and later send back further tuples.

2.4 Recursive Processes in RelCCS

Recursive processes extend the expressiveness from that of relational algebra (trees) to that of recursive Datalog, which e.g. allows to compute transitive closure. Recursive processes are defined by (i) giving and naming a *process definition*, and (ii) then *using* this definition somewhere in the process/tree.

Since logical variables can be bound only once, variables that are bound to different values in each iteration must be considered to be *local* to the current

iteration. They can be bound either when starting the process, or in some step inside the process. Only the final result is then bound to the actual logical variable. For a process expression $P \in \mathcal{P}$, $pname[\mathsf{local}\colon lv_1, \ldots, lv_n] := P$ defines $pname$ to be P where the variables $lv_1, \ldots, lv_n$ are local. Syntactically, the use of process definitions is of the form (e.g. in a sequence)

$$\mathsf{Seq}(\ldots, \mathsf{UseDefinition}(pname[lv_{k_1} \leftarrow v_{\ell_1}, \ldots, lv_{k_m} \leftarrow v_{\ell_m}]), \ldots)$$

with the following semantics: let Var denote the set of active variables used in the surrounding context. The definition of $pname$ is invoked based on the current tuples, where each tuple is extended or modified by initializing the local variables $lv_{k_1}, \ldots, lv_{k_m}$ ($k_i \in \{1, \ldots, n\}$) with the values of the variables $v_{\ell_1}, \ldots, v_{\ell_m} \in \mathsf{Var}$. Formally,

$$\begin{aligned}
&[\![\mathsf{UseDefinition}(pname[lv_{k_1} \leftarrow v_{\ell_1}, \ldots, lv_{k_m} \leftarrow v_{\ell_m}])[R]]\!] := \\
&[\![P[\{t \in \mathsf{Tuples}(\mathsf{Var} \cup \{lv_{k_1}, \ldots, lv_{k_m}\}) \mid \text{exists } t' \in R \text{ s.t.} \\
&\qquad \rho[v_{\ell_1} \leftarrow lv_{k_1}, \ldots, v_{\ell_m} \leftarrow lv_{k_m}](\pi[\{lv_{k_1}, \ldots, lv_{k_m}\}](t)) = \pi[v_{l_1}, \ldots, v_{l_m}](t') \\
&\qquad \text{and } \pi[\mathsf{Var} \setminus \{lv_1, \ldots, lv_n\}](t) = \pi[\mathsf{Var} \setminus \{lv_1, \ldots, lv_n\}](t')\}]\!]
\end{aligned}$$

which is a subset of $\mathsf{Tuples}(\mathsf{Var} \cup \{lv_1, \ldots, lv_n\})$. Note that recursive processes call themselves inside their definition; in this case, $\{lv_{k_1}, \ldots, lv_{k_m}\} \subseteq \mathsf{Var}$.

2.5 Data-Oriented RelCCS Operators

While the above operators extend the classical CCS operators that focus on the *control flow* with relational state, additional operators integrate unary relational operators: projection, duplicate elimination, and top-k.

Projection and Duplicate Elimination. In relational algebra, projection is a very useful operator to reduce intermediate results when some variables are no longer needed. For RelCCS, $\mathsf{Projection}(v_1, \ldots, v_n)$ with a specification which variables to keep does the analogue during execution of a process. Distinct removes duplicate tuples, and is usually applied after a projection.

The RelCCS Top-K Operator. The top-k operator is known as a useful extension for many applications. It allows to "take the best k answers" and continue. For instance, when a set of potentially relevant airports are known, only the nearest 10 to the starting place should be considered for continuing the process. Here we adapt the top-k functionality to asynchronous processing of RelCCS workflows: Applied to a set of tuples over variables $v_1, \ldots, v_n$,

$\mathsf{TopK}(k, m, t, \mathsf{mapfct}, \mathsf{datatype}, \mathsf{order}, \mathsf{cont})$ acts as follows:

- wait until either m tuples are present, or t time units have passed,
- then, for each tuple, compute $\mathsf{mapfct}(v_1, \ldots, v_n)$ (expressed as an embedded query) which yields a value of datatype. Order them according to order (which can be either asc or desc) and take the top k, and return them.
- if cont is true, then for every tuple coming in later, check if it is amongst the best k up to now. If yes, return it, otherwise discard it.

2.6 Embedding Algorithmic Webservices

The design of the MARS framework allows to use Web Services that communicate via the atomic metaphors of MARS: actions, tests, queries, and optionally events. Such auxiliary Web Services can for instance provide the functionality of abstract datatypes that embed algorithmic aspects in form of external support in the declarative specification of RelCCS workflows.

A recurring motive when designing query workflows is the computation of (parts of) transitive closures in graphs, as in our travel scenario. For this, a GDT –Graph DataType– Service provides a configurable API to graphs. Edges and paths in an application usually have properties where the properties of the paths are defined inductively via its edges. For the present paper, we take a GDT instance as given that provides the following actions and queries:

- a query $gid \leftarrow$ newGraph() with the side effect of initializing an empty graph,
- an action addEdge(gid, $from$, to, $[p_1 \leftarrow v_1, \ldots, p_n \leftarrow v_n]$), that adds the respective edge with values of v_i for parameters p_i, and computes new paths accordingly,
- an accessor (query) $v \leftarrow$ getNewVertices(gid) that returns all vertices that have been added since the preceding call of getNewVertices(gid).
- an action reportPaths($start$, $dest$, $[v_1 \leftarrow p_1, \ldots, v_m \leftarrow p_m]$) that returns all paths that connect $start$ with $dest$ with their parameters p_i bound to variables v_i.

2.7 Technical Realization

RelCCS has been implemented as a language service within the MARS framework. RelCCS processes are given as XML documents (or as RDF graphs), borrowing the main principles from MARS' ECA-ML markup language [13]. The language markup has the usual form of a tree structure over the CCS composers in the ccs: namespace. Every expression (i.e., the CCS process, its CCS subprocesses, the event specification, test, queries, and the atomic actions) is an XML (sub)tree whose namespace (i.e., the URI associated with the prefix) indicates the language.

The services are implemented in Java, using a common set of basic classes that handle e.g. the variable bindings. For larger numbers of tuples, an SQL database is used as backend [12]. The actual data exchange is done in an XML format for results and variable bindings. Determining an appropriate service and organizing the communication is performed by a *Languages and Services Registry (LSR)* and a *Generic Request Handler (GRH)* [6].

3 Application Scenario: Travel Planning

The RelCCS process for sample travel planning scenario can now be given: If the overall distance is less than 400km, only train connections are searched for. Otherwise, train connections (for less than 800km) and flights are investigated.

For the latter, train connections to potential airports are searched, and the remaining distance is bridged by connecting flights and, if necessary, a final train connection.

Recall that the whole connection graph is not accessible like a database, but must be explored by Web queries. The design of the *process* is thus significantly different from straightforward bottom-up evaluation of transitive closure queries in Datalog. The relevant fragment of the connection graph is built stepwise online during the workflow, using the GDT service as described above. The strategy is based on reachability by breadth first search for shortest paths. Edges (i.e., connections) and their properties (as slotted name-value pairs; i.e., departure and arrival time and price) are obtained from Web queries, and added to the graph. Note that each single connection can be useful in several combinations.

The process uses variables *start* and *dest* (destination), *date* (which are initialized when calling the process), *ap* (relevant near airports), *dist* (distance to airport), i, j (intermediate places), *rd* (remaining distance), *dt* and *at* (departure and arrival time), *pr* (price), and *gid* (graph id). We use the prefixes ccs and gdt to indicate the respective languages. We abstract from the concrete data sources, which are for this example actually wrapped Deep Web sources, e.g. `http://www.bahn.de` for (not only German) Railways, and flights/airline portals:

```
ccs:Seq(ccs:Query(rd ← distance(start, dest)),  # process input: (start, dest)
  ccs:Query(gid ← gdt:newGraph()),
  ccs:Union(
    ccs:Seq(ccs:Test(rd < 800),  # consider to go by train
            ccs:Query((dt, at, pr) ← getTrainConnection(start, dest, date)),
            ccs:Action(gdt:addEdge(gid, start, dest,
                [deptTimeLocal ← dt, arrTimeLocal ← at, price ← pr]))),
    ccs:Seq(ccs:Test(rd ≥ 400),   # consider also to use flights
            ccs:Query(ap ← getAirports()),
            ccs:Query(dist ← distance(start, ap)),
            ccs:TopK(10,100,null,dist,xsd:decimal,asc,false), # consider 10 nearest airports
            ccs:Query((dt, at, pr) ← getTrainConnection(start, ap, date)),
            ccs:Action(gdt:addEdge(gid, start, ap,
                [deptTimeLocal ← dt, arrTimeLocal ← at, price ← pr])),
            ccs:Projection(start, dest, date), ccs:Distinct,
            ccs:UseDefinition(runGraph[]))),
    gdt:reportPaths(start, dest, [pathId ← pid, price ← price])),

  ccs:Definition(runGraph[local:i, rd, dt, at, pr]) :=
    # additional global vars gid, dest, date are known
    ccs:Seq(
      ccs:Query(i ← gdt:getNewVertices(gid)), # consider all newly reached places
      ccs:Query(rd ← distance(i, dest)),
      ccs:Union(
        ccs:Seq(ccs:Test(i = dest)),  # no recursive call in this case → return
        ccs:Seq(ccs:Test(rd < 400 ∧ i ≠ dest )  # reach destination by train
```

```
                ccs:Query((dt, at, pr) ← getTrainConnection(i, dest, date)),
                ccs:Action(gdt:addEdge(gid, i, dest,
                    [deptTimeLocal ← dt, arrTimeLocal ← at, price ← pr])),
                ccs:Projection(start, dest, date), ccs:Distinct,
                ccs:UseDefinition(runGraph[])),
        ccs:Seq(ccs:Test(rd ≥ 200), # try to get even nearer by flight
                ccs:Query((j, dt, at, pr) ← getFlights(i, date)),
                ccs:Action(gdt:addEdge(gid, i, j,
                    [deptTimeLocal ← dt, arrTimeLocal ← at, price ← pr]))
                ccs:Projection(start, dest, date), ccs:Distinct,
                ccs:UseDefinition(runGraph[])))))
# postcondition: reached, either by train or train+flight+, or train+flight++train
```

The workflow proceeds stepwise, set-oriented:

Example 2. *Consider to start the workflow with the single tuple*

$\{start/\text{“Heidelberg”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}\}$.

The query for the remaining distance extends the tuple to

$\{start/\text{“Heidelberg”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}, rd/784\}$.

It then starts two branches, one for a train-only travel (since $rd < 800$*), and one that includes consideration of flight connections (since* $rd > 400$*); the results of both will be returned at the end. We follow the second one:*

It will first state a query for all known airports, which results in a set of (thousands of) tuples, where each is extended in the subsequent step with distance between start and the respective airport, i.e.,

$\{\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}, rd/784, ap/\text{“FRA”}, dist/85\},$
$\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}, rd/784, ap/\text{“STG”}, dist/95\},$
$\vdots$
$\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“...”}, rd/784, ap/\text{“JFK”}, dist/6230\}, \ldots\}$

The next topK *step keeps the ten nearest ones, amongst them FRA and STG (and not JFK). For these, the next step looks up (multiple) train connections to each of them, binding price etc., and for each tuple, the connection is put into the graph, and the tuples are projected back down to* $start, dest, date$*, and duplicates are removed (so only the single tuple* $\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}\}$ *remains). Then, the process definition for* runGraph *is invoked. In its first step, the new vertices, which are the 10 nearest airports, are retrieved from the graph and bound to the variable* i *(intermediate):*

$\{\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}, i/\text{“FRA”}\},$
$\{start/\text{“Heidelb.”}, dest/\text{“St.Malo”}, date/\text{“1.1.2009”}, i/\text{“STG”}\}, \ldots\}$

The following iterative process is then concerned with extending the graph in parallel (i.e., set-oriented for all tuples) breadth-first search until connections to $dest$ *(i.e., when* $i = dest$*) are found.*

Figure 1 illustrates a fragment of the contents of the GDT. The sample also illustrates that (i) some paths are not needed to be inserted as already better ones are known [*] and, on the other hand, "longer" (in the number of steps) paths can be better wrt. the user's criteria (price, duration; [**]).

id	head	tail	label	start	end	dept	arr	price	
p_1	e_1	null	HD→FRA	HD	FRA	7:30	8:10	29.00	
p_2	e_2	null	HD→STG	HD	STG	7:50	8:50	39.00	
p_3	e_{13}	p_1	HD→FRA→CDG	HD	CDG	7:30	11:20	229.00	
p_4	e_{14}	p_1	HD→FRA→LON	HD	LON	7:30	11:50	129.00	
p_5	e_{15}	p_2	HD→STG→CDG	HD	CDG	7:50	13:30	289.00	
p_{71}	e_{23}	p_3	HD→FRA→CDG→StM	HD	StM	7:30	+1:08:30	345.00	
p_{85}	e_{25}	p_3	HD→FRA→CDG→RNS	HD	RNS	7:30	15:20	459.00	
p_{86}	e_{25}	p_5	HD→STG→CDG→RNS	HD	RNS	7:50	17:50	519.00	[*]
p_{93}	e_{29}	p_4	HD→FRA→LON→DNR	HD	DNR	7:30	16:50	159.00	
p_{103}	e_{25}	p_{85}	HD→FRA→CDG→RNS→StM	HD	StM	7:30	18:20	487.00	
p_{123}	e_{39}	p_{93}	HD→FRA→LON→DNR→StM	HD	StM	7:30	17:50	175.00	[**]
:	:	:	:	:	:	:	:	:	

Fig. 1. Sample contents of the GDT data structure

The termination condition, i.e., that when all "open" paths are more expensive than the best k known paths to the destination is enforced by the insertion policy of the graph. This guarantees that the k preferable paths will be reported, and that the workflow terminates. Recall that such completeness is not *guaranteed by the heuristic methods applied by current travel agencies to prune the search space, since this may result in the fact that "unexpected" connections –like reaching St.Malo via Stansted or Jersey– are excluded.*

4 Related Work

Two already traditional areas that are related to our work are (i) query plans for relational algebra expressions that work on the operator level, and also on the choice of actual algorithms for, e.g., joins, and (ii) conjunctive queries over homogeneous or heterogeneous sources, including HTML and XML Web sources, for querying issues according to the yet classical wrapper-mediator architecture that provide integrated *views* on data, but without considering process-oriented aspects of data-oriented *workflows*. Since in these areas, the control flow does not play a central role, we do not further discuss them.

Dataflow and Data Exchange: Comparison to Tuple Spaces. A frequently asked question is the relationship between the dataflow model in MARS and RelCCS, and *Tuple Spaces* [7] (in the following abbreviated as "TS") and its variants. TS are a middleware approach for cooperation and coordination between distributed processors, in the TS context usually called agents. A TS is an unstructured collection of tuples without fixed schema that allows for associative access: insert,

read, read with delete; updates are accomplished by removing and inserting. IBM TSpaces [19] support four further types of Queries: MatchQuery, IndexQuery, AndQuery, and OrQuery, that all result in sets of tuples.

Similarities between RelCCS and TS are thus in the support for data exchange between autonomous, distributed processors. Also, in both approaches, the data is decoupled from the programs. In TS, data can explicitly exist without being assigned to a certain agent. Communication is anonymous from the point of view of the processors – they get and put tuples from/to the TS.

TS are in many aspects similar to relational databases, but they are used differently. We shortly analyze the main aspects wrt. MARS and RelCCS:

- TS: Use as communication bus, not permanent storage. This characteristic is shared with MARS and RelCCS.
- TS: Unstructured set of tuples. MARS: sets of tuples that belong together.
- TS: Associative access operations. Not needed by MARS and RelCCS.
- TS: Generally, no predefined schema. In MARS and RelCCS, for each state, all tuples have the same schema, which changes during the processing.

So, MARS and RelCCS do not need some of the features of TS. On the other hand, a core requirement of MARS is not covered by TS: Tuples in MARS are grouped into sets of tuples (the above relations over the active variable names) and usually assigned to a (single) current processor, and exchanged between processors in a directed and controlled way. Moreover, the RelCCS operators require to apply *relational operations* on the *sets of tuples*. Functionally, the definition of sets of tuples belonging together could be emulated in TS by an additional column c_0 of the tuples. Nevertheless, TS do not efficiently support *operations* on such sets of tuples, like e.g. joining the result relation R of a query with the previous tuples, joins of branches of concurrent subprocesses, projection, duplicate elimimation, and top-k. For MARS and RelCCS, using a relational database as "communication bus" is preferable since the operations can easily be mapped to relational operations on database tables [12]. Note that there is also a realization without any middleware (except plain internet communication) using data exchange by XML (i.e., sets of tuples serialized as XML) and operations performed on an internal (Java) data structure.

Workflow and Dataflow. Several approaches have been presented that combine dataflow with control flow: The focus of Petri Net-based approaches is to express workflows completely in a uniform graphical formalism, with a concise formal semantics to be able to apply formal analysis and verification techniques. Extensions of Petri Nets with nested relational structures are investigated in [15] (*NR/T*-nets) and [9] (*Workflow Nets/Dataflow Nets*). The language YAWL [1], which has been designed based on an exhaustive analysis of workflow patterns [2] and has its roots in Petri Nets, also treats dataflow as a first class citizen.

Petri nets are, like RelCCS, process-oriented. While RelCCS is based on a set of operators, in Petri Nets, the control flow patterns such as concurrent execution and recursion have also to be encoded within the Petri Net formalism. Additionally, abstract data types, such as the GDT, must also be encoded.

Many current approaches to workflow languages, such as BPEL [5] also provide an XML markup. In contrast to RelCCS in the MARS framework, where the XML markup carries important language information for enabling the processing of embedded language fragments, these languages use XML just as a serialization format. Dataflow in BPEL is described by BPEL variables, which can, using appropriate database products like e.g. IBM WebSphere, reference database tables, and thus be made set-valued. Also datatypes like GDT can be embedded into BPEL processes. In [18], optimization strategies of such approaches are discussed.

In *Transaction Logic* $\mathcal{TR}$ [4] and *Concurrent Transaction Logic* $\mathcal{CTR}$ [16] the description of a workflow consists of *rules* that make use of temporal connectives instead of just the Datalog conjunction. The semantics of $\mathcal{TR}$ is inherently set-valued. Such rules can be formulated over embedded literals/atoms (called *elementary transitions*) that are *not* part of Transaction Logic, but are contributed externally. This is similar to the embedding of the use of the GDT data type in RelCCS.

Furthermore, systems for data-oriented workflows in general can be applied for query answering tasks. Such systems usually have a set-oriented dataflow. The Lixto Suite [8] is an integrated system for implementing data-oriented workflows with a focus on data acquisition and integration. Its process model is less explicit, and the workflows are solely built upon Lixto's own modules. *Kepler* [3] is an extensible system for design and execution of scientific workflows whose goal is to capture, formalize, and reuse workflows. It supports a concept of individual, reusable workflow steps.

Although these approaches *can* encode the same behavior, the advantage of RelCCS is that it provides both the primitives for control structures and data flow on the same level of the language. A further feature of the language is provided by its embedding in the MARS meta model: RelCCS fragments can be used e.g. as action part in MARS' ECA rules, and fragments in other languages for specifying complex events, queries and atomic actions can be embedded in RelCCS processes without having to revert to Web Services as intermediate wrappers.

5 Conclusion

In this paper, we presented the RelCCS approach for specifying and executing data-oriented workflows and discussed its use for solving tedious, repetitive, although complex tasks related to answering queries based on Web data. For such tasks, it is often simpler to design the process *how* to solve the problem, than stating a single query. We also illustrated how complementing module-like data structures can be embedded to support the algorithmic issues of such processes, and gave an impression what processes in this framework look like. Apart from the use for query answering as described above, RelCCS can also be applied for specifying data-oriented workflows in general.

RelCCS is implemented in a prototype which can be found with sample processes and further documentation at http://www.semwebtech.org/mars/frontend/ → run CCS Process.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Inf. Syst. 30(4), 245–275 (2005)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
3. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B., Mock, S.: Kepler: An extensible system for design and execution of scientific workflows. In: SSDBM 2004, pp. 423–424 (2004)
4. Bonner, A.J., Kifer, M.: An overview of Transaction Logic. Theoretical Computer Science 133(2), 205–265 (1994)
5. Business Process Execution Language (BPEL), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
6. Fritzen, O., May, W., Schenk, F.: Markup and Component Interoperability for Active Rules. In: Calvanese, D., Lausen, G. (eds.) RR 2008. LNCS, vol. 5341, pp. 197–204. Springer, Heidelberg (2008)
7. Gelernter, D.: Generative communication in Linda. ACM TOPLAS 7(1), 80–112 (1985)
8. Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: The Lixto data extraction project - back and forth between theory and practice. In: ACM PODS, pp. 1–12 (2004)
9. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., den Bussche, J.V.: DFL: A Dataflow Language Based On Petri Nets and Nested Relational Calculus. Inf. Syst. 33(3), 261–284 (2008)
10. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
11. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM 42(4), 741–843 (1995)
12. May, W.: A Database-Based Service for Handling Logical Variable Bindings. Databases as a Service, Technical Report, Univ. Münster, Germany (2009)
13. May, W., Alferes, J.J., Amador, R.: Active rules in the Semantic Web: Dealing with language heterogeneity. In: Adi, A., Stoutenburg, S., Tabet, S. (eds.) RuleML 2005. LNCS, vol. 3791, pp. 30–44. Springer, Heidelberg (2005)
14. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science, 267–310 (1983)
15. Oberweis, A., Sander, P.: Information system behavior specification by high-level Petri Nets. ACM TOIS 14(4), 380–420 (1996)
16. Roman, D., Kifer, M.: Reasoning about the Behavior of Semantic Web Services with Concurrent Transaction Logic. In: VLDB, pp. 627–638 (2007)
17. SPARQL Query Language for RDF (2006), `http://www.w3.org/TR/rdf-sparql-query/`
18. Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., Kraft, T.: An Approach to Optimize Data Processing in Business Processes. In: VLDB, pp. 615–626 (2007)
19. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T Spaces. IBM Systems Journal 37(3), 454–474 (1998)