ServLog: A Unifying Logical Framework for Service Modeling and Contracting

Dumitru Roman [a] and Michael Kifer [b]
[a] *SINTEF, Forskningsveien 1,0314 Oslo, Norway*
E-mail: dumitru.roman@sintef.no
[b] *Stony Brook University, Stony Brook, NY 11794-2424, U.S.A.*
E-mail: kifer@cs.stonybrook.edu

Abstract.
Implementing semantics-aware services, which includes semantic Web services, requires novel techniques for modeling and analysis. The problems include automated support for service discovery, selection, negotiation, and composition. In addition, support for automated service contracting and contract execution is crucial for any large scale service environment where multiple clients and service providers interact. Many problems in this area involve reasoning, and a number of logic-based methods to handle these problems have emerged in the field of Semantic Web Services. In this paper, we lay down theoretical foundations for service modeling, contracting, and reasoning, which we call *ServLog*, by developing novel techniques for modeling and reasoning about service contracts with the help of Concurrent Transaction Logic. With this framework, we significantly extend the modeling power of the previous work by allowing expressive data constraints and iterative processes in the specification of services. This approach not only captures typical procedural constructs found in established business process languages, but also greatly extends their functionality, enables declarative specification and reasoning about services, and opens a way for automatic generation of executable business processes from service contracts.

Keywords: service modeling and contracting, service process, constraints, Semantic Web service, automated reasoning, declarative modeling of processes.

1. Introduction

The move towards service-aware systems, starting with emergence of service-oriented architectures and service computing [35], and newer approaches such as artifact-centric [28] and data-aware systems [12], microservices [34], and not least the emerging field of service science [43], calls for the development of novel techniques to support various service-related tasks such as service modeling and discovery, service process specification, automated contracting for services, service enactment and monitoring. These issues have been partially addressed by a number of pioneering projects in the area of Semantic Web Services, such as WSMF [19], OWL-S [31],[1] WSMO [37],[2] SWSL,[3] DIP [46],[4] and SUPER [26,27],[5] and more recently [40]. Nevertheless, many core issues remained largely

[1]http://www.daml.org/services/owl-s/
[2]http://www.wsmo.org/
[3]http://www.w3.org/Submission/SWSF-SWSL/
[4]http://dip.semanticweb.org/
[5]http://ip-super.org/

unsolved. The present paper builds on the previous efforts while primarily addressing the behavioral aspects of services, including service contracting and service contract execution. It complements approaches such as OWL-S and WSMO, which primary focused on semantic annotations for Web services, brings new insights, and points to new directions for research in Semantic Web Services.

In a service-oriented environment, interaction is expected among large numbers of clients and service providers, so making contracts through human interaction is not feasible. To enable automatic establishment of contracts, a formal contract description language is needed, and a reasoning mechanism must verify that the contract terms are fulfilled, as well as support the execution of the contract. A contract specification has to describe the functionality of a service, values to be exchanged, procedures, and guarantees. A service contracting and contract execution reasoning mechanism has to decide whether such a specification can actually satisfy the constraints of the parties involved in the contract, and if so, support the execution of the contract in a way that the constraints are satisfied. The present paper develops just such a unifying logical framework, called *ServLog*.

ServLog is based on *Concurrent Transaction Logic* (CTR) [11] and continues the line of research that employs CTR as a unifying formalism for modeling, discovering, choreographing, contracting, and enactment of Web services [16,42,17,38,39]. Transaction Logic has also been successfully used in a number of other domains ranging from security verification policies to reasoning about actions and other service-related issues [6,36,23,7,21]. The present work builds on the results reported in [38,39], but greatly extends that previous work through generalization and addition of new modeling and reasoning techniques. All this is achieved while at the same time significantly simplifying the technical machinery. The contributions with respect to our previous work are detailed in Section 6. With *ServLog* we lay down the theoretical foundations for service contracting. Specifically, we extend the expressive power of the constraint language used for specifying contracts, allow iterative processes, and allow to pass arguments to processes. We also extend our reasoning techniques to deal with the new expressive variety of modeling primitives, making it possible to address an array of issues in service contracts, ranging from complex process descriptions to temporal and data constraints. The inference procedure for CTR developed here also contributes to the body of results about CTR itself — it covers CTR conjunctive formulas that enable execution of *constrained transactions*, which previous CTR proof theory was not able to handle. We also develop a logical language for specifying and enacting processes of great complexity.

While this paper aims to be self-contained and we went to great length to provide sufficient details on CTR, it is clear that we must assume certain background from the reader. Specifically, the paper requires proficiency in basic predicate calculus and Logic Programming.

The remainder of this paper is organized as follows. Section 2 informally describes the basic techniques from service contract specification used in *ServLog* and introduces the problem of service contracting and service contract execution. Section 3 gives a short introduction to CTR to keep the paper self-contained. Section 4 formally defines modeling constructs. Section 5 describes the reasoning procedure of *ServLog*—the key component of service contracting and contract execution in our framework. Section 6 discusses related works and contrasts them with *ServLog*. Section 7 concludes the paper.

2. Service Modeling, Contracting, and Contract Execution

There is a number of modeling languages for capturing interactions between services and clients (or among internal tasks within the same service), some focusing on specific features and targeting different audiences (e.g. business analysts, Web service developers, etc.). For example, the Business Process Model and Notation (BPMN)[6] is a standard for business process modeling and provides a graphical notation for specifying business processes (BPMN distinguishes between public and private processes, choreographies, and collaborations; it can provide different views of internal and external interactions). Another approach to modeling service processes is the WSMO model of choreography,[7] which is limited to server-side interactions. In contrast, the model of the W3C Choreography Group

[6]http://www.omg.org/spec/BPMN/2.0/
[7]http://www.wsmo.org/TR/d14/

includes both service-side interactions and client-side interactions.[8] At a higher level of abstraction, however, all interactions can be represented through *control* and *data* dependencies between tasks. *ServLog* captures this level of abstraction in a logic and enables powerful forms of automated reasoning about it. Figure 1 depicts the main aspects of service behavior addressed by *ServLog*.

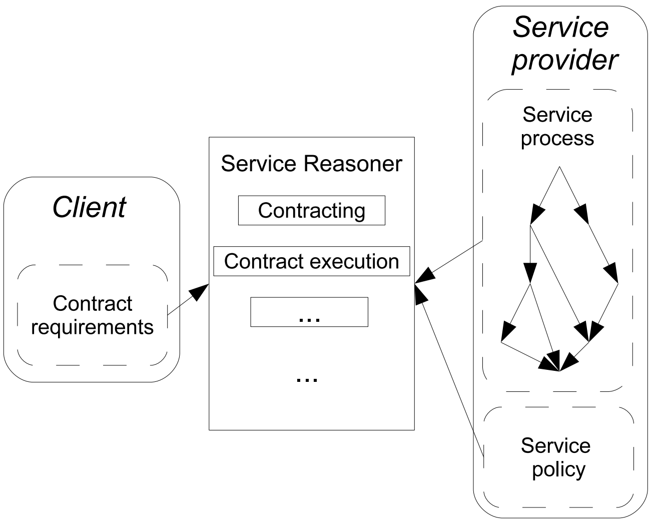

Fig. 1. Elements of the reasoning architecture for services in *ServLog*.

The *service process* is described through its *control and data flows*—a specification that tells how to interact with the service and how data flows among tasks. A *service process* may also expose inner workings of the service (interactions that are not between the service and the client but between internal tasks of the service or third parties) for common situations in which the service allows the client to impose constraints on how the service process is to be executed.

The *service policy* component in Figure 1 is a set of constraints on a service process and on its input. The *contract requirements* included on the client side of the figure represent the contractual requirements of the user that go beyond the basic functions of a service. (Examples of basic services are selling books and helping with travel arrangements, while an example of a service requirement is the request that the amount should be charged only after shipping.) In *ServLog*, service processes are described via *control* and *data flow graphs*, while service policy and client contract requirements are described via *constraints*.

We will now discuss these modeling aspects in more detail using a typical order placement scenario. This scenario describes the flow of interaction between a client and a service, where the interaction starts with the user placing an order, after which the service initiates a concurrent execution of processing the order items, handling shippers for the items, and receipt of a payment. The order processing workflow ends once the above tasks all finish. Processing the order items, handling shippers, and payment receipt are specified in further details, leaving the possibility for the client and the service to make some choices during the interaction (for example, providing a full payment for the order or paying per item). At the same time, the scenario shows how data (e.g. $Order\#$) flows through the workflow tasks as the interactions happen. In addition, the scenario includes a set of non-trivial constraints imposed by the client and the service, which affect the execution. For example, the service has the policy (expressed as a constraint) of booking a shipper only if there are at least seven items to be sent with the shipper. The description of the scenario ends with the definition of the problems addressed in this paper, namely service contracting and service contract execution.

[8]http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/

Service process. Figure 2 shows the *service process* described earlier as a hierarchical *control and data flow graph*, called a *process graph*. The control flow aspect of process graphs is typically used to specify local execution dependencies among the interactions of the service; it is a good way to visualize the overall flow of control. Data flow complements the control flow by specifying the data dependencies among the interactions.

With Figure 2 we are not attempting to suggest yet another notation for service processes; the purpose is to introduce and explain our running example in a compact and focused way. Representing the same information, say, in BPMN would have been much bulkier and would require inventing additional notation to compensate for features (such as data flow) that BPMN lacks.

Control flow. The nodes in a service process graph represent *interaction tasks*, which can be thought of as functions that take inputs and/or produce outputs. Some tasks are meant to be executed by the service and some by the client. The distinction between service and client tasks is part of the service process description. In general there can be several actors involved, some acting as clients in one context and services in another.

In Figure 2, tasks are represented as labeled rectangles. The label of a rectangle is the name of the task and the graph inside the rectangle is the *definition* of that task. Such a task is called *composite* because it has nontrivial internal structure. Tasks that do not have associated graphs are *primitive*. A service process graph can thus be viewed as containing a hierarchy of tasks. The graph shown at the top is the *root* of the hierarchy. In our example, the tasks of the top-level graph include **process_order_items**, **handle_shippers**, and **handle_payment**. These tasks are composite and their rectangles are shown separately. The task **place_order** is an example of a *primitive* task. Such tasks have grey background in the figure. Three such tasks, **place_order**, **full_payment**, and **pay_one_item**, are client tasks. The rest are service tasks.

The top-level graph and each composite task has an initial and a final interaction task, the successor task(s) for each interaction task, and a sign that tells whether all these successors must be executed concurrently (represented by **AND**-split nodes), or whether only one of the alternative branches needs to be executed non-deterministically (represented by **OR**-nodes).[9] For instance, in the top-level graph, all successors of the initial interaction **place_order** must be executed whereas in the definition of **pay** either **full_payment** or **pay_per_item** is to be executed.

Composite tasks may be marked with the suffix "*", which means that these tasks may execute multiple times. We call these tasks *iterative* and differentiate them from *non-iterative* tasks. Iteration is indicated through recursive occurrences of the same tasks—by placing tasks inside their own definitions. Figure 2 shows two iterative tasks: **process_order_items** and **pay_per_item**. For example, **process_order_items** is an iterative task where a sequence of sub-tasks, **select_item** followed by **process_item** can be executed multiple times (for example for each item in the purchase order). Iteration is indicated by an occurrence of a **process_order_items** box to the right of the **process_item** box. Note that recursive occurrences of tasks may be followed by other tasks, which gives us a general mechanism for capturing different kinds of iterations, including loops and nesting.

It should now be clear how the control flow aspect of the service process graph in Figure 2 represents the virtual manufacturer scenario described earlier: first the order is placed (**place_order**), then the items in the purchase order are processed (**process_order_items**), delivery is arranged (**handle_shippers**), and payment is settled (**handle_payment**). These three tasks are executed in parallel. Once all of them complete, handling of the order is finished (**end_order**). The other parts of the figure show how each of the above tasks is executed. The important thing to observe here is that some tasks are complex and some primitive; some are to be executed in parallel (the **AND**-nodes) and some in sequence; some tasks have non-deterministic choice (the **OR**-nodes) and some are iterative.

Data flow. Interaction with a service typically involves passing data and the flow of that data is normally captured using data dependencies between tasks. Such dependencies complement the control flow and complete the description of the service process graph.

Since tasks can be conceptualized as functions that take input and produce output, arguments are attached to tasks to capture both input and output. In Figure 2, each task-label has one or more arguments. For example, **handle_payment** has the arguments *(Order#,Price)*, meaning that, to execute **handle_payment**, *Order#* and *Price* must

[9]This non-determinism has an XOR flavor.

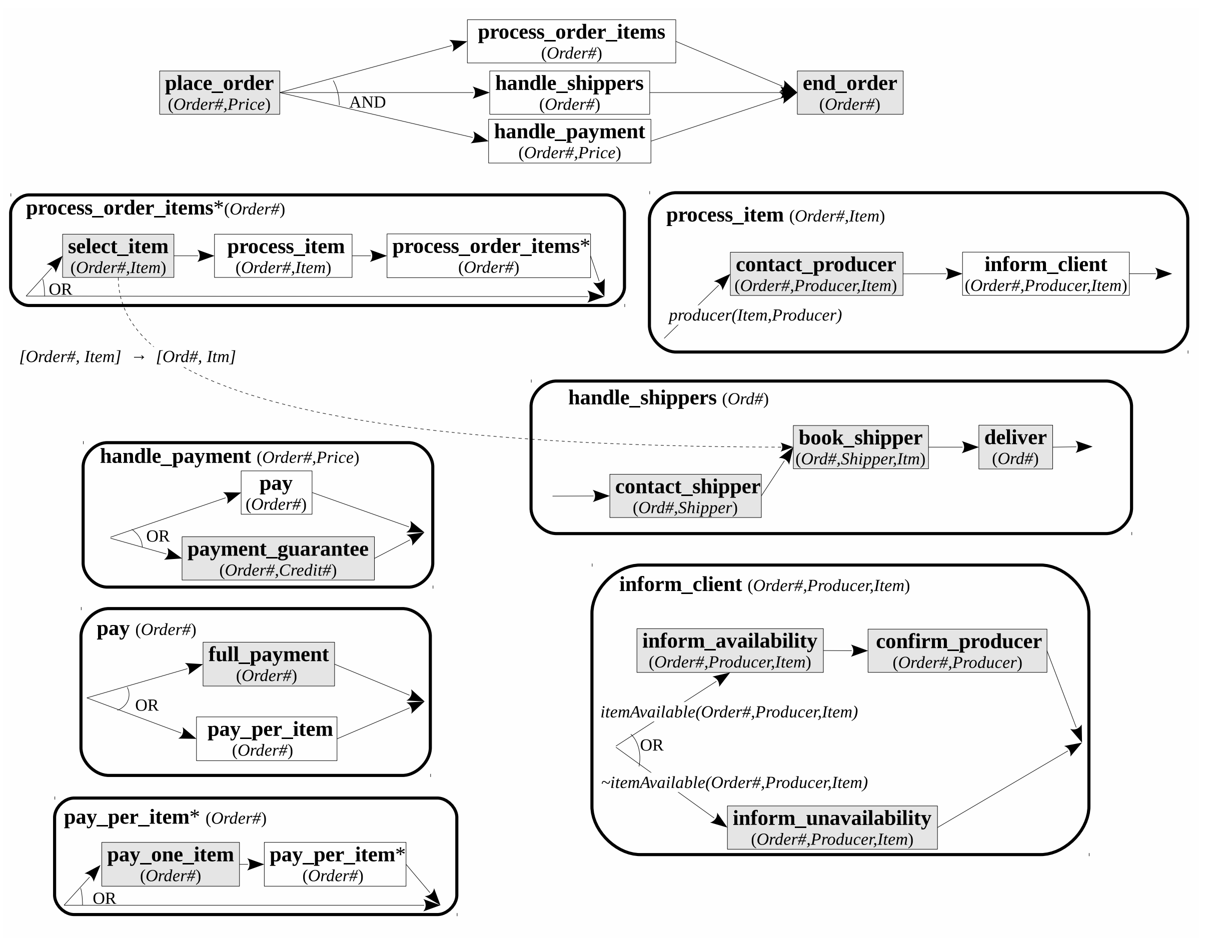
process_order_items (Order#)
place_order (Order#,Price)
AND
handle_shippers (Order#)
handle_payment (Order#,Price)
end_order (Order#)
process_order_items*(Order#)
select_item (Order#,Item)
process_item (Order#,Item)
process_order_items* (Order#)
OR
process_item (Order#,Item)
contact_producer (Order#,Producer,Item)
inform_client (Order#,Producer,Item)
producer(Item,Producer)
[Order#, Item] → [Ord#, Itm]
handle_shippers (Ord#)
book_shipper (Ord#,Shipper,Itm)
deliver (Ord#)
contact_shipper (Ord#,Shipper)
handle_payment (Order#,Price)
pay (Order#)
OR
payment_guarantee (Order#,Credit#)
inform_client (Order#,Producer,Item)
inform_availability (Order#,Producer,Item)
confirm_producer (Order#,Producer)
itemAvailable(Order#,Producer,Item)
OR
~itemAvailable(Order#,Producer,Item)
inform_unavailability (Order#,Producer,Item)
pay (Order#)
full_payment (Order#)
OR
pay_per_item (Order#)
pay_per_item* (Order#)
pay_one_item (Order#)
pay_per_item* (Order#)
OR

Fig. 2. Service process example: A hierarchical control and data flow graph.

be provided.[10] In our scenario, *Order#* and *Price* will be provided by the service's task **place_order**, which will generate an order number and compute the price based on the items selected by the client (and the pricing data stored in the database). These data items will then be passed to other tasks, such as **handle_payment**.

Data-passing between tasks is captured via the shared argument names and through a shared data space (e.g., a database) of the workflow process.[11] Data-passing through shared arguments is possible between a task and its direct successors, or within the definition of the same composite task. The scope of arguments is relevant in this case: argument names of a task are *logical variables*. When they are shared with the task's direct successors, they refer to the same data items (i.e. data is passed from tasks to their direct successors using shared arguments). This aspect should be familiar from basic logic and Logic Programming. For example, data identified by *Order#* in **process_order_items***(Order#)* is the same as the data identified by *Order#* in **place_order***(Order#,Price)*, i.e. **place_order** passes *Order#* to **process_order_items**. In case of a composite task, the names of its arguments are global to that task's definition, meaning that if subtasks in its definition use the same argument names as the composite task then they refer to the same data items. In this way, the composite task passes data to its subtasks. For example, the composite task **process_order_items** passes *Order#* to its **process_item** subtask.

[10]In general, arguments can be **in** or **out** (and even **in-out**). In logic programming, this is typically specified via **modes**. In our description, the mode should be clear from the context. We avoid specifying the modes explicitly in order to avoid unnecessary distraction.

[11]*ServLog* is independent of the choice of such a shared space.

Note also that sharing via shared variables is bi-directional, as pure logic has no notion of explicit input and output. However, for practical reasons, some logic programming systems accept mode specification and do mode-inference, which allows the user to identify the producers of data.

Data-passing through shared data space is used when passing data is not possible through shared arguments due to the difference in scope of the arguments. This often occurs when data needs to be shared between tasks that have no control dependencies in the control flow part of the service process graph. For example, if **select_item***(Order#,Item)* needs to pass *Order#* and *Item* to **book_shipper***(Ord#,Shipper,Itm)*, data-passing through shared argument names is not an option, since these subtasks appear inside composite tasks that are not related via control dependencies. To capture such data-passing, a shared database can be used as follows: **select_item** can store *Order#* and *Item*, and **book_shipper** can read them later. This is depicted by the dashed arc going from **select_item** to **book_shipper**. The label on the arc represents the data items that are being passed. In our case, *Order#* is being passed as *Ord#* and *Item* as *Itm*.

Data items can be *consumable* or *non-consumable.* In case of data-passing through shared argument names, data is non-consumable: data-producing tasks *share* data via all of their out-ports with the receiving tasks. The latter can further share this data with their descendants, and so on. In our example, **place_order** produces *Order#*-items, and that item is then shared with **process_order_items**, **handle_shippers**, and **handle_payment**. Note the emphasis on *sharing*: all tasks involved work on the *same* copy of the data—the phenomenon that is familiar from basic predicate logic and logic programming. In case of data-passing through a shared database, data can be consumable or *non*-consumable. It is consumable when each query to the databases is followed by deletion of the queried data item. It is non-consumable data if such deletion is not implied. In our example, all data items are consumable.

One other aspect of the service process graphs concerns *transition conditions* on arcs. Examples of such conditions are *producer(Item,Producer)* in the definition of **process_item** and *itemsAvailable(Order#,Producer,Item)* or *itemsUnavailable(Order#,Producer,Item)* in the definition of **inform_client**. Another example of transition condition is a test of the form $Quantity > 3$. A transition condition signifies that in order for the next interaction with the service to take place, the condition must be satisfied. Transition conditions are Boolean tests attached to the arcs in the service process graphs. These tests may also be queries to the underlying database. Only the arcs whose transition conditions evaluate to true can be followed at run time. For uniformity, *ServLog* treats transition conditions formally as a separate type of task.

The final remark concerns the nature of *primitive tasks*. A primitive task is a black box that performs operations in a way that is completely hidden from *ServLog*'s reasoning system. It does not mean that the work performed by the task is trivial. For example, **place_order** may perform database updates to record the order number, price and customer's information, send an email notification to the customer, perform a credit check, and do many other things. The point is that all these operations might not be of much interest to the service's logic designer and she might decide to abstract them away. If, however, the details of some formerly primitive task might become important for the reasoning mechanism, the tasks may be elaborated upon and become composite. We will illustrate this idea in a very concrete way in Figure 5 of Section 4.

Service Policies and Client Contract Requirements. Apart from the local dependencies represented directly in control flow graphs, *global constraints* often arise as part of *policy* specification. Another case where global constraints arise is when a client has specific requirements to the interaction with the service. These requirements usually have little to do with the functionality of a service (e.g., handling orders); instead they represent guarantees that the client wants before entering into a contract with the service. We call such constrains *client contract requirements*. In Figure 3 we give an example of global constraints that represent service policies and client contract requirements for our running example.

Constraints can be imposed on separate tasks (e.g., a task must or must *not* execute, *may* execute a certain number of times) or it can involve several tasks (a task must execute in a certain relationship to another task, e.g., before, after, between). Furthermore, constraints can be combined using Boolean connectives (e.g., a task must execute but after its execution some other task must *not* execute or must execute some number of times).

Other constraints may involve data only. Examples of such constraints include service pre- and post-conditions. For instance, the requirement that "a confirmation number must be available after the execution of the **book_shipper** service" is a post-condition for that service, where the confirmation number is a data item in the constraint. Since data

Service policy
1. A shipper is booked only if the user accepts at least 7 items.
2. If pay-per-item is chosen by the user, then the payment must happen immediately before each item delivery.
3. Payment guarantee must be given before a shipper is booked.

Client contract requirements
4. All items in the same order must be shipped at the same time.
5. If full payment is chosen by the client, then it must happen only after all purchased items are delivered.
6. Before the client purchases items, the service must book a shipper.

Fig. 3. Global behavioral constraints on iterative processes.

in such constraints arise as a result of interactions, this kind of constraint can be seen as a special case of constraints on interactions. Other types of constraints involve Quality of Services (QoS) and Service Level Agreements (SLAs). For instance, "availability provided by the **book_shipper** service is always greater than the requested availability" is a QoS requirement. *ServLog* can also model QoS constraints but the treatment of such specialized constraints is outside the scope of this paper.

Service Contracting and Service Contract Execution. With a modeling mechanism in place, we define service contracting and service contract execution in *ServLog* as follows:

– *Service contracting*: Given a service process (i.e., control and data flow) and a set of service policies and client contract requirements (i.e., constraints), decide whether an execution of the service process that satisfies both the service policies and the client contract requirements exists. Note that it does not matter in the end if this is actually executed but the important aspect is that there is at least one and execution of the contract can proceed.
– *Service contract execution*: Execute tasks in the process in a way where client and service take turns as prescribed by the control and data flows and the constraints. When a step is proposed, the logic's proof system verifies if acceptance of that step still leaves the possibility of a successful execution of the entire service process that satisfies all the constraints. If so, the step is accepted and executed; it is rejected otherwise. A list of possible allowed steps can also be suggested by the system at each turn.

To solve the above two problems, Section 4 formally defines service processes, service policies, and client contract requirements using Concurrent Transaction Logic (CTR). Section 5 then extends the original proof theory of CTR to make it possible to address the above reasoning tasks.

3. Overview of CTR

Concurrent Transaction Logic (CTR) [11] is an extension of classical predicate logic, which allows programming and reasoning about state-changing processes. Here we summarize the relevant parts of CTR's syntax and give an informal account of its semantics. For details we refer the reader to [11].

Basic syntax. The atomic formulas of CTR are identical to those of classical logic, i.e., they are expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and the t_i's are terms constructed of constants, variables, and function symbols. Complex formulas are built with the help of connectives and quantifiers. Apart from the classical $\vee$, $\wedge$, $\neg$, $\forall$, and $\exists$, CTR has two additional infix connectives, $\otimes$ (*serial conjunction*) and $|$ (*concurrent conjunction*), and a modal operator $\odot$ (*isolated execution*). For instance, $\odot(p(X) \otimes q(X)) \mid (\forall Y(r(Y) \vee s(X,Y)))$ is a well-formed formula in CTR. The following is an example well-formed formula that represents the top-level composite task of Figure 2: $place\_order(Order\#, Price) \otimes (process\_order\_items(Order\#) \mid handle\_shippers(Order\#) \mid handle\_payment(Order\#, Price)) \otimes end\_order(Order\#)$.

Informal semantics. Underlying the logic and its semantics is a set of database *states* and a collection of *paths*. For this paper, the reader can think of states as just relational databases, but the logic is more general and can deal with a wide variety of states. Formally, in this paper, a *state* is a pair consisting of a state identifier and a relational database.

A *path* is a finite sequence of state identifiers (constants used to refer to the actual states). For instance, if $\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n$ are state identifiers, then $\langle \mathbf{s}_1 \rangle$, $\langle \mathbf{s}_1, \mathbf{s}_2 \rangle$, and $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ are paths of length 1, 2, and n, respectively.

As in classical logic, CTR formulas take truth values. However, *unlike* classical logic, the truth of CTR formulas is determined over paths, *not* at states. If a formula, ϕ, is true over a path $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$, it means that ϕ can *execute* starting at state $\mathbf{s}_1$. During the execution, the current state will change to $\mathbf{s}_2$, $\mathbf{s}_3$, ..., etc., and the execution terminates at state $\mathbf{s}_n$. In such a case we will also call the path $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$ an *execution* of ϕ.

Although we are interested in execution of CTR formulas over paths, if a formula involves the concurrency operator, the subformulas may be executed in an *interleaved* fashion, like database transactions. For instance, if $\phi = (p \otimes q \otimes r) \mid (u \otimes v)$, the concurrency operator means that legal executions of ϕ consist of an execution of some part of $p \otimes q \otimes r$, e.g., p, then of some execution of $u \otimes v$, e.g., u then again of some part of $p \otimes q \otimes r$, such as q or even $q \otimes r$, then the remaining part of $u \otimes v$, i.e., v, etc. The concurrency operator does not preclude the two parts of ϕ from executing one after another (in any order), but this type of non-interleaved execution is less interesting. In the first, interleaved execution, the two parts of ϕ execute not on paths but on *multi-paths*, i.e., on sequences of paths. Execution of one part of ϕ may be broken by executions of another part, so the intervening gaps in the execution of $p \otimes q \otimes r$ are filled by executions of $u \otimes v$ (and vice versa).

A *multi-path* (or an *m-path*) is a sequence $(\pi_1, ..., \pi_k)$ of paths. If $\mu = (\pi_1, ..., \pi_k)$ and $\mu' = (\pi'_1, ..., \pi'_n)$ are two m-paths, their concatenation, $\mu \bullet \mu'$, is the m-path $(\pi_1, ..., \pi_k, \pi'_1, ..., \pi'_n)$ and their *interleaving*, $\mu \| \mu'$, is an m-path of the form $(\kappa_1, ..., \kappa_{k+n})$ such that it is a topological sort of the two sequences μ and μ'. For example, one interleaving of $(\langle \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \rangle, \langle \mathbf{s}_6, \mathbf{s}_7 \rangle)$ and of $(\langle \mathbf{s}_4, \mathbf{s}_5 \rangle, \langle \mathbf{s}_8, \mathbf{s}_9 \rangle)$ is $(\langle \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \rangle, \langle \mathbf{s}_4, \mathbf{s}_5 \rangle, \langle \mathbf{s}_6, \mathbf{s}_7 \rangle, \langle \mathbf{s}_8, \mathbf{s}_9 \rangle)$. Another interleaving is $(\langle \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \rangle, \langle \mathbf{s}_4, \mathbf{s}_5 \rangle, \langle \mathbf{s}_8, \mathbf{s}_9 \rangle, \langle \mathbf{s}_6, \mathbf{s}_7 \rangle)$. Yet another is $(\langle \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \rangle, \langle \mathbf{s}_6, \mathbf{s}_7 \rangle), \langle \mathbf{s}_4, \mathbf{s}_5 \rangle, \langle \mathbf{s}_8, \mathbf{s}_9 \rangle)$, a degenerate interleaving.

Finally, a path $\pi = \langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ is a *merge* of an m-path $(\pi_1, ..., \pi_n)$ if there are integers $1 = i_0 \le i_1 \le i_2 \le ... \le i_{n-1} \le i_n = m$ such that $\pi_1 = \langle \mathbf{s}_{i_0}, ..., \mathbf{s}_{i_1} \rangle$, $\pi_2 = \langle \mathbf{s}_{i_1}, ..., \mathbf{s}_{i_2} \rangle$, ..., $\pi_{n-1} = \langle \mathbf{s}_{i_{n-2}}, ..., \mathbf{s}_{i_{n-1}} \rangle$, $\pi_n = \langle \mathbf{s}_{i_{n-1}}, ..., \mathbf{s}_{i_n} \rangle$. Note that for the merge to be possible, the end-state of each path π_l in the m-path must be the start-state of the subsequent path π_{l+1} for each $1 \le l < n$. For instance, $\langle \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4, \mathbf{s}_5, \mathbf{s}_6 \rangle$ is a merge of the m-path $(\langle \mathbf{s}_1, \mathbf{s}_2 \rangle, \langle \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4 \rangle, \langle \mathbf{s}_4 \rangle, \langle \mathbf{s}_4, \mathbf{s}_5, \mathbf{s}_6 \rangle)$.

A *multi-path structure* is a mapping that, for each multi-path μ, tells which ground atomic formulas are true on μ. Informally, this can be understood as telling which ground atomic transactions can *execute* along μ. Note that CTR formulas hold truth values not over states, but over multi-paths.

First, we connect truth over path of length 1 to database states.

– If $\mathbf{s}$ is a state identifier and p is a fact that is true in the database associated with $\mathbf{s}$ then p is true over the path $\langle \mathbf{s} \rangle$ (and the m-path $(\langle \mathbf{s} \rangle)$).

CTR connectives are used to construct composite formulas out of the atomic ones, and the statements below define which composite formulas are true on which multi-paths.

– $\phi \otimes \psi$: *execute ϕ then execute ψ*. Model-theoretically: $\phi \otimes \psi$ is true over an m-path μ in a multi-path structure if ϕ is true over a prefix m-path of μ, μ_1 (in that same structure), and ψ is true over the suffix m-path, μ_2. That is, if $\mu = \mu_1 \bullet \mu_2$.
– $\phi \mid \psi$: *ϕ and ψ execute concurrently, in an interleaved fashion.* That is, $\phi \mid \psi$ is true over an m-path μ in a multi-path structure if ϕ is true over an m-path μ_1 (in that same structure), ψ is true over an m-path μ_2, and μ is one of the interleavings $\mu_1 \| \mu_2$.
– $\phi \wedge \psi$: *ϕ and ψ execute along the* same *path.* That is, $\phi \wedge \psi$ is true on an m-path μ if both ϕ and ψ are true on μ. In practice, this is best understood in terms of *constraints* on execution. For instance, ϕ can be thought of as a non-deterministic transaction and ψ as a constraint on the execution of ϕ. It is this feature of the logic that lets us specify constraints as part of service contracts.

- $\phi \vee \psi$: *execute* ϕ or *execute* ψ *non-deterministically*. That is, $\phi \vee \psi$ is true on an m-path μ if either ϕ or ψ is true on μ.
- $\neg\phi$: *execute in any way provided that this will* not *be a valid execution of* ϕ. That is, $\neg\phi$ is true on any m-path on which ϕ is not true. Negation is an important ingredient in temporal constraint specifications.
- $\odot\phi$: *execute* ϕ *in isolation, i.e., without interleaving with other concurrently running tasks*. That is, $\odot\phi$ is true on any *singleton* m-path (an m-path that contains just one path) where ϕ is true. Note: $\odot\phi$ is never true on an m-path that consists of more than one path, so the execution of $\odot\phi$ cannot be broken by other executions. This operator enables us to specify non-interleaved parts of service contacts.

When considering the entire service, we are interested in its executions over paths, not m-paths: executions over m-paths are used only to represent concurrently running subtasks of the service. To complete the picture, we define truth of CTR formulas over paths:

- ϕ is true over a path, π, if it is true over an m-path, μ, and π is a merge of μ.

CTR contains a special propositional constant, `state`, which is true only on paths of length 1, that is, on database states. In service processes, `state` is often used as the exit condition for iterative tasks. Another propositional constant that we will use to represent constraints is `path`, defined as `state` $\vee \neg$`state`; this constant is true on every path.

Concurrent-Horn subset of CTR. The implication $p \leftarrow q$ is defined as $p \vee \neg q$. The form and the purpose of the implication in CTR is similar to that of Datalog: p can be thought of as the name of a procedure and q as the definition of that procedure. However, unlike Datalog, both p and q take truth values over execution paths, not at individual states.

More precisely, $p \leftarrow q$ means: if q can execute along a path $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$, then so can p. If p is viewed as a task name, then the meaning can be re-phrased as: one way to execute task p is to execute its definition, q.

To specify service processes we use *concurrent-Horn goals* and *concurrent-Horn rules*.

Definition 3.1 (Concurrent-Horn goal) A *concurrent-Horn goal* is either an atomic formula or has the form $\phi \otimes \psi$, $\phi \mid \psi$, $\phi \vee \psi$, or $\odot\phi$, where ϕ and ψ are concurrent-Horn goals.

When confusion does not arise, we will often talk about CTR goals, omitting the "concurrent-Horn" adjective. □

Concurrent-Horn goals occur in our setting in two places: as bodies of the rules that are used to define composite tasks and as formulas that are formal embodiments of control flow graphs. In the latter case, we will be interested in finding out whether a control flow graph can be enacted. Such a question corresponds to proving a statement of the form $\exists \overline{X}\phi$, where ϕ is a CTR goal and $\overline{X}$ is the set of variables that occur in ϕ.

Definition 3.2 (Concurrent-Horn rule) A *concurrent-Horn rule* is a CTR formula of the form

$$\forall \overline{X} \; (head \leftarrow body) \tag{1}$$

where $head$ is an atomic formula, $body$ is a concurrent-Horn goal, and $\overline{X}$ is the set of variables that occurs in $head$ and $body$. □

Since all variables in a rule are quantified the same way (universally outside of the rule), we will usually omit explicit quantifiers—a common practice that simplifies the notation.

The concurrent-Horn fragment of CTR has an SLD-style proof procedure that proves concurrent-Horn formulas and *executes* them at the same time [11]. The present paper significantly extends this proof theory to formulas that contain the $\wedge$ connective thus enabling execution of *constrained transactions*, which are non-Horn. We also deal with a much larger class of constraints than [16,38], including iterative processes.

Primitive updates. In CTR, *primitive updates* are *ground* (i.e., variable-free) atomic formulas that change the underlying database state. Semantically they are represented by binary relations over state identifiers. For instance, if $\langle \mathbf{s}_1, \mathbf{s}_2 \rangle$ belongs to the relation corresponding to a primitive update u, it means that u can cause a transition from state $\mathbf{s}_1$ to state $\mathbf{s}_2$. We will conveniently represent this kind of situation using the following notation:

$$\mathbf{s}_1 \xrightarrow{u} \mathbf{s}_2 \tag{2}$$

Usually the binary relations that represent primitive updates are defined outside of CTR. In that case, they are called *transition oracles* [8,11,10,9]. Transition oracles can be defined using formal English or a number of other formal languages. They can also be represented *in* CTR as *partially defined actions* [36]. In either case, the primitive updates can be defined to perform any kind of transformation. For instance, they can add or delete single tuples or sets of tuples, add and delete entire relations, and so on.

In the examples, we will be representing primitive updates using predicate symbols that have variables (e.g., *place_order*(*Order#,Price*)). It should be understood that such a predicate represents a *family* of related updates, one for each instantiation of the variables. Clearly, *place_order*(`12365409, $123`)) and *place_order*(`09865412, $321`)) cause similar, but different state transitions.

Constraints. Because formulas are defined on paths, CTR can express a wide variety of constraints on the way formulas may execute. One can place existential constraints on execution (these are based on serial conjunction), or universal constraints, which are based on serial implication. To express the former, we use the propositional constant `path` introduced above. For example, `path` $\otimes\, \psi \otimes$ `path` is a constraint that is true on a path if ψ is true *somewhere* on that path. To express universal constraints, the binary connectives "$\Leftarrow$" and "$\Rightarrow$" are used, which are defined via $\otimes$ and $\neg$ as follows: $\psi \Leftarrow \phi \overset{def}{=} \neg(\neg\psi \otimes \phi)$ and $\psi \Rightarrow \phi \overset{def}{=} \neg(\psi \otimes \neg\phi)$). A moment's reflection should convince the reader that $\psi \Leftarrow \phi$ means that whenever ϕ occurs then ψ must have occurred just before it and that $\psi \Rightarrow \phi$ means that whenever ψ occurs then ϕ must occur right after it. Thus, `path` $\Rightarrow \psi \Leftarrow$ `path` constrains executions to be such that *every* subpath encountered in the course of the execution satisfies ψ (including subpaths of the form $\langle \mathbf{s} \rangle$, where $\mathbf{s}$ is an arbitrary intermediate state).

Executional entailment. The notion of *executional entailment* is the key semantic concept in CTR that brings the informal notion of execution into the logic. Let **P** be a set of CTR formulas, ϕ is a CTR formula and $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_n$ is a sequence of database states. Then the statement

$$\mathbf{P}, \mathbf{s}_0\, \mathbf{s}_1\, \ldots\, \mathbf{s}_n \models \phi \tag{3}$$

is true if and only if $\mathbf{M}, \langle \mathbf{s}_0,\ \mathbf{s}_1,\ \ldots,\ \mathbf{s}_n \rangle \models \phi$ (i.e., ϕ is true on the path $\langle \mathbf{s}_0,\ \mathbf{s}_1,\ \ldots,\ \mathbf{s}_n \rangle$ in **M**), for every multi-structure **M** that satisfies **P**. Related to this is the statement

$$\mathbf{P}, \mathbf{s}_0 \text{---} \models \phi \tag{4}$$

which are true iff (3) is true for some sequence of database states $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_n$.

The aforementioned proof theory for CTR assumes that **P** is a set of concurrent-Horn rules and it manipulates the statements of the form $\mathbf{P}, \mathbf{s} \text{---} \vdash \phi$. It is sound and complete in the sense that there is a proof of $\mathbf{P}, \mathbf{s}_0 \text{---} \vdash \phi$ if and only if $\mathbf{P}, \mathbf{s}_0 \text{---} \models \phi$ is true.

4. Formalizing Service Contracts in *ServLog*

This section formally defines the core modeling elements of *ServLog*. First we define service processes directly in CTR. Section 4.2 then introduces service policies and contract requirements as constraints that can be expressed in CTR. Section 4.3 then defines an important notion, which we call the *service contract assumption*.

4.1. Modeling Service Processes

Definition 4.1 (Task) A *task* is represented by a predicate. The name of the predicate is the *name of that task* and the arity specifies the number of arguments that the predicate takes. For notational simplicity, we assume that each predicate name has exactly one arity, so each task is uniquely defined by its name. □

In service contract specifications, the actual invocations of tasks are represented by task atoms.

Definition 4.2 (Task atom) A *task atom* is a statement of the form

$$p(t_1, \ldots, t_n) \tag{5}$$

where p is a task predicate of arity n ($n \geq 0$) and $t_1, \ldots, t_n$ are terms (defined as in first-order logic) that represent the arguments that p takes. The terms representing the arguments of the task are placeholders for data items that the task manipulates (its inputs and outputs).

For brevity, we will often write task atoms as $p(\overline{T})$, $p(\overline{U})$, etc., where $\overline{T}$, $\overline{U}$, ... stand for the tuples of arguments that the task takes. □

When confusion does not arise, the term "task" will refer both to tasks and task atoms.

We distinguish between two main types of task predicates: *composite* and *primitive*. Composite task predicates are the ones defined by rules (i.e., they are allowed in the rule heads) and primitive tasks are *not* allowed in the rule heads. The primitive task predicates are further subdivided into *update-tasks*, *query-tasks*, and *builtin test tasks*. The update task predicates are those used as the primitive updates of CTR, the query tasks are the ones whose predicates are used to represent the facts stored in database states, and the builtin tests use predicates whose truth is independent of the database state. These three categories of predicates are disjoint. The transition conditions on the arcs of service process graphs, which were introduced in Section 2, are also treated in *ServLog* as tasks: specifically as query tasks or builtin tests—whichever applies in each case.

In this paper, we will be using the builtins "=" (identity), "!=" (distinct values), ">", "<", and others, as needed. The identity predicate $a = b$ is true if and only if a and b are the same *ground* (i.e., variable-free) term and $a! = b$ holds if a and b are distinct ground terms. Clearly, the truth of these predicates is independent of the database state (or of a path) where the predicate is evaluated.

From now on, when talking about CTR goals and rules, we assume that the atomic formulas are task atoms only. In addition, the predicates occurring in the rule heads must correspond to composite tasks only.

Definition 4.3 (Task occurrence) A task p $occurs$ in a CTR goal Ω if Ω contains a task atom $p(\overline{T})$ for some $\overline{T}$. □

Definition 4.4 (Immediate subtask) Let p and q be a pair of tasks and **R** be a set of rules. We say that p is an *immediate subtask* of q with respect to **R** if and only if **R** contains a rule of the form $q(\overline{T}) \leftarrow \Omega$ and p occurs in Ω. □

Definition 4.5 (Subtasks) Let p and q be a pair of tasks and **R** be a set of rules. Then p is a $subtask$ of q with respect to **R** if and only if p is either an immediate subtask of q or there is an immediate subtask r of q such that p is a subtask of r. □

Definition 4.6 (Non-iterative rule) A rule in **R** is *non-iterative* if and only if it has the form

$$q(\overline{T}) \leftarrow \Omega \tag{6}$$

where q does not occur in Ω and none of the tasks that occur in Ω have q as a subtask. □

Here is an example of a non-iterative rule (p and r are assumed to be primitive tasks here): $q(?X) \leftarrow p(?X, ?Y) \otimes r(?Y)$. As seen in this example, variables in *ServLog* are represented as symbols prefixed with "?".

Definition 4.7 (Iterative rule) A rule $q(\overline{T}) \leftarrow \Omega$ in **R** is $iterative$ if and only if q either occurs in Ω directly or it is a subtask of a task that occurs in Ω. □

Here are examples of iterative rules:

$$q(?X) \leftarrow (p(?X,?Y) \otimes q(?Y) \otimes r(?Y,?Z)) \vee s(?Y)$$
$$q(?X) \leftarrow p(?X,?Y) \otimes q(?Y) \otimes qq(?Y,?Z)$$
$$qq(?X) \leftarrow (pp(?X,?Y) \otimes q(?Y)) \vee t(?X)$$

Note that here q and qq are both iterative tasks that are mutually dependent on each other (are subtasks of each other). In practice, however, the most common form of iterative tasks is a loop of the form

$$q(\overline{T}) \leftarrow \Phi \otimes q(\overline{U})$$
$$q(\overline{T}) \leftarrow \Psi$$

where Φ and Ψ do not depend on q.

Definition 4.8 (Service process) A *service process* is a pair $(\Omega, \mathbf{R})$, where Ω is a CTR goal and **R** is a set of iterative and non-iterative rules whose heads are task atoms of the tasks that occur in Ω or are subtasks of these tasks. □

Recall that primitive tasks come in three guises: updates, queries, and builtins. Similarly, we classify composite tasks based on the rules that define them. Namely, if a task is defined by at least one iterative rule (i.e. there exists an iterative rule with the task as its head), we call the task *iterative*; if it is defined only by non-iterative rules then the task is *non-iterative*.

Equipped with this mechanism for defining service processes in *ServLog*, one can capture a wide range of control and data flow constructs that often appear in business process languages and notations. For example, the service process introduced in Figure 2 is represented in *ServLog* as shown in Figure 4.

The top-level graph is specified as a CTR goal at the very beginning. The tasks appearing in that goal are defined by the rules that follow. This service process illustrates data flow through variables as well as via a shared database. For example, passing data from **process_order_items** to **book_shipper** is done via the underlying database by having **process_order_items** insert $selected\_item(?Order\#, ?Item)$ and then querying this data item by the task **handle_shippers**. To fully capture the dataflow, we introduce three additional database query predicates: *producer(?Item,?Producer)*, which returns a producer for the given item, *itemAvailable(?Order#,?Producer,?Item)*, which is true if the given item is produced by the given producer and the item is in stock, and *itemUnavailable(?Order#,?Producer,?Item)*, which is the negation of *itemAvailable*. While the set of available producers is relatively static, the relation *itemAvailable* can be modified by the task **contact_producer***(?Producer,?Item)*. For instance, after contacting the producer an item might become reserved.

Data flow types supported by *ServLog* are simple yet powerful: tasks can share data through shared variables or through the underlying shared database — the former is standard in classical logic and in logic programming languages, the latter is a feature of CTR. For instance, in the subprocess **process_item** in Figure 4, the same data is passed through the shared variables $?Order\#$ and $?Item$ to the query *producer*, and also other tasks (e.g., **contact_producer**). These data come from the task **process_order_items** and then are passed along to the top-level invocation of **process_item**. Inside **process_item**, new data is obtained by the query *producer* and then is passed to the subtasks **contact_producer** and **inform_client** through the shared variable *?Producer*.

In Section 2, we explained the nature of primitive updates as “black boxes” whose inner workings are hidden from *ServLog*'s reasoning mechanism. In Figure 4, for example, the tasks **place_order**, **select_item**, and some others are said to be primitive CTR updates that correspond to primitive tasks in Figure 2 and their implementation is opaque to the system. However, as explained there, *ServLog* lets the service logic designer to represent tasks at different levels of abstraction and primitive tasks may be expanded into complex tasks, if desired. Figure 5 illustrates this point using some earlier primitive tasks as an example.

4.2. Modeling Constraints

We now formalize service policies and contract requirements as constraints in *ServLog*. Note that constraints are not defined directly as CTR formulas (unlike task definitions). The main reason for this is that constraints represent

```
Goal:
place_order($?Order\#, ?Price$) $\otimes$                          // primitive update
(process_order_items($?Order\#$) |                           // composite tasks
  handle_shippers($?Order\#$) |
  handle_payment($?Order\#, ?Price$)) $\otimes$
end_order($?Order\#$)                                         // primitive update

Rules:
process_order_items($?Order\#$) $\leftarrow$                        // composite task
    select_item($?Order\#, ?Item$)$\otimes$
    $insert.selected\_item(?Order\#, ?Item)$ $\otimes$     //primitive update that inserts $selected\_item(...)$ into database
    process_item($?Order\#, ?Item$)$\otimes$
    process_order_items($?Order\#$)
process_order_items($?Order\#$) $\leftarrow$ state
process_item($?Order\#, ?Item$) $\leftarrow$                        // composite task
    $producer(?Item, ?Producer)$$\otimes$
    contact_producer($?Order\#, ?Producer, ?Item$)$\otimes$
    inform_client($?Order\#, ?Producer, ?Item$)
handle_shippers($?Ord\#$) $\leftarrow$
    contact_shipper($?Ord\#, ?Shipper$)$\otimes$
    $selected\_item(?Ord\#, ?Itm)$ $\otimes$                   // database query
    book_shipper($?Ord\#, ?Shipper, ?Itm$)$\otimes$
    deliver($?Ord\#$)
handle_payment($?Order\#, ?Price$) $\leftarrow$ pay($?Order\#$)
handle_payment($?Order\#, ?Price$) $\leftarrow$ payment_guarantee($?Order\#, ?Credit\#$)
inform_client($?Order\#, ?Producer, ?Item$) $\leftarrow$
    $itemAvailable(?Order\#, ?Producer, ?Item)$ $\otimes$              // database query
    inform_availability($?Order\#, ?Producer, ?Item$) $\otimes$
    confirm_producer($?Order\#, ?Producer$)
inform_client($?Order\#, ?Producer, ?Item$) $\leftarrow$
    $itemUnavailable(?Order\#, ?Producer, ?Item)$ $\otimes$            // database query
    inform_unavailability($?Order\#, ?Producer, ?Item$)         // primitive update
pay($?Order\#$) $\leftarrow$ full_payment($?Order\#$) $\vee$ pay_per_item($?Order\#$)
pay_per_item($?Order\#$) $\leftarrow$ pay_one_item($?Order\#$) $\otimes$ pay_per_item($?Order\#$)
pay_per_item($?Order\#$) $\leftarrow$ state
```

Fig. 4. *ServLog* representation of the service process from Figure 2.

patterns that executions of service processes must follow and specialized language constructs for such patterns make specification of constraints easier to understand. Nevertheless, the constraints of *ServLog* can be expressed as CTR formulas (see Appendix C), so CTR is indeed used here as a unifying formalism for both service task definition and constraints. Recall that whereas CTR can represent constraints, they are not Concurrent Horn formulas and are therefore not handled by the existing CTR proof theory — the extension of the CTR proof theory to handle constraints is proposed in Section 5.2 and is an important contribution of this paper.

A constraint specifies the rules governing the occurrences of various tasks during the execution of a service. Each occurrence of a task is represented by a pattern, which specifies the task name and various conditions on the arguments with which that task can be invoked during the execution. These conditions can require that certain arguments must be bound to specific values and they can also require that certain arguments must be shared within a task occurrence or across the occurrences of different tasks.

Definition 4.9 (Task Pattern) A *task pattern* has the form $p(t_1, ..., t_n)$ where each t_i is either a regular ground term (of the kind that may occur in a task atom) or a *placeholder*. A placeholder is either a named logical variable (which

```
place_order(?Order#,?Price) ←
    generate_order_number(?Order#) ⊗        // a builtin; instantiates  ?Order#
    get_item_list(?Itemlist) ⊗               //  get items from the user; a builtin using a Web form
    compute_price(?Itemlist,?Price) ⊗        //  a builtin
    get_private_info(?Name,?Address) ⊗       //  a builtin
    insert.status(?Order#,processing) ⊗
    save_order_in_db(?Order#,?ItemList,?Name,?Address)     //  primitive update; can be expanded further
end_order(?Order#) ←
    delete.status(?Order#,?) ⊗
    insert.status(?Order#,complete)
select_item(?Order#,?Item) ←
    order_items(?Order#,?Itemlist) ⊗              // a query
    select(?Item,?Itemlist,?ItemlistSansItem) ⊗   // a builtin: picks ?Item from ?Itemlist & creates ?ItemlistSansItem,
                                                  // as ?Itemlist with ?Item removed
    delete.order_items(?Order#,?Itemlist) ⊗
    insert.order_items(?Order#,?ItemlistSansItem) ⊗
    decrement_stock_quantity(?Item)              // primitive update; can be expanded further
```

Fig. 5. Expansion of some primitive updates from Figure 4.

will be designated with the prefix '_', e.g., `_Ord#`) or a *don't care* placeholder, denoted by `'_'`. Each occurrence of a don't care placeholder represents a *new* logical variable that does not occur in other patterns. □

We will often need to perform two operations: *matching* and *refinement*. The former is the usual matching operator of first-order logic: it is a substitution, θ such that $\theta(pattern) = task\_atom$. Since $task\_atom$ is ground, θ will normally be a ground substitution. In this case we will say that the pattern and the ground task *match*. Note that different occurrences of `'_'` may be mapped by θ to different constants, since such occurrences represent different logical variables. Refinement is defined next.

Definition 4.10 (Refinement) The *refinement operation* takes a ground task atom and a pair of task patterns and yields another task pattern as follows:

$$\texttt{refine}(out\_pattern;\ in\_ground\_task,\ in\_pattern) = refined\_pattern$$

Here the arguments *in_ground_task* and *in_pattern* must have the same task name and *in_ground_task* must match *in_pattern*. The task-pattern *out_pattern* may have a different task name. The result of the operation, *refined_pattern*, is defined as follows: Let θ be the substitution that matches *in_pattern* against *in_ground_task*. If *out_pattern* has variables other than those in *in_pattern*, θ can map them to anything (to some other variable or constant). Then

$$refined\_pattern = \theta(out\_pattern)$$

□

One can verify by direct inspection that the task atom `p(2,1,abc,cde,13,cde,13,5)` matches the pattern `p(_,1,abc,_foo,_bar,_foo,_bar,_)`, that `refine(p(_ff,5,_);p(1,2,3), p(_,_ff,3)) = p(2, 5,_)`, and that `refine(q(_f,5,_f,_,_h);p(1,2,2, 3),p(_g,_f,_f,3)) = q(2,5,2, _,_h)`. The last example also illustrates the situation where *in_pattern* has named placeholders that do not occur in *out_pattern*; the number of arguments in the input and output patterns can also differ.

Definition 4.11 (Constraints) In this definition, we will use $\overline{t}$, $\overline{u}$, etc., to represent tuples that include placeholders as some of the arguments in task patterns. The uppercase symbols $\overline{T}$, $\overline{U}$, etc., will denote tuples of arguments in task atoms (i.e., they do not contain placeholders). The task names p, q, r, and the task patterns mentioned in the constraints, below, do not need to be distinct.

The set $\mathcal{C}\textsc{onstr}$ of constraints supported by *ServLog* is formally defined as follows. For each constraint we first give its syntax followed by a brief informal explanation and then provide a formal semantic definition. Appendix C provides alternative representation of these constraints as CTR formulas.

1. **Existence constraints**:

   – $\texttt{atleast}_n(p(\bar{t}))$: *task p must execute at least n times ($n \geq 1$).*

     Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if there are ground task atoms $p(\overline{T_1})$, ..., $p(\overline{T_n})$ that executed at some states $\mathbf{s}_{i_1}$, ..., $\mathbf{s}_{i_n}$ (i.e., $\mathbf{s}_{i_1} \xrightarrow{p(\overline{T_1})} \mathbf{s}_{i_1+1}$, ..., $\mathbf{s}_{i_n} \xrightarrow{p(\overline{T_n})} \mathbf{s}_{i_n+1}$) such that $p(\bar{t})$ matches $p(\overline{T_1})$, $p(\overline{T_2})$, ..., $p(\overline{T_n})$.[12]

   – $\texttt{absence}(p(\bar{t}))$: *task p must not execute.*

     Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if there is no state $\mathbf{s}_i$ in that execution such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$ and $p(\bar{t})$ matches $p(\overline{T})$.

   – $\texttt{exactly}_n(p(\bar{t}))$: *task p must execute exactly n times ($n \geq 1$).*

     An execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if it satisfies $\texttt{atleast}_n(p(\bar{t}))$ but not $\texttt{atleast}_{n+1}(p(\bar{t}))$.

2. **Serial constraints**:

   – $\texttt{after}(p(\bar{t})\ \texttt{-->}\ q(\bar{u}))$: *whenever p executes, q must execute after it.* Task q is *not* required to execute immediately after p, and several other instances of p might execute before q actually does.

     Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution, such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$ and $p(\bar{t})$ matches $p(\overline{T})$, there must be a state $\mathbf{s}_j$ in that same execution such that $j \geq i + 1$, $\mathbf{s}_j \xrightarrow{q(\overline{U})} \mathbf{s}_{j+1}$, and $\texttt{refine}(q(\bar{u});p(\overline{T}),p(\bar{t}))$ matches $q(\overline{U})$.

     For instance, if the above constraint has the form $\texttt{after}(p(\_foo, \_)\ \texttt{-->}\ q(\_, \_foo))$ then the sequence $p(a, 1), q(2, a)$ is a valid execution, but $p(a, 1), q(2, b)$ is not.

   – $\texttt{before}(p(\bar{t})\ \texttt{<--}\ q(\bar{u}))$: *whenever q executes, it must be preceded by an execution of p.* Task p does not have to execute immediately prior to q.

     An execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ is said to satisfy this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{q(\overline{U})} \mathbf{s}_{i+1}$ and $q(\bar{u})$ matches $q(\overline{U})$, there must be a state $\mathbf{s}_j$ in that same execution such that $j \leq i - 1$, $\mathbf{s}_j \xrightarrow{p(\overline{T})} \mathbf{s}_{j+1}$, and $\texttt{refine}(p(\bar{t});q(\overline{U}),q(\bar{u}))$ matches $p(\overline{T})$.

     For instance, if the constraint is $\texttt{before}(p(\_foo, \_)\ \texttt{<--}\ q(\_, \_foo))$ then the sequence $p(a, 1), q(2, a)$ is a valid execution, but $p(a, 1), q(2, b)$ is not.

   – $\texttt{blocks}(p(\bar{t})\ \texttt{-/>}\ q(\bar{u}))$: *if task p executes then task q cannot execute in the future.*

     Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$ and $p(\bar{t})$ matches $p(\overline{T})$, there is *no* state $\mathbf{s}_j$ in that execution such that $j > i$, $\mathbf{s}_j \xrightarrow{q(\overline{U})} \mathbf{s}_{j+1}$, and $\texttt{refine}(q(\bar{u});p(\overline{T}),p(\bar{t}))$ matches $q(\overline{U})$.

[12]The notation $s \xrightarrow{p} s'$ was introduced in (2) in Section 3. Recall that since $p(\overline{T_1})$, ..., $p(\overline{T_n})$ cause state transitions, they are primitive update tasks.

- between($p(\bar{t})$ $\dashrightarrow$ $q(\overline{u})$ $\dashleftarrow$ $r(\overline{v})$): *task q must execute between any two executions of p and r.* That is, after an execution of p, any subsequent execution of r has to wait until q is executed.

  An execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there are states $\mathbf{s}_i$, $\mathbf{s}_k$ $(k > i + 1)$ such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$, $\mathbf{s}_k \xrightarrow{r(\overline{V})} \mathbf{s}_{k+1}$, $p(\bar{t})$ matches $p(\overline{T})$, and $r(\overline{v})$ matches $r(\overline{V})$, then there must be a state $\mathbf{s}_j$ such that $i < j < k$, $\mathbf{s}_j \xrightarrow{q(\overline{U})} \mathbf{s}_{j+1}$, and both refine($q(\overline{u})$;$p(\overline{T})$,$p(\bar{t})$) and refine($q(\overline{u})$;$r(\overline{V})$,$r(\overline{v})$) match $q(\overline{U})$.

- not_between($p(\bar{t})$ $\not\dashrightarrow$ $q(\overline{u})$ $\not\dashleftarrow$ $r(\overline{v})$): *task q must not execute between any pair of executions of p and r.* Thus, if q executes after p, no future execution of r is possible.

  An execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there are states $\mathbf{s}_i$, $\mathbf{s}_k$ $(k > i + 1)$ such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$, $\mathbf{s}_k \xrightarrow{r(\overline{V})} \mathbf{s}_{k+1}$, $p(\bar{t})$ matches $p(\overline{T})$, and $r(\overline{v})$ matches $r(\overline{V})$, then there is *no* state $\mathbf{s}_j$ such that $i < j < k$, $\mathbf{s}_j \xrightarrow{q(\overline{U})} \mathbf{s}_{j+1}$, and both refine($q(\overline{u})$;$p(\overline{T})$,$p(\bar{t})$) and refine($q(\overline{u})$;$r(\overline{V})$,$r(\overline{v})$) match $q(\overline{U})$.

3. **Immediate serial constraints**:

- right_after($p(\bar{t}) \rightarrow q(\overline{u})$): *whenever p executes, q must execute immediately after it.*

  Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$ and $p(\bar{t})$ matches $p(\overline{T})$, then $\mathbf{s}_{i+1} \xrightarrow{q(\overline{U})} \mathbf{s}_{i+2}$ must hold and refine($q(\overline{u})$;$p(\overline{T})$,$p(\bar{t})$) must match $q(\overline{U})$.

- right_before($p(\bar{t}) \leftarrow q(\overline{u})$): *whenever q executes, p must have been executed immediately before it.*

  Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ satisfies this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{q(\overline{U})} \mathbf{s}_{i+1}$ and $q(\overline{u})$ matches $q(\overline{U})$, then $\mathbf{s}_{i-1} \xrightarrow{p(\overline{T})} \mathbf{s}_i$ must hold and refine($p(\bar{t})$;$q(\overline{U})$,$q(\overline{u})$) must match $p(\overline{T})$.

- not_right_after($p(\bar{t}) \not\rightarrow q(\overline{u})$): *whenever p and q execute, q must not execute immediately after p.* That is, after p there must be an execution of a task other than q before q is allowed again.

  An execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m \rangle$ is said to satisfy this constraint if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{p(\overline{T})} \mathbf{s}_{i+1}$, where $p(\bar{t})$ matches $p(\overline{T})$, and $\mathbf{s}_{i+1} \xrightarrow{r(\overline{U})} \mathbf{s}_{i+2}$ for some $r(\overline{U})$, then refine($q(\overline{u})$,$p(\overline{T})$,$p(\bar{t})$) must *not* match $r(\overline{U})$.

4. **Composite constraints**: If C_1, $C_2 \in \mathcal{CONSTR}$ then so are $C_1 \wedge C_2$ (a conjunctive constraint) and $C_1 \vee C_2$ (a disjunctive constraint).

Nothing else is in $\mathcal{CONSTR}$. □

Note the use of different arrows between the arguments in some of the constraints. The convention here is that the task at the tail of the arrow represents the condition of the constraint (*if or whenever the task executes*) and the task at the head of the arrow indicates the effect of the constraint (*the task must or must not execute in a given relationship to the task at the tail of the arrow*). We use strong arrows to indicate immediacy (*execution must take place right before or after*) and dashed arrows indicate that immediacy is not required. Slashed arrows indicate negative relationships (e.g., the task in the head must *not* execute). Note also that the negation of right_before can be defined as follows:

not_right_before($p(\bar{t}) \not\leftarrow q(\overline{u})$) $\equiv$ not_right_after($p(\bar{t}) \not\rightarrow q(\overline{u})$).

The following is an example of a legal constraint in $\mathcal{CONSTR}$:

$$\begin{aligned}&\texttt{atleast}_2(p(\_X,1,\_X)) \wedge \texttt{exactly}_3(q(\_,\_,\_))\\ &\quad\wedge\ \texttt{right\_after}(p(\_X,\_,\_) \rightarrow r(\_,\_X))\end{aligned} \tag{7}$$

The constraint $\texttt{atleast}_2(p(\_X,1,\_X))$ requires that p executes at least twice and arguments 1 and 3 have the same value in each execution, while the second argument is the integer 1. In $\texttt{right\_after}(p(\_X,\_,\_) \rightarrow r(\_,\_X))$, the placeholder $\_X$ is shared between p and r. This means that whenever p is executed, r must follow immediately and r's second argument must be the same as the first argument in p.

We can now show how the constraints from Figure 3 can be represented in *ServLog*:

1. $(\texttt{atmost}_6(\textbf{accept}(\_,\_)) \wedge \texttt{absence}(\textbf{book\_shipper}(\_,\_,\_)))$
   $\vee\ (\texttt{atleast}_7(\textbf{accept}(\_,\_))$
   $\wedge\ \texttt{after}(\textbf{accept}(\_Ord\#,\_) \dashrightarrow \textbf{book\_shipper}(\_Ord\#,\_,\_))$
   where $\texttt{atmost}_n(p)$ is a shorthand for
   $\texttt{absence}(p) \vee \texttt{exactly}_1(p) \vee \texttt{exactly}_2(p) \vee \ldots \vee \texttt{exactly}_n(p)$
2. $\texttt{absence}(\textbf{pay\_per\_item}(\_))$
   $\vee\ \texttt{right\_before}(\textbf{pay\_one\_item}(\_Ord\#) \leftarrow \textbf{deliver}(\_Ord\#))$
3. $\texttt{before}(\textbf{payment\_guarantee}(\_Ord\#,\_) \dashleftarrow \textbf{book\_shipper}(\_Ord\#,\_,\_))$
4. $\texttt{exactly}_1(\textbf{deliver}(\_))$
5. $\texttt{absence}(\textbf{full\_payment}(\_))\ \vee$
   $(\texttt{before}(\textbf{deliver}(\_Ord\#) \dashleftarrow \textbf{full\_payment}(\_Ord\#))$
   $\wedge\ \texttt{blocks}(\textbf{full\_payment}(\_Ord\#) \not\dashrightarrow \textbf{deliver}(\_Ord\#)))$
6. $\texttt{before}(\textbf{pay}(\_Ord\#) \dashleftarrow \textbf{book\_shipper}(\_Ord\#,\_,\_))$

Many other types of constraints can be naturally expressed in $\mathcal{CONSTR}$, as shown below:

- $\texttt{atmost}_n(p(\_,1))$ — task p can execute at most n times and each time the second argument must be 1. This constraint was defined in item 1 above.
- $\texttt{absence}(p(\_,4)) \vee \texttt{atleast}_1(q(2,\_,3))$ — if p is executed with its second argument 4, then q must also execute (before or after p) and its first and last arguments must be 2 and 3 respectively.
- $(\texttt{absence}(p()) \vee \texttt{atleast}_1(q())) \wedge (\texttt{absence}(q()) \vee \texttt{atleast}_1(p()))$ — if p is executed, then q must also be executed, and vice versa.
- $\texttt{after}(p(\_X) \dashrightarrow q(\_X)) \wedge \texttt{before}(p(\_X) \dashleftarrow q(\_X))$ — every occurrence of task p must be followed by an occurrence of task q with the same argument and there must be an occurrence of p before every occurrence of q and their arguments must be the same.
- $\texttt{absence}(p(\_)) \vee \texttt{between}(p(\_X) \dashrightarrow q(\_X) \dashleftarrow p(\_))$ — if task p is executed then q must execute after it, with the same argument, and before that q there can be no other p.
- $\texttt{absence}(q(\_)) \vee (\texttt{before}(p(\_) \dashleftarrow q(\_)) \wedge \texttt{between}(q(\_) \dashrightarrow p(\_) \dashleftarrow q(\_)))$ — if task q is executed, it has to be preceded by an occurrence of p. The next instance of q can execute only after another occurrence p.
- $\texttt{between}(p(\_X) \dashrightarrow q(\_X) \dashleftarrow p(\_X)) \wedge \texttt{between}(q(\_X) \dashrightarrow p(\_X) \dashleftarrow q(\_X))$ — tasks p and q must alternate when they execute with the same argument.
- $\texttt{right\_after}(p(\_) \rightarrow q(\_)) \wedge \texttt{right\_before}(p(\_) \leftarrow q(\_))$ — executions of p and q must be next to each other with no intervening tasks in-between.
- $\texttt{absence}(p(\_)) \vee \texttt{absence}(q(\_))$ — it is not possible for p and q to execute in the same service process run.
- $\texttt{not\_between}(p(\_X) \not\dashrightarrow q(\_) \not\dashleftarrow p(\_X)) \wedge \texttt{not\_between}(q(\_X) \not\dashrightarrow p(\_) \not\dashleftarrow q(\_X))$ — q must not execute between any two executions of p with the same arguments, and p must not execute between any two executions of q with the same arguments.

4.3. The Service Contract Assumption

We now introduce modeling assumptions, which tighten the form of the constraints and tasks that are allowed in service processes. These assumptions do not limit the modeling power of the language in the sense that any service process can be represented by another process that satisfies these assumptions. However, these assumptions greatly simplify the reasoning system of Section 5.

Definition 4.12 (Service Contract Assumption) A service process G and a set of constraints $\mathcal{C}$ satisfy the *service contract assumption* if and only if the set of constraints $\mathcal{C}$ is *based on primitive update tasks* and the primitive update tasks of G satisfy the *independence* assumption. □

The last two notions in this definition are spelled out in Definitions 4.13 and 4.14 below. Also recall that a primitive task is one that is not defined by a rule and a primitive update task is just a primitive CTR update.

Definition 4.13 (Constraints based on primitive update tasks) A set of constraints is *based on primitive update tasks* if and only if all tasks appearing in the constraints are primitive update tasks. □

This restriction does not limit the modeling power of the language, since one can always instrument composite tasks in such a way that the resulting set of constraints will be based on primitive update tasks. More specifically, one can insert "bounding" primitive update tasks at various locations in the definition of composite tasks, and then transform constraints on composite tasks into constraints on those bounding primitive update tasks. These bounding tasks are defined as no-ops and their only purpose is to capture the various stages in the life cycle of a task. Examples include the beginning and end of a task, the beginning and end of an iteration, and so on.

The rationale behind restricting constraints to primitive update tasks is that specifying constraints directly over composite tasks can be highly ambiguous. For instance, what should the sentence "task `b` must start after task `a`" mean exactly? Should `b` start after `a` begins or after `a` ends? Similar ambiguity exists with other constraints, such as *before* and *between* constraints. Requiring that constraints are based on primitive update tasks avoids ambiguity and complications without limiting the modeling power.

The following example illustrates the process of inserting bounding tasks to delineate the beginning and the end of a composite task:

– Every $non\text{-}iterative$ composite task of the form $p \leftarrow \Omega$ can be changed to:

$$p \leftarrow p_{start} \otimes \Omega \otimes p_{end}$$

– Every $iterative$ composite task, for example, of the form $p \leftarrow (\Phi \otimes p) \vee \Psi$, can be changed to:

$$p \leftarrow (p_{start} \otimes \Phi \otimes p \otimes p_{end}) \vee (p_{start} \otimes \Psi \otimes p_{end})$$

In fact, there are many other ways to insert bounding tasks, which would enable many more kinds of constraints. For instance,

$$p \leftarrow p_{start} \otimes \Phi_{start} \otimes \Phi \otimes \Phi_{end} \otimes p \otimes p_{end}$$

These bounding tasks are regular primitive updates that insert unique tokens every time they execute. For example, p_{start} might insert $token(p_{start}, 0)$, $token(p_{start}, 1)$, $token(p_{start}, 2)$, and so on, on each successive execution.

A constraint such as $\texttt{after}(p \dashrightarrow q)$, where p and q are composite tasks, can now be interpreted as $\texttt{after}(p_{start} \dashrightarrow q_{start})$, or $\texttt{after}(p_{end} \dashrightarrow q_{start})$, or $\texttt{after}(p_{start} \dashrightarrow q_{start}) \wedge \texttt{after}(p_{end} \dashrightarrow q_{end})$. By exposing the bounding primitive subtasks, *ServLog* enables many kinds of constraints that cannot be specified on composite tasks directly. For instance, $\texttt{before}(p_{start} \dashleftarrow q_{end})$ or $\texttt{between}(p_{start} \dashrightarrow q_{start} \dashleftarrow p_{end})$.

Definition 4.14 (Independence Assumption) Two primitive update tasks are said to be *independent* if and only if they are represented by *disjoint* binary relations over database states.

A service process *satisfies the independence assumption* if and only if all its primitive update tasks are independent of each other. □

Independence implies that any transition between a pair of states is caused by precisely one primitive update task, and no other task can cause that transition.

Any set of primitive update tasks can be instrumented so that the tasks would become independent. For example, each primitive update task, t, can be made to insert a unique token every time it executes. Specifically, t might insert $token(t, 0)$ on the first execution and then $token(t, 1)$, $token(t, 2)$, etc., on subsequent executions. As a result, any transition between any pair of states would be possible by at most one primitive update task.

Without the independence assumption it is hard to come up with an effective algorithm for checking satisfaction of constraints by service executions, and it is hard to develop a simple enough proof theory for finding service executions that satisfy such constraints. To see this, suppose that the independence assumption is not satisfied and there are two distinct primitive update tasks such that $s \xrightarrow{p} s'$ and $s \xrightarrow{q} s'$ hold. Suppose that we are now trying to execute p at state s. In the presence of constraints such as $\texttt{before}(r \dashleftarrow q)$, $\texttt{absence}(q)$, and the like, it would be hard to determine whether p can be executed, since one must first determine if execution of p amounts to execution of another, prohibited task, such as q, in this example.

5. Reasoning about Contracts in *ServLog*

We begin with an example that shows how service contracting and contract execution are intended to work. The example illustrates most of the aspects of service modeling introduced in Section 4: service processes (control and data flows), client contract requirements, and service policies. The last two are represented via constraints. Section 5.2 formalizes the decision procedure as an extension to the proof procedure of the original CTR.

5.1. Informal Example

For simplicity, the example uses propositional tasks only, but the proof procedure in Section 5.2 is designed to work for the more general case where tasks have arguments.

For concreteness and to illustrate the interactive aspect of our model, we assume the following division among the tasks involved:

- Service tasks: *a, f, g*
- Client tasks: *d, e, h*

Service process:

Process formula:

$$a \otimes (B \mid C) \otimes (g \vee h)$$

Rules: (8)

$$\begin{array}{lcl} B & \leftarrow & d \vee e \\ C & \leftarrow & (f \otimes C) \vee \texttt{state} \end{array}$$

Client contract requirements:

$$\texttt{absence}(e) \wedge \texttt{atleast}_2(f) \quad (9)$$

Service policy:

$$\texttt{after}(d \dashrightarrow f) \wedge \texttt{absence}(g) \quad (10)$$

Service contracting. As explained at the end of Section 2, service contracting is an interactive decision procedure that checks whether an execution of the service process exists and satisfies both the service policies and the client contract requirements. In our example this amounts to finding an execution path of (8) such that the constraints (9) and (10) are satisfied. For example, *{a,d,f,f,f,h}* is an execution path for (8) path that satisfies the constraints, but *{a,f,f,d,h}* is an execution that violates $\texttt{after}(d \dashrightarrow f)$. The proof procedure introduced in Section 5.2 is designed to find paths on which the constraints are satisfied, if such paths exist. Note it does not matter if the path that was found will actually be the one to be executed—all that is needed is to find out if the contract is satisfiable. If service contracting does not find a path that satisfies the constraints, it means that no execution will ever be successful and the contract is unsatisfiable.

Service contract execution. If a contract is satisfiable, its execution deals with the actual interactions performed by the client and the service. The idea is based on the same proof theory that is used for service contracting, but it is applied differently. When a task is chosen for execution by the client or the service, the proof theory of Section 5.2 checks if acceptance of that task still leaves the possibility of a successful execution of the remainder of the service process (that satisfies the constraints). If so, the task is accepted and executed; otherwise the task is rejected and a different task must be chosen for execution by the agent in question. Note that such a task must exist because when accepting the previous task we must have checked that some continuation is possible.

Contract execution for our example works as follows:

1. Suppose the service selects task *a*. (This is the only task that can possibly be chosen according to our process specification.) We already checked (while doing service contracting) that there is a legal execution that starts with *a*, so the task is accepted.
2. The remainder of the process is $((d \vee e) \mid ((f \otimes C) \vee \texttt{state})) \otimes (g \vee h)$ (where *B* and *C* are replaced with their definitions). The next step can be taken either by the client (e.g., by picking *d* or *e*) or by service (e.g., by picking *f*). It is also possible for the system itself to execute an internal action by picking $\texttt{state}$ (here the client and the service "take a short break"). Let us assume that the client takes initiative and picks e for execution. The system checks if e can be executed and finds that this would violate the constraint $\texttt{absence}(e)$; so e is rejected. Suppose that the client does not give up and selects *d* for execution. Again, the system checks if d is allowed to execute given the constraints. In this case no constraint forbids the execution of d because a, d, f, f, f, h is a valid execution of the remainder of the process, so *d* is accepted.
   It would be very inefficient if the system had to go back to the beginning of the path in order to check if acceptance of an action permits a valid continuation. To avoid this, we modify the constraints after acceptance of each action so that we will never have to look back in order to decide whether to accept or reject an action. For instance, after accepting *d* the system revises the constraints by replacing $\texttt{after}(d \dashrightarrow f)$ with $\texttt{atleast}_1(f)$. This is possible because after the execution of *d* the constraint $\texttt{after}(d \dashrightarrow f)$ will be satisfied iff f is executed at some point in the future, whence $\texttt{atleast}_1(f)$. The system now checks whether an execution of the remaining process $((f \otimes C) \vee \texttt{state}) \otimes (g \vee h)$ exists under the updated set of constraints $\texttt{absence}(e) \wedge \texttt{atleast}_2(f) \wedge \texttt{atleast}_1(f) \wedge \texttt{absence}(g)$ (since we have two constraints $\texttt{atleast}_2(f)$ and $\texttt{atleast}_1(f)$, the last one can be removed).
3. For the remainder of the process, $((f \otimes C) \vee \texttt{state}) \otimes (g \vee h)$, only f and $\texttt{state}$ can be chosen. Let us assume that the internal action $\texttt{state}$ is picked for execution. The system now checks whether an execution of the remaining part of the process, $(g \vee h)$, exists given the updated set of constraints $\texttt{absence}(e) \wedge \texttt{atleast}_2(f) \wedge \texttt{absence}(g)$. It is easy to see that $(g \vee h)$ cannot execute to satisfy $\texttt{atleast}_2(f)$ because there is no f in $(g \vee h)$). Therefore, $\texttt{state}$ is rejected. The only other way to proceed is for the service to pick *f* (recall that *f* can be executed only by the service). Proceeding as before, the proof theory would check if f can execute. There are no constraints to prevent that, so the system updates the constraints and checks if a legal continuation is possible. The update replaces $\texttt{atleast}_2(f)$ with $\texttt{atleast}_1(f)$, since executing f means that the number of required occurrences of f decreases by 1. The remaining part of the process, $C \otimes (g \vee h)$ has a legal execution *{f,h}* with respect to the updated set of constraints $\texttt{absence}(e) \wedge \texttt{atleast}_1(f) \wedge \texttt{absence}(g)$, so *f* is accepted.

4. To proceed, we need to expand C using the rule in (8), which gives us $((f \otimes C) \vee \mathtt{state}) \otimes (g \vee h)$ — the same process as in the previous step. Although the set of constraints has now changed, because of $\mathtt{atleast}_1(f)$ the task f must still be executed for the same reasons as in the previous step. This task is accepted because a legal continuation exists, and now the set of constraints gets changed to $\mathtt{absence}(e) \wedge \mathtt{absence}(g)$.
5. Now the remainder of the process is $(C \vee \mathtt{state}) \otimes (g \vee h)$ and $\mathtt{state}$ can be successfully picked for execution and the remainder of the process becomes $g \vee h$. Either g can now be attempted by the client or h by the service. However, the constraint $\mathtt{absence}(g)$ prevents the former, so the service proceeds by picking h and, since no constraint prevents it from going ahead, it is executed. At this point, the remainder of the process is empty and we are done.

Note that other executions are also possible and could have been taken. For instance, if the service continued to press initiative in step 2, it could have picked f and the execution could have become *a,f,d,f,f,h* or *a,f,f,d,f,h*.

As we saw above, both service contracting and contract execution rely on the same inference rules which do not explicitly differentiate between client and service tasks, however the difference plays out in the way the inference rules are applied.

5.2. Proof System

Let $\mathcal{C} \in \mathcal{CONSTR}$ be a constraint (which can be composite), where $\mathcal{CONSTR}$ includes both the service policy and the client contract requirements. Let G be a service process and G and $\mathcal{C}$ satisfy the service contracts assumption. We consider the following reasoning problems in *ServLog*:

1. **Contracting**: The problem of determining if contracting for a service is possible amounts to finding out if there is an execution of the CTR formula $G \wedge \mathcal{C}$. Formally, contracting aims to determine if there is a path on which $G \wedge \mathcal{C}$ is true in every multi-path structure that makes all composite task definitions true.
2. **Contract Execution**: The problem of contract execution amounts to producing an *interactive* proof that $G \wedge \mathcal{C}$ can execute along some path. In that proof, the client and the service take turns that are prescribed by the process specification and by the ownership of the primitive tasks, as illustrated in the previous subsection. This proof must be *constructive*—a sequence of applications of the inference rules of CTR, which starts with an axiom and ends with the aforesaid formula $G \wedge \mathcal{C}$. Each such proof provides a way to execute the process without violating any of the constraints in $\mathcal{C}$.

The rest of this section develops a proof theory for formulas of the form $G \wedge\ \mathcal{C}$, where G is a service process and $\mathcal{C} \in \mathcal{CONSTR}$.

To simplify matters, we will assume that the service process G has no disjunctions in the rule bodies and in its CTR goal part. This does not limit the generality, as such disjunctions can always be eliminated through a simple transformation similar to the one in classical logic. For instance, the disjunction in

$$p \leftarrow q \otimes (r \vee s) \otimes t$$

can be eliminated by transforming this rule into

$$\begin{aligned} &p \leftarrow q \otimes newpred \otimes t \\ &newpred \leftarrow r \\ &newpred \leftarrow s \end{aligned}$$

Hot components. We recall the notion of *hot components* of a CTR goal from [11]: $hot(\psi)$ is a set of subformulas of ψ that are "ready to be executed" and corresponds to the notion of goal selection in SLD-style resolution proof theories. Hot components are defined inductively as follows:

1. hot(()) = { }, where () is the empty goal
2. hot(ψ) = $\{\psi\}$, if ψ is an atomic formula
3. hot($\psi_1 \otimes ... \otimes \psi_n$) = hot($\psi_1$)

4. hot(ψ_1 | ... | ψ_n) = hot(ψ_1) ∪...∪ hot(ψ_n)
5. hot($\odot\psi$) ={$\odot\psi$}

Additional constraints. The set of constraints is changing as tasks in the process execute. The exact mechanism is explained in the inference rule "*executing primitive update tasks*," below. It involves three new constraints: force($p(\bar{t})$), suspend($p(\bar{t})$), and next_right_before($q(\overline{u}') \rightsquigarrow p(\bar{t}) \leftarrow q(\overline{u})$), plus a generalization of the constraint before, which allows exceptions. We did not introduce these before, since the new constraints are technical means by which the proof theory works and they are unlikely to be employed by users directly.

The meaning of the constraint force($p(\bar{t})$), where $p(\bar{t})$ is a task pattern for a primitive update, is that the very next task to be executed must match $p(\bar{t})$. More precisely, an execution $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\rangle$ satisfies force($p(\bar{t})$) if $\mathbf{s}_1 \xrightarrow{p(\overline{T})} \mathbf{s}_2$ holds and $p(\overline{T})$ matches $p(\bar{t})$.

The constraint suspend($p(\bar{t})$) means that no task that matches $p(\bar{t})$ can execute at the current state. Formally, an execution $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\rangle$, such that $\mathbf{s}_1 \xrightarrow{r(\overline{V})} \mathbf{s}_2$ holds for some task atom $r(\overline{V})$, satisfies suspend($p(\bar{t})$) if $r(\overline{V})$ does *not* match $p(\bar{t})$.

The constraint next_right_before($q(\overline{u}') \rightsquigarrow p(\bar{t}) \leftarrow q(\overline{u})$) says that $q(\overline{u}')$ must be immediately preceded by $p(\bar{t})$ unless it is the first task to be executed. That first task can be a q-task, if it matches $q(\overline{u}')$. Formally, an execution $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\rangle$ satisfies this constraint if and only if either $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n\rangle$ satisfies right_before($p(\bar{t}) \leftarrow q(\overline{u})$) or $\mathbf{s}_1 \xrightarrow{q(\overline{V})} \mathbf{s}_2$ holds, where $q(\overline{V})$ matches $q(\overline{u}')$, and $\langle \mathbf{s}_2, ..., \mathbf{s}_n\rangle$ satisfies right_before($p(\bar{t}) \leftarrow q(\overline{u})$).

The generalization of before has the following syntax:

$$\texttt{before}(p(\bar{t}) \texttt{ <- } q(\overline{u}) \setminus \{q(\overline{u_1}), ..., q(\overline{u_n})\}) \tag{11}$$

where $p(\bar{t})$, $q(\overline{u})$, and $q(\overline{u_1})$, ..., $q(\overline{u_n})$, $n \geq 0$ ($n = 0$ means that the sets of exceptions are empty), are task patterns. The use of "$\setminus$" here indicates that before($p(\bar{t})$ <- $q(\overline{u})$) must hold, except for the tasks that match one of the exceptions $q(\overline{u_1})$,..., $q(\overline{u_n})$.

Formally, an execution $\langle \mathbf{s}_1, ..., \mathbf{s}_m\rangle$ satisfies the constraint (11) if and only if whenever there is a state $\mathbf{s}_i$ in this execution such that $\mathbf{s}_i \xrightarrow{q(\overline{U})} \mathbf{s}_{i+1}$ and $q(\overline{U})$ matches $q(\overline{u})$ but *none* of the $q(\overline{u_i})$'s then there is j, $j < i$, such that $\mathbf{s}_j \xrightarrow{p(\overline{T})} \mathbf{s}_{j+1}$ holds and $p(\overline{T})$ matches refine($p(\bar{t})$;$q(\overline{U})$,$q(\overline{u})$).

Note that, when the set of exceptions is empty, the constraint (11) reduces to the old form of the before-constraint. Therefore, to simplify the language, in the rest of this section we will be using only the generalized form of this constraint.

Substitutions. As usual in logic proof theories, we will rely on the notion of *substitution*, which is a mapping from variables to terms. If σ is a substitution and ψ is a service process or a term then we write $\psi\sigma$ for the result of applying the substitution σ to ψ. We call $\psi\sigma$ an *instance* of ψ. If $\psi\sigma$ has no variables left, we say that $\psi\sigma$ is a *ground instance* and that σ is a *ground substitution*.

Sequents. Let **P** be a set of composite task definitions. The proof theory manipulates expressions of the form **P**, **s**--- $\vdash$ $(\exists)\, \psi \wedge \mathcal{C}$, called *sequents*, where **P** is a set of task definitions, **s** is an identifier for the underlying database state, ψ is a CTR goal, and $\mathcal{C}$ is a (possibly composite) constraint, which may include the constraints in $\mathcal{C}\mathcal{ONSTR}$ as well as the new constraints (force, suspend, etc.) introduced just above. Informally, a sequent is a statement that $(\exists)\, \psi$, which is defined by the rules in **P**, can execute along some path that starts at state **s** so that all the constraints in $\mathcal{C}$ will be satisfied. Each inference rule has two sequents, one above the other. This is interpreted as: *if the upper sequent is inferred, then the lower sequent should also be inferred.* As in classical logic, any instance of an answer-substitution is a valid answer to a query.

The inference system presented here extends the inference system for Horn CTR [11] with two additional inference rules (Rules 2 and 3). Other rules from [11] (e.g., Rule 6) are also significantly modified. The new system reduces to the old one when the constraint $\mathcal{C}$ is a CTR tautology (path). The new system also extends and simplifies the proof theory developed in [38].

All rules and the axioms operate with constraints, which get modified as a result of the rule application. However, some of the rules require the constraint to be a conjunction of the existence, serial, and the additional constraints introduced in this section. We call such constraints *conjunctive*. A conjunctive constraint can be viewed as a set, so we will often write $c \in \mathcal{C}$ meaning that c is a conjunct in $\mathcal{C}$.

The notion of a proof. A *proof* of a sequent *seq* is a series of sequents, $seq_0, seq_1, \ldots, seq_{n-1}, seq_n$, where $seq_n = seq$ and each seq_i is either an axiom-sequent (below) or is derived from earlier sequents by one of the inference rules below.

Axiom. All axioms have the form $\mathbf{P}, \mathbf{s}$ --- $\vdash () \wedge \mathcal{C}$, where **s** is a database state identifier and $\mathcal{C}$ is a conjunctive constraint that does not contain constraints of the form $\texttt{force}(\overline{r})$, $\texttt{atleast}_k(\overline{r})$, and $\texttt{exactly}_k(\overline{r})$, where $k \geq 1$.

Inference Rules. In Rules 1-7 below, σ denotes a substitution, ψ and ψ' are service processes, $\mathcal{C}$ and $\mathcal{C}'$ are constraints, **s**, $\mathbf{s}_1$, $\mathbf{s}_2$ are database state identifiers, and p is a task.

1. *Eliminating disjunctive constraints*: Let ψ be a CTR goal and $\mathcal{C}'$ a disjunct in the disjunctive normal form of $\mathcal{C}$ (i.e., $\mathcal{C}'$ is a conjunctive constraint). Then

$$\frac{\mathbf{P}, \mathbf{s} \text{ ---}\vdash \psi \wedge \mathcal{C}'}{\mathbf{P}, \mathbf{s} \text{ ---}\vdash \psi \wedge \mathcal{C}}$$

Note that $\mathcal{C}'$ is a conjunction of existence and serial constraints.

2. *Solving builtin tests*: Let χ be a conjunction of builtin test tasks. Suppose there is a ground substitution σ such that $\chi\sigma$ evaluates to true. Then

$$\frac{\mathbf{P}, \mathbf{s} \text{ ---}\vdash () \wedge \mathcal{C}}{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ \chi \wedge \mathcal{C}}$$

3. *Commutativity with respect to builtin tests*: Let χ be a conjunction of builtin test tasks and ψ a CTR goal. Then

$$\frac{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ (\psi \otimes \chi) \wedge \mathcal{C}}{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ (\chi \otimes \psi) \wedge \mathcal{C}}$$

4. *Applying composite task definitions*: Let $r \leftarrow \beta$ be a rule in **P**, and assume that its variables have been renamed so that none are shared with ψ. If p and r unify with the most general unifier σ then

$$\frac{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ \psi'\sigma \wedge \mathcal{C}}{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ \psi \wedge \mathcal{C}}$$

where ψ' is obtained from ψ by replacing a hot occurrence of p with β.

5. *Executing query tasks*: Suppose that $p\sigma$ and $\psi'\sigma$ share no variables and either (i) p is a primitive query task such that $(\exists)p\sigma$ is true in the state **s**; or (ii) $p = \texttt{state}$ and σ is the identity substitution. Then

$$\frac{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ \psi'\sigma \wedge \mathcal{C}}{\mathbf{P}, \mathbf{s} \text{ ---}\vdash (\exists)\ \psi \wedge \mathcal{C}}$$

where ψ' is obtained from ψ by deleting a hot occurrence of p.

6. *Executing primitive update tasks*: Let $p\sigma$ be a primitive update task such that $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$, $p \in hot(\psi)$, and $\mathcal{C}$ has no constraint of either of the forms below. In the description below, we assume that $\bar{p}$ is a task pattern such that $p\sigma$ matches $\bar{p}$ and that $\bar{r}$ denotes an arbitrary task pattern:

   – $\texttt{absence}(\bar{p})$

- $\texttt{suspend}(\bar{p})$
- $\texttt{force}(\bar{r})$ such that $p\sigma$ does not match $\bar{r}$
- $\texttt{before}(\bar{r} \dashleftarrow \bar{p} \setminus \{\bar{p}_1, ..., \bar{p}_k\})$ $k \geq 0$ and $p\sigma$ matches neither $\bar{r}$ nor any of the $\bar{p}_i$'s
- $\texttt{right\_before}(\bar{r} \leftarrow \bar{p})$
- $\texttt{next\_right\_before}(\bar{p}' \not\leftsquigarrow \bar{r} \leftarrow \bar{p})$, where $p\sigma$ does not match $\bar{p}'$.

Then the inference rule has the following form:

$$\frac{\mathbf{P}, \mathbf{s}_2 \dashrightarrow\vdash (\exists)\ \psi'\sigma \wedge \mathcal{C}'}{\mathbf{P}, \mathbf{s}_1 \dashrightarrow\vdash (\exists)\ \psi \wedge \mathcal{C}}$$

where $\mathcal{C}$ is a conjunctive constraint, ψ' is obtained from ψ by deleting the hot component p, and $\mathcal{C}'$ is constructed out of $\mathcal{C}$ as follows.
Initially, $\mathcal{C}'$ is empty (a tautology, $\texttt{path}$). Then constraints are added to it (as conjuncts) according to the cases below. (Again, in all these cases, we assume that $p\sigma$ matches $\bar{p}$ and $\bar{r}$, $\bar{s}$ are arbitrary task patterns.)

(a) If $\texttt{atleast}_n(\bar{p}) \in \mathcal{C}$, where $n > 1$, add the following to $\mathcal{C}'$:

- $\texttt{atleast}_{n-1}(\bar{p})$;

(b) If $\texttt{exactly}_n(\bar{p}) \in \mathcal{C}$, where $n > 1$, add the following to $\mathcal{C}'$:

- $\texttt{exactly}_{n-1}(\bar{p})$;

(c) If $\texttt{exactly}_1(\bar{p}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{absence}(\bar{p})$;

(d) If $\texttt{after}(\bar{p} \dashrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{after}(\bar{p} \dashrightarrow \bar{r})$
- $\texttt{atleast}_1(\bar{r}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$

(e) If $\texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r})$
- $\texttt{absence}(\bar{r}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$

(f) If $\texttt{right\_after}(\bar{p} \rightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{right\_after}(\bar{p} \rightarrow \bar{r})$
- $\texttt{force}(\bar{r}')$, where $\bar{r}' = refine(\bar{r}; p\sigma, \bar{p})$.

(g) If $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})$ where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$;

(h) If $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$
- $\texttt{suspend}(\bar{r}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$

(i) If $\texttt{right\_before}(\bar{p} \leftarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{next\_right\_before}(\bar{r}' \not\leftsquigarrow \bar{p} \leftarrow \bar{r})$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$;

(j) If $\texttt{next\_right\_before}(\bar{r}' \not\leftsquigarrow \bar{s} \leftarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{right\_before}(\bar{s} \leftarrow \bar{r})$, where $\bar{s}, \bar{r}$ are arbitrary task patterns;

(k) If $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$

  - `before`$(\bar{r}'$ `<-` $\bar{s}')$, where $\bar{r}' =$ `refine`$(\bar{r}; p\sigma, \bar{p})$ and $\bar{s}' =$ `refine`$(\bar{s}; p\sigma, \bar{p})$

(l) If `not_between`$(\bar{p}$ `-/>` $\bar{r}$ `</-` $\bar{s}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

  - `not_between`$(\bar{p}$ `-/>` $\bar{r}$ `</-` $\bar{s})$
  - `blocks`$(\bar{r}'$ `-/>` $\bar{s}')$ where $\bar{r}' =$ `refine`$(\bar{r}; p\sigma, \bar{p})$ and $\bar{s}' =$ `refine`$(\bar{s}; p\sigma, \bar{p})$

(m) For all other constraints in $\mathcal{C}$, copy them over to $\mathcal{C}'$, but leave out the constraints of the form:

  - `atleast`$_1(\bar{p})$
  - `force`$(\bar{p})$
  - `suspend`$(\bar{r})$, for any task pattern $\bar{r}$

7. *Executing atomic tasks*: If $\odot\alpha$ is a hot component in ψ then

$$\frac{\mathbf{P}, \mathbf{s} \;\text{---}\vdash (\exists)\, (\alpha \otimes \psi') \wedge \mathcal{C}}{\mathbf{P}, \mathbf{s} \;\text{---}\vdash (\exists)\, \psi \wedge \mathcal{C}}$$

where ψ' is obtained from ψ by deleting a hot occurrence of $\odot\alpha$.

Theorem 1 *The above inference system is sound and complete for proving constrained service processes, if the service processes and the constraints satisfy the service contracts assumptions.*

Proof: Soundness of the inference system is proved in Appendix A and completeness in B. □

Example. The following example illustrates the inference procedure.

Goal G: $\exists ?x\ ((p(?x) \otimes q) \mid r(?x))$
Rules: $r(?x) \leftarrow (xyz(?y, ?z) \otimes s(?x, ?y, ?z) \otimes r(?y))$
$r(?x) \leftarrow$ `state`
Constraint $\mathcal{C}$: $($`atleast`$_2(s(\_,\_,\_)) \vee$ `before`$(p(\_x)$ `<-` $s(\_x, 1, \_z)))$

Here p, q, and s are assumed to be primitive tasks, and in the case of p and s they can be executed with any integer argument. Furthermore, in this example we assume that the execution of p, s, and q modifies the database, as follows: $p(1)$ adds $xyz(3,7)$; $s(1,3,7)$ adds $xyz(2,7)$; $s(3,2,7)$ adds $abc(1)$; and q deletes $xyz(3,7)$. These assumptions were needed in order to show that the constraint is satisfied in this example. If we did not make these assumptions, then the constraint might not be satisfied in which case the inference procedure would not infer the constraint. The goal G can be executed in several ways such that $\mathcal{C}$ is satisfied. We show one possibility, which corresponds to one derivation of the sequent **P**, **s** ---⊢ $G \wedge \mathcal{C}$, where **s** is an identifier for $\{\,\}$, the empty state. In this derivation, we use the top-down method, i.e., we start with the goal and apply the inference rules backwards. Each sequent is derived from the previous one by an inference rule. The deduction succeeds when the last sequent is an axiom. We start with the sequent

$$P, \{\} \;\text{---}\vdash (\exists ?x\ ((p(?x) \otimes q) \mid r(?x))) \\ \wedge (\texttt{atleast}_2(s(\_,\_,\_)) \vee \texttt{before}(p(\_x) \texttt{ <- } s(\_x, 1, \_)))$$

Here, instead of a state identifier (**s**) we put the corresponding database state (`{ }`) explicitly. To make the sequent easier to read, we will continue doing this in the rest of this example.

Hot components: $\{p(?x), r(?x)\}$. By inference rule 1 (eliminating disjunctive constraints) we obtain:

$$P, \{\} \;\text{---}\vdash (\exists ?x\ ((p(?x) \otimes q) \mid r(?x))) \\ \wedge\, \texttt{atleast}_2(s(\_,\_,\_))$$

Hot components: $\{p(?x), r(?x)\}$. By inference rule 4 (composite task definitions) we obtain:

$$P, \{\} \text{---} \vdash (\exists\, ?x, ?y, ?z\ (p(?x) \otimes q) \mid (xyz(?y, ?z) \otimes s(?x, ?y, ?z) \otimes r(?y))) \\ \wedge\, \mathtt{atleast}_2(s(\_, \_, \_))$$

Hot components: $\{p(?x), xyz(?y, ?z)\}$. By inference rule 6, choosing $p(?x)$ and executing this primitive task with the argument $?x = 1$, we obtain (recall that the execution of $p(1)$ adds the fact $xyz(3, 7)$ to the database):

$$P, \{xyz(3, 7)\} \text{---} \vdash (\exists\, ?y, ?z\ q \mid (xyz(?y, ?z) \otimes s(1, ?y, ?z) \otimes r(?y))) \\ \wedge\, \mathtt{atleast}_2(s(\_, \_, \_))$$

Hot components: $\{q, xyz(?y, ?z)\}$. By inference rule 5 (executing query tasks):

$$P, \{xyz(3, 7)\} \text{---} \vdash q \mid (s(1, 3, 7) \otimes r(3)) \\ \wedge\, \mathtt{atleast}_2(s(\_, \_, \_))$$

Hot components: $\{q, s(1, 3, 7)\}$. By inference rule 6 applied to the primitive task $s(1, 3, 7)$ and the earlier assumption, this execution adds the fact $xyz(2, 7)$ to the database:

$$P, \{xyz(3, 7), xyz(2, 7)\} \text{---} \vdash (q \mid r(3)) \wedge \mathtt{atleast}_1(s(\_, \_, \_))$$

Hot components: $\{q, r(3)\}$. By inference rule 6 applied to the primitive task q (which, as mentioned above, deletes $xyz(3, 7)$):

$$P, \{xyz(2, 7)\} \text{---} \vdash r(3) \wedge \mathtt{atleast}_1(s(\_, \_, \_))$$

Hot components: $\{r(3)\}$. By inference rule 4 (composite task definitions):

$$P, \{xyz(2, 7)\} \text{---} \vdash (\exists\, ?y, ?z\ (xyz(?y, ?z) \otimes s(3, ?y, ?z) \otimes r(?y))) \\ \wedge\, \mathtt{atleast}_1(s(\_, \_, \_))$$

Hot components: $\{xyz(?y, ?z)\}$. By inference rule 5 (executing query tasks) and choosing $xyz(?y, ?z)$:

$$P, \{xyz(2, 7)\} \text{---} \vdash (s(3, 2, 7) \otimes r(2)) \wedge \mathtt{atleast}_1(s(\_, \_, \_))$$

Hot components: $\{s(3, 2, 7)\}$. By inference rule 6 for executing the primitive task $s(3, 2, 7)$ and by the earlier assumption, it adds the fact $abc(1)$ to the database:

$$P, \{xyz(2, 7), abc(1)\} \text{---} \vdash r(2)$$

Hot components: $\{r(2)\}$. By inference rule 4 (composite task definitions), where we use the second rule for r:

$$P, \{xyz(2, 7), abc(1)\} \text{---} \vdash \mathtt{state}$$

Hot components: $\{\mathtt{state}\}$. By rule 5 applied to `state` we derive:

$$P, \{xyz(2, 7), abc(1)\} \text{---} \vdash ()$$

During the deduction, we executed the following sequence of primitive tasks:

{p(1),xyz(3,7),s(1,3,7),q,xyz(2,7),s(3,2,7),`state`}

It is easy to see that this sequence indeed satisfies the constraint $\mathcal{C}$, which required $s(\_,\_,\_)$ to be executed at least twice.

Decidability and Complexity. In general, query answering in CTR is semi-decidable [11], like in classical logic, so no effective procedure exists to answer all possible queries and terminate. Appendices A and B show that the same is true for *ServLog*. In [7], various subsets of CTR were investigated for their decidability and complexity properties. One important restriction studied there is called *full boundedness*. In terms of *ServLog*, this roughly means that every update task must diminish some bounded from below, discrete measure (e.g., a positive integer measure). In that case, the CTR proof procedure is data-complete for NP [7] (i.e., NP-complete when the rules are fixed but data is allowed to vary). Fortunately, all existing workflow modeling systems are implicitly based on the assumption that any useful workflow must be fully bounded (where the bounds can be defined for different processes). Typically this is manifested by imposing upper bounds on the number of iterations, the number of steps, and other similar restrictions. It is also easy to instrument *ServLog* service processes so that they become fully bounded. For instance, primitive update tasks could be forced to diminish a global discrete and bounded resource. More importantly, the recent trend in Logic Programming is to avoid restricting the expressive power of programs by curtailing the usefulness of logical frameworks with restrictions, such as full boundedness. Instead, new approaches aim to develop tools for detecting non-terminating behavior and help the programmer correct the problem [30,22]. It is our contention that this is a more productive direction for Transaction Logic-based approaches than the restriction-based approaches. We should also note that some decidability results developed for other approaches carry over to *ServLog*—see the discussion of artifact systems [15] in Section 6.

6. Related Work

ServLog builds on the service contracting framework that was partially developed in [39,38], greatly expanding and generalizing it, while at the same time simplifying the technical details. The major simplifications and extensions include:

- Removing the unique task occurrence restriction and obviating the need for complex simplification transformations.
- Tasks are no longer limited to propositional constants. This provides the ability to represent complex data flow among tasks as well as general transitional constraints.
- General iterative and even mutually recursive tasks in the specification of service processes.
- Generalization of the proof theory, which now deals with the many new additions to the language, and handles constrained (non-Horn) formulas, which previous CTR proof theory was not able to handle.
- Proofs of soundness and completeness for the generalized proof theory.

The present paper significantly extends this proof theory to formulas that contain the $\wedge$ connective thus enabling execution of *constrained transactions*, which are non-Horn. We also deal with a much larger class of constraints than [16,38], including iterative processes.

Declare [45] is a service flow language that is closely related to *ServLog*. It uses Linear Temporal Logic to formalize service flows and automata theory to enact service specifications. The relations between tasks are described entirely in terms of constraints. Apart from the obvious radical differences in the formalisms, some other important differences are worth noting. First, the constraint algebra $\mathcal{CONSTR}$ of *ServLog* is more expressive than the one used in Declare. Second, by combining constraints with service processes (conditional control flow and data flow), *ServLog* incorporates current practices in workflow modeling. Third, data flow and conditional control flow are easily available in *ServLog*, while they have not been developed in the context of Declare. Declare was also formalized using Abductive Logic Programming in [33]. While this formalization supports different verification tasks, the focus remains on modeling service flow exclusively in terms of constraints and does not deal with control and data flow.

In [49,48] the authors propose a combination of colored Petri nets, Declare, and DCR graphs as a way of modeling procedural processes with data support. This combination can be seen as either "adding declarative control-flow to CP-nets" or as "adding data-flow to declarative formalisms." From the modeling perspective, the approach requires the user to be familiar with three formalisms, as opposed to our framework where all aspects are represented within a single logical language. The authors do not provide a precise formalization of the combination of the three languages, so it is unclear how certain elements such as atomic transactions, hierarchical definitions of tasks, and constraints over complex tasks can be expressed in the combination of those three languages. From the analysis point of view, the authors address the problem of simulation: checking whether a transition is enabled in the CP-net model, and subsequently whether it is also allowed according to the declarative constraint. This is done using model checking. Our approach differs conceptually in that we rely on a logical proof theory as opposed to model checking.

In the same spirit of extending Declare with data elements, [32] extends the Declare notation by allowing activities to have associated multiple ports (denoting events associated to the activity lifecycle) and constraints to be attached to ports thereby allowing data-aware conditions. These extensions are then formalized in the Event Calculus (EC). In terms of modeling, the focus remains on modeling service flow exclusively in terms of constraints and this does not address the aforesaid limitations with respect to control and data flow (e.g., it is unclear how elements such as hierarchical definitions of tasks can be achieved with the proposed extensions). For reasoning, the paper defers to generic EC reasoners, which are significantly more complicated than those for CTR.

Another related approach to service modeling and verification is based on the *business artifact model* [28,18,15]. In this approach, tasks (which they call "services") are represented as transformations on objects, called *artifacts* and they have pre- and post-conditions. In addition, various constraints can be specified using Linear Temporal Logic (like Declare) where first-order statements are allowed in place of propositions. In *ServLog*, artifact-based systems correspond to a very special form of service processes of the following form:

$$\begin{array}{l}
serv(?X) \leftarrow termination\_condition(?X) \\
serv(?X) \leftarrow precond_1(?X) \otimes task_1(?X,?Y) \otimes postcond_1(?X,?Y) \otimes serv(?Y) \\
serv(?X) \leftarrow precond_2(?X) \otimes task_2(?X,?Y) \otimes postcond_2(?X,?Y) \otimes serv(?Y) \\
\ldots
\end{array}$$

The artifacts are represented here via variables, but they can also be represented as data items passed around via the underlying database. While the control structure of artifact systems is a small subset of what *ServLog* services can have, the constraints used in those systems form a superset of the constraints in *ServLog*: they can be arbitrary LTL formulas. (It is not clear, however, whether this generality makes a difference in practice.) It is interesting to note that the decidability results from [18] carry over from artifact systems to the special case of the *ServLog* service processes described above.

An emerging area related to our work is that of compliance checking between business processes and business contracts. For example, in [24,25] both processes and contracts are represented in a formal language called FCL. FCL is based on a formalism for the representation of contrary-to-duty obligations, i.e., obligations that arise when other obligations are violated as typically happens with penalties embedded in contracts. Based on this, the authors give a semantic definition for compliance, but no practical algorithms. In contrast, *ServLog* provides a proof theory for verifying feasibility of service contracting as well as for contract execution.

Several other approaches to service contracting and contract execution are relevant to our work [41,2,3,13], but not directly related. Most of these present logical modeling languages for contracts in various settings. Being based on normative deontic notions of obligation, prohibition, and permission, we believe that these works are complementary to ours and the approaches could be combined.

Other popular tools for process modeling are based on Petri nets, process algebras, and temporal logic. A related area of research is data-centric business and service modeling and verification, for which a representative approach is [47]. Approaches in this area primarily combine databases and model checking techniques for the purpose of automated verification. The advantage of CTR over these approaches is that it is a *unifying* formalism that integrates a number of process modeling paradigms ranging from conditional control flows to data flows to hierarchical modeling to constraints, and even to game-theoretic aspects of multiagent processes (see, for example, [17]). Moreover, as shown in [16], CTR modeling sometimes leads to algorithms with better known complexity for special cases than general model checking.

Finally, it is worth mentioning our work in the context of general AI techniques on reasoning about actions. A key differentiation of our approach based on CTR is that, while many works in AI focus on reasoning *about* actions, CTR also has constructs for *defining* transactions and *executing* them. These issue and further comparison are discussed in detail in the original and follow-up papers [8,9,11,36,5].

7. Conclusions

The main contribution of this paper is *ServLog*, a *unifying* logical framework for semantics-aware services addressing two core aspects of services: modeling and contracting. In particular, this work adds a few new building blocks to the theoretical foundations for service contracting. It offers an expressive set of modeling primitives and addresses a wide variety of issues in service contracts. These primitives range from complex process description (including iteration and conditional transitions between tasks) to temporal and data constraints. Despite its expressive power, *ServLog* still provides a reasoning procedure for automated service contracting.

In the context of established business process languages (e.g., BPMN, WS-BPEL), *ServLog* not only captures typical procedural constructs found in such languages, but also greatly extends them, enables declarative specification and reasoning, and opens the way for automatic generation of business processes from service contracts. Furthermore, in the context of Semantic Web Services, ServLog complements established approaches such as OWL-S and WSMO, which primary focused on semantic annotations for Web services, and brings new directions for research in Semantic Web Services related to service contracts.

The approach presented in this paper has been implemented for the propositional case (when tasks do not have parameters) in the GEECoP (Graphical Editor and Engine for Constrained Processes)[13] tool with promising results (further described in [44]).

Future work may include support for reasoning about quality of service (QoS) and non-functional properties in service contracts—see, e.g., [14] for an example of a framework for QoS-based Web service contracting, which could be combined with *ServLog*, and a more complete implementation of the reasoning framework based on recent results on efficient implementation of Transaction Logic [20]. Another challenging issue is generation of service contracts in *ServLog* from informal, often ambiguous, natural language contracts. It would be also interesting to do a methodological comparison of the expressive power of *ServLog* with workflow specification languages based on automata, pre/post conditions, and temporal constraints. In this respect, the recent framework introduced in [1] for comparing distinct workflow models by means of views might be relevant. Explicitly treating violations and reasoning about the effects of violations in contracts is also an interesting direction for further research in *ServLog*. Service discovery is complementary to the problem of service contracting addressed by *ServLog*. Some approaches to semantic service discovery are based on Transaction Logic (e.g. [29]) and could be combined with *ServLog*. Such a combination would be yet another intriguing continuation for the present work. *ServLog*, as a logical framework for specification of and reasoning about service contracts, is orthogonal to implementation issues such as centralized or distributed provisioning of services. *ServLog* can be instantiated in different settings; for example, when different tasks are provisioned in a distributed or decentralized environment. Studying the implications of instantiating *ServLog* in different environments is interesting from a practical perspective. We are also investigating applicability of *ServLog* and CTR to other areas such as *smart contracts*[14] in the context of cryptocurrencies.[15] Here, *ServLog* could prove to be a useful approach for modeling and reasoning about distributed cryptocurrency contracts.[16]

Acknowledgements. We thank the anonymous referees for their valuable comments, which helped to improve this paper. Michael Kifer was partially supported by the NSF grant 0964196.

[13] http://sourceforge.net/projects/gecop/
[14] http://szabo.best.vwh.net/smart_contracts_idea.html
[15] See for example Bitcoin contracts at https://en.bitcoin.it/wiki/Contracts.
[16] For a relevant related idea, see [4], where the authors proposed timed automata for formalizing Bitcoin contracts.

Appendix

A. Soundness of the Inference System

Theorem 2 (Soundness of the Inference System). *Suppose* $\mathbf{P}$ *is a set of composite task definitions,* $\boldsymbol{s}$ *a database identifier,* ψ *a service process,* $\mathcal{C}$ *a conjunction of primitive or serial constraints. We also assume that* ψ *and* $\mathcal{C}$ *satisfy the service contracts assumption. Then:*

$$If \quad \mathbf{P}, \boldsymbol{s} \text{ --- } \vdash \psi \land \mathcal{C} \quad then \quad \mathbf{P}, \boldsymbol{s} \text{ --- } \models \psi \land \mathcal{C}$$

Proof: It suffices to prove that the axiom and each inference rule are sound. The only nontrivial case here is soundness of the inference rule 6, *execution of primitive update tasks*. This proof is given in Proposition 1. □

Proposition 1 (Soundness of the $executing\ primitive\ tasks$ inference rule)

Let $p\sigma$ *be a primitive update task such that* $\boldsymbol{s}_1 \xrightarrow{p\sigma} \boldsymbol{s}_2$ *holds,* $p \in hot(\psi)$*, and* $\mathcal{C}$ *has no constraint of either of the forms below. We assume that* $\bar{p}$ *is a task pattern such that* $p\sigma$ *matches* $\bar{p}$ *and that* $\bar{r}$ *denotes an arbitrary task pattern:*

- `absence`$(\bar{p})$
- `suspend`$(\bar{p})$
- `force`$(\bar{r})$ *such that* $p\sigma$ *does not match* $\bar{r}$
- `before`$(\bar{r} \dashleftarrow \bar{p} \setminus \{\bar{p}_1, ..., \bar{p}_k\})$ $k \geq 0$ *and* $p\sigma$ *matches neither* $\bar{r}$ *nor any of the* $\bar{p}_i$*'s*
- `right_before`$(\bar{r} \leftarrow \bar{p})$
- `next_right_before`$(\bar{p}' \leftsquigarrow \bar{r} \leftarrow \bar{p})$, *where* $p\sigma$ *does not match* $\bar{p}'$.

Then the inference rule has the following form:

$$\frac{\boldsymbol{P}, \boldsymbol{s}_2 \text{ --- } \vdash (\exists)\ \psi'\sigma \land \mathcal{C}'}{\boldsymbol{P}, \boldsymbol{s}_1 \text{ --- } \vdash (\exists)\ \psi \land \mathcal{C}}$$

where ψ' *is obtained from* ψ *by deleting the hot component* p *and* $\mathcal{C}'$ *is constructed out of* $\mathcal{C}$ *as follows.*

Initially, $\mathcal{C}'$ *is empty (a tautology,* `path`*). Then constraints are added (as conjuncts) according to the cases below. As before, in all these cases we assume that* $p\sigma$ *matches* $\bar{p}$ *and* $\bar{r}$, $\bar{s}$ *represent arbitrary task patterns.*

(a) If `atleast`$_n(\bar{p}) \in \mathcal{C}$, where $n > 1$, add the following to $\mathcal{C}'$:

- `atleast`$_{n-1}(\bar{p})$;

(b) If `exactly`$_n(\bar{p}) \in \mathcal{C}$, where $n > 1$, add the following to $\mathcal{C}'$:

- `exactly`$_{n-1}(\bar{p})$;

(c) If `exactly`$_1(\bar{p}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- `absence`$(\bar{p})$;

(d) If `after`$(\bar{p} \dashrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- `after`$(\bar{p} \dashrightarrow \bar{r})$
- `atleast`$_1(\bar{r}')$, where $\bar{r}' =$ `refine`$(\bar{r}; p\sigma, \bar{p})$

(e) If `blocks`$(\bar{p} \not\dashrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- `blocks`$(\bar{p} \not\dashrightarrow \bar{r})$
- `absence`$(\bar{r}')$, where $\bar{r}' =$ `refine`$(\bar{r}; p\sigma, \bar{p})$

(f) If `right_after`$(\bar{p} \rightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{right\_after}(\bar{p} \rightarrow \bar{r})$
- $\texttt{force}(\bar{r}')$, where $\bar{r}' = refine(\bar{r}; p\sigma, \bar{p})$.

(g) If $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})$ where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$;

(h) If $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$
- $\texttt{suspend}(\bar{r}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$

(i) If $\texttt{right\_before}(\bar{p} \leftarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{p} \leftarrow \bar{r})$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$;

(j) If $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{s} \leftarrow \bar{r}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{right\_before}(\bar{s} \leftarrow \bar{r})$, where $\bar{s}, \bar{r}$ are arbitrary task patterns.

(k) If $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$
- $\texttt{before}(\bar{r}' \dashleftarrow \bar{s}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$ and $\bar{s}' = \texttt{refine}(\bar{s}; p\sigma, \bar{p})$

(l) If $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s}) \in \mathcal{C}$, add the following to $\mathcal{C}'$:

- $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s})$
- $\texttt{blocks}(\bar{r}' \not\dashrightarrow \bar{s}')$ where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$ and $\bar{s}' = \texttt{refine}(\bar{s}; p\sigma, \bar{p})$

(m) For all other constraints in $\mathcal{C}$, copy them over to $\mathcal{C}'$, but leave out the constraints of the form:

- $\texttt{atleast}_1(\bar{p})$
- $\texttt{force}(\bar{p})$
- $\texttt{suspend}(\bar{r})$, for any task pattern $\bar{r}$

Proof: Suppose $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \models (\exists)\ \psi'\sigma \wedge \mathcal{C}'$. By soundness of the CTR proof theory, it follows that $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \models (\exists)\ \psi$. So, it remains to prove that $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \models (\exists)\ \psi \wedge \mathcal{C}$ and, as a special case, that:

$$\mathbf{P}, \langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle \models \mathcal{C} \tag{12}$$

In the proof, we will rely on the premise

$$\mathbf{P}, \langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle \models \mathcal{C}' \tag{13}$$

which is a special case of the assumption $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \models (\exists)\ \psi'\sigma \wedge \mathcal{C}'$.

Proof of (12) To prove (12), we need to consider each case in the inference rule 6 and, for each constraint that is not copied directly from $\mathcal{C}$ to $\mathcal{C}'$ via Case (m), we need to show that it holds over the execution $\langle \mathbf{s}_1 ... \mathbf{s}_n \rangle$.

Case (a): In this case, $\mathcal{C}$ has $\texttt{atleast}_n(\bar{p})$ and $\mathcal{C}'$ has $\texttt{atleast}_{n-1}(\bar{p})$. By (13), it follows that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \models \texttt{atleast}_{n-1}(\bar{p})$.

If $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \models \texttt{atleast}_{n-1}(\bar{p})$ then since $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ and $p\sigma$ matches $\bar{p}$, it follows that $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \models \texttt{atleast}_n(\bar{p})$ holds.

Case (b): similar to Case (a).

Case (c): In this instance, $\mathcal{C}'$ contains $\texttt{absence}(\bar{p})$, so $\mathbf{s}_i \xrightarrow{r} \mathbf{s}_{i+1}$, $1 < i < n$, is not possible for any r that matches $\bar{p}$. Therefore, $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{exactly}_1(\bar{p})$, i.e., $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \models \texttt{exactly}_1(p)$ holds.

Case (d): In this case, $\mathcal{C}'$ contains $\texttt{after}(\bar{p} \dashrightarrow \bar{r})$ and $\texttt{atleast}_1(\bar{r}')$, and $\mathcal{C}$ contains $\texttt{after}(\bar{p} \dashrightarrow \bar{r})$. By (13), it follows that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \models \texttt{atleast}_1(\bar{r}')$. Suppose $\texttt{after}(\bar{p} \dashrightarrow \bar{r})$ does not hold on $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. Since the transition from $\mathbf{s}_1$ to $\mathbf{s}_2$ can be made only by $p\sigma$ (by the primitive update task independence assumption), it follows that $\texttt{absence}(\bar{r}')$ must hold on $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. But this contradicts the fact that $\texttt{atleast}_1(\bar{r}')$ must hold on the same path. Thus, $\texttt{after}(\bar{p} \dashrightarrow \bar{r})$ must hold on $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$, i.e., $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \vDash \texttt{after}(\bar{p} \dashrightarrow \bar{r})$ holds.

Case (e): Here $\mathcal{C}'$ contains $\texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r})$ and $\texttt{absence}(\bar{r}')$, and $\mathcal{C}$ contains $\texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r})$. By (13), $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{absence}(\bar{r}')$. Suppose $\texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r})$ does not hold on $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. Since the transition from $\mathbf{s}_1$ to $\mathbf{s}_2$ can be made only by $p\sigma$ (by the primitive update task independence assumption), it follows that $\texttt{atleast}_1(\bar{r}')$ must hold on $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. But this contradicts $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{absence}(\bar{r}')$. Thus $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \vDash \texttt{blocks}(\bar{p} \not\dashrightarrow \bar{r})$ must hold.

Case (f): Here $\mathcal{C}'$ contains both $\texttt{right\_after}(\bar{p} \rightarrow \bar{r})$ and $\texttt{force}(\bar{r}')$, while $\mathcal{C}$ contains $\texttt{right\_after}(\bar{p} \rightarrow \bar{r})$. By (13), it follows that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{force}(\bar{r}')$, and thus $\mathbf{s}_2 \xrightarrow{r'} \mathbf{s}_3$ must hold for some r' that matches $\bar{r}'$, a refinement pattern of $\bar{r}$. Since $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ and $p\sigma$ matches $\bar{p}$, it follows that $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \vDash \texttt{right\_after}(\bar{p} \rightarrow \bar{r})$ holds.

Case (g): In this instance, $\mathcal{C}'$ contains $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})$ while $\mathcal{C}$ contains $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\})$. By (13), it follows that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})$. Since $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ holds, $p\sigma$ matches $\bar{p}$, and $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$, it follows that $\texttt{before}(p\sigma \dashleftarrow \bar{r}')$ (and, as a special case, $\texttt{before}(\bar{p} \dashleftarrow \bar{r}')$) is satisfied by $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. Since $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})$, we conclude that it also satisfies $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\})$. But the transition from $\mathbf{s}_1$ to $\mathbf{s}_2$ was made by $p\sigma$, which matches $\bar{p}$, so $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ cannot violate $\texttt{before}(\bar{p} \dashleftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\})$.

Case (h): In this case, $\mathcal{C}'$ contains $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$ and $\texttt{suspend}(\bar{r}')$, and $\mathcal{C}$ contains $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$. By (13), it follows that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{suspend}(\bar{r}') \wedge \texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$. Suppose $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$ does not hold on $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$. Since the transition from $\mathbf{s}_1$ to $\mathbf{s}_2$ can be made only by $p\sigma$ (by the primitive update task independence assumption), it follows that the only way to violate $\texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$ over $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ is to cause a transition $\mathbf{s}_2 \xrightarrow{r'} \mathbf{s}_3$ using some r' that matches $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$. But this contradicts the assumption that $\mathbf{P}, \mathbf{s}_2 ... \mathbf{s}_n \vDash \texttt{suspend}(\bar{r}')$ holds. Therefore $\mathbf{P}, \mathbf{s}_1 ... \mathbf{s}_n \vDash \texttt{not\_right\_after}(\bar{p} \nrightarrow \bar{r})$ holds.

Case (i): In this case, $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{p} \leftarrow \bar{r}) \in \mathcal{C}'$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$, and $\texttt{right\_before}(\bar{p} \leftarrow \bar{r}) \in \mathcal{C}$. Since the path $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies the constraint $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{p} \leftarrow \bar{r})$, we have that either (i) $\mathbf{s}_2 \xrightarrow{\bar{r}'} \mathbf{s}_3$ holds and $\langle \mathbf{s}_3, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{right\_before}(\bar{p} \leftarrow \bar{r})$; or (ii) $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{right\_before}(\bar{p} \leftarrow \bar{r})$. In either case, since $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ holds by the assumption, it follows that $\texttt{right\_before}(\bar{p} \leftarrow \bar{r})$ holds on the entire execution path $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$.

Case (j): Here $\texttt{right\_before}(\bar{s} \leftarrow \bar{r}) \in \mathcal{C}'$ and $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{s} \leftarrow \bar{r}) \in \mathcal{C}$, where $\bar{s}, \bar{r}$ are arbitrary task patterns. We thus have that $\mathbf{P}, \langle \mathbf{s}_2 ... \mathbf{s}_n \rangle \vDash \texttt{right\_before}(\bar{s} \leftarrow \bar{r})$ and $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ both hold. By the premise of rule 6, either

- $p\sigma$ does not match $\bar{r}$; or
- $p\sigma$ matches $\bar{r}'$.

Since the constraint $\texttt{right\_before}(\bar{s} \leftarrow \bar{r})$ is satisfied by $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$, in either of these cases the constraint $\texttt{next\_right\_before}(\bar{r}' \leftsquigarrow \bar{s} \leftarrow \bar{r}) \in \mathcal{C}$ holds on $\langle \mathbf{s}_1, ..., \mathbf{s}_n \rangle$.

Case (k): In this case, $\mathcal{C}'$ contains $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$ and $\texttt{before}(\bar{r}' \dashleftarrow \bar{s}')$, where $\bar{r}' = \texttt{refine}(\bar{r}; p\sigma, \bar{p})$ and $\bar{s}' = \texttt{refine}(\bar{s}; p\sigma, \bar{p})$, while $\mathcal{C}$ contains $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$ instead. In addition, $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ holds. Since $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{before}(\bar{r}' \dashleftarrow \bar{s}')$ and $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$, it follows that $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{between}(p\sigma \dashrightarrow \bar{r}' \dashleftarrow \bar{s}')$. Therefore, $\texttt{between}(\bar{p} \dashrightarrow \bar{r} \dashleftarrow \bar{s})$ cannot be violated by $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$.

Case (l): Here $\mathcal{C}'$ contains $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s})$ and $\texttt{blocks}(\bar{r}' \not\dashrightarrow \bar{s}')$, while $\mathcal{C}$ contains the constraint $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s})$. In addition, $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ holds. Since $\langle \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{blocks}(\bar{r}' \not\dashrightarrow \bar{s}')$, and $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s})$, it follows that $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$ satisfies $\texttt{not\_between}(p\sigma \not\dashrightarrow \bar{r}' \not\dashleftarrow \bar{s}')$. Therefore, $\texttt{not\_between}(\bar{p} \not\dashrightarrow \bar{r} \not\dashleftarrow \bar{s})$ cannot be violated by $\langle \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n \rangle$.

Case (m): In this catch-all case, all constraints are copied over from $\mathcal{C}$ to $\mathcal{C}'$ except the following:

- $\texttt{atleast}_1(\bar{p})$: In this case, $\mathcal{C}$ has $\texttt{atleast}_1(\bar{p})$, while $\mathcal{C}'$ does not. Recall that the initial transition was $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ and $p\sigma$ matches $\bar{p}$. Therefore, $\mathbf{P}, \mathbf{s}_1 \dots \mathbf{s}_n \models \texttt{atleast}_1(\bar{p})$ holds.
- $\texttt{force}(\bar{p})$: In this case $\mathcal{C}$ has $\texttt{force}(\bar{p})$, while $\mathcal{C}'$ does not. Since the initial transition was $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$ and $p\sigma$ matches $\bar{p}$, we conclude that $\mathbf{P}, \mathbf{s}_1 \dots \mathbf{s}_n \models \texttt{force}(\bar{p})$ holds. (Note that the case of $\texttt{force}(\bar{r}) \in \mathcal{C}$, where $p\sigma$ does not match $\bar{r}$, is precluded by the precondition to rule 6, so we are not missing cases by considering $\texttt{force}(\bar{p})$ only.)
- $\texttt{suspend}(\bar{r})$, for some task pattern $\bar{r}$: Here $\mathcal{C}$ has $\texttt{suspend}(\bar{r})$, but $\mathcal{C}'$ does not. Suppose $\mathbf{P}, \mathbf{s}_1 \dots \mathbf{s}_n \models \texttt{suspend}(\bar{r})$ does *not* hold. This means that the initial transition from $\mathbf{s}_1$ to $\mathbf{s}_2$ was caused by a task that matches $\bar{r}$. At the same time, the initial transition was $\mathbf{s}_1 \xrightarrow{p\sigma} \mathbf{s}_2$, so $p\sigma$ must match $\bar{r}$. However, this contradicts the precondition for rule 6, which says that $p\sigma$ must not match any pattern that appears inside a $\texttt{suspend}$ constraint in $\mathcal{C}$, including $\bar{r}$. Thus $\mathbf{P}, \mathbf{s}_1 \dots \mathbf{s}_n \models \texttt{suspend}(\bar{r})$ holds. $\square$

B. Completeness of the Inference System

Theorem 3 (Completeness of the Inference System). *Suppose* $\mathbf{P}$ *is a set of composite task definitions,* s *a database identifier,* ψ *a service process,* $\mathcal{C}$ *a conjunction of primitive or serial constraints, and* ψ *and* $\mathcal{C}$ *satisfy the service contracts assumption. Then:*

$$\mathbf{P}, \boldsymbol{s} \text{---} \models (\psi \wedge \mathcal{C}) \quad \textit{implies} \quad \boldsymbol{P}, \boldsymbol{s} \text{---} \vdash (\psi \wedge \mathcal{C})$$

Proof: The proof of completeness is based on the following facts:

- The original CTR proof theory is complete.
- The current proof theory restricts the original CTR proof theory by eliminating certain derivation paths. Soundness implies that all the remaining paths satisfy $\mathcal{C}$. Completeness means that none of the eliminated path satisfies $\mathcal{C}$.

The key step in the proof is to show that the eliminated derivation paths correspond to executions that violate some of the constraints.

First, by the inference rule 1, it is always possible to ensure that $\mathcal{C}$ in rule 6 and in the axiom is a conjunctive constraint.

One reason why a derivation path may be eliminated by the proof theory is that in the inference rule 6 (executing primitive update tasks) a *hot* task $p\sigma$ is blocked because $\mathcal{C}$ contains:

- $\texttt{absence}(\bar{p})$
- $\texttt{force}(\bar{r})$ such that $p\sigma$ does not match $\bar{r}$
- $\texttt{before}(\bar{r} \dashleftarrow \bar{p} \setminus \{\bar{p}_1, \dots, \bar{p}_k\})$ $k \geq 0$ and $p\sigma$ does not match any of the $\bar{p}_i$'s
- $\texttt{right\_before}(\bar{r} \leftarrow \bar{p})$
- $\texttt{next\_right\_before}(\bar{p}' \leftsquigarrow \bar{r} \leftarrow \bar{p})$, where $p\sigma$ does not match $\bar{p}'$.

where $\bar{r}$ is an arbitrary task pattern and $p\sigma$ matches $\bar{p}$.

If such a *hot* task $p\sigma$ is executed at some state, s_1, and causes a transition to state s_2 then either $\texttt{absence}(\bar{p})$, or $\texttt{before}(\bar{r} \dashleftarrow \bar{p} \setminus \{\bar{p}_1, \dots, \bar{p}_k\})$, $\texttt{right\_before}(\bar{r} \leftarrow \bar{p})$, or $\texttt{next\_right\_before}(\bar{p}' \leftsquigarrow \bar{r} \leftarrow \bar{p})$, or $\texttt{force}(\bar{r})$ are not satisfied on any path of the form $\langle \mathrm{s}_1 \mathrm{s}_2 \dots \mathrm{s}_n \rangle$. Therefore, such an eliminated path is not a valid execution.

The other case when our proof theory may eliminate an execution path that would have been otherwise found by the plain CTR proof procedure is when we derive $\mathbf{P}, \mathbf{s} \text{---} \vdash () \wedge \mathcal{C}$ but cannot declare success because the axiom does not apply due to the fact that $\mathcal{C}$ contains a constraint of the form $\texttt{force}(\bar{r})$, $\texttt{atleast}_k(\bar{r})$, or $\texttt{exactly}_k(\bar{r})$, where $k \geq 1$. But then, if the eliminated execution path satisfied the original constraint $\mathcal{C}$ then, by soundness of the inference rules, the path $\langle \mathbf{s} \rangle$ would have to satisfy $\mathcal{C}$ and thus also one of the aforesaid constraints: $\texttt{force}$, $\texttt{atleast}$, or $\texttt{exactly}$, which is not the case.

Since the remaining inference rules do not restrict execution of transactions and are essentially identical to the corresponding rules in CTR (except that the constraints are tacked on to them), the result follows. □

C. Representing Constraints of *ServLog* as CTR Formulas

This appendix provides direct definitions of the constraints introduced $\mathcal{C}\textsc{onstr}$ (Section 4.2) and the additional ones from Section 5.

The followings are CTR representations for the constraints in $\mathcal{C}\textsc{onstr}$ that are equivalent to the semantic definitions for these constraints given in Sections 4.2 and 5. In the formulas below we use $vars(\bar{t})$ to represent the set of variables appearing in $\bar{t}$,[17] and $\blacktriangledown p(\bar{t})$ denotes[18]

$$\mathtt{path} \otimes p(\bar{t}) \otimes \mathtt{path} \tag{14}$$

The constraints in $\mathcal{C}\textsc{onstr}$ are now represented in CTR as follows:

– $\mathtt{atleast}_n(p(\bar{t}))$:

$$\exists\, vars(\cup_{i=1}^{n} \overline{t_i})\ \ (\blacktriangledown p(\overline{t_1}) \otimes ... \otimes \blacktriangledown p(\overline{t_n})) \tag{15}$$

Where $\overline{t_i}$ is $\bar{t}$ in which all underscores (unnamed placeholders) are replaced with new variables. (Recall that named placeholders *are* regular variables.)

– $\mathtt{absence}(p(\bar{t}))$:

$$\forall\, vars(\bar{t})\ \ \neg \blacktriangledown p(\bar{t}) \tag{16}$$

– $\mathtt{exactly}_n(p(\bar{t}))$:

$$\mathtt{atleast}_n(p(\bar{t})) \wedge \neg \mathtt{atleast}_{n+1}(p(\bar{t})) \tag{17}$$

– $\mathtt{after}(p(\bar{t})\ \dashrightarrow\ q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\bar{t})\ \ \exists\, vars(\overline{u}) \setminus vars(\bar{t}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow \blacktriangledown q(\overline{u}) \end{array} \tag{18}$$

Here $vars(\overline{u}) \setminus vars(\bar{t})$ is the set of variables in $\overline{u}$ minus those that appear in $\bar{t}$.

– $\mathtt{before}(p(\bar{t})\ \leftarrow\!\!-\ q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\overline{u})\ \ \exists\, vars(\bar{t}) \setminus vars(\overline{u}) \\ \quad \blacktriangledown p(\bar{t}) \Leftarrow q(\overline{u}) \otimes \mathtt{path} \end{array} \tag{19}$$

– $\mathtt{before}(p(\bar{t})\ \leftarrow\!\!-\ q(\overline{u}) \setminus \{q(\overline{v}_1), ..., q(\overline{v}_n)\})$:

$$\begin{array}{l} \forall\, vars(\overline{u})\, \forall\, vars(\cup_{i=1}^{n} \overline{v_i})\ \ \exists\, vars(\bar{t}) \setminus vars(\overline{u}) \\ \quad \bigvee_{i=1}^{n} (\overline{u} = \overline{v_i}) \vee (\blacktriangledown p(\bar{t}) \Leftarrow q(\overline{u}) \otimes \mathtt{path}) \end{array} \tag{20}$$

[17]Note that all underscores (unnamed placeholders) are replaced with new variables, e.g. $vars(p(\_,\_)) = \{?X, ?Y\}$.

[18]Informally, $\blacktriangledown p(\bar{t})$ means that $p(\bar{t})$ eventually executes.

– $\mathtt{blocks}(p(\bar{t}) \not\dashrightarrow q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\bar{t}, \overline{u}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow \neg \blacktriangledown q(\overline{u}) \end{array} \tag{21}$$

– $\mathtt{between}(p(\bar{t}) \dashrightarrow q(\overline{u}) \dashleftarrow r(\overline{v}))$:

$$\begin{array}{l} \forall\, vars(\bar{t}, \overline{v}) \quad \exists\, vars(\overline{u}) \setminus vars(\bar{t}, \overline{v}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow \blacktriangledown q(\overline{u}) \Leftarrow r(\overline{v}) \otimes \mathtt{path} \end{array} \tag{22}$$

– $\mathtt{not\_between}(p(\bar{t}) \not\dashrightarrow q(\overline{u}) \not\dashleftarrow r(\overline{v}))$:

$$\begin{array}{l} \forall\, vars(\bar{t}, \overline{u}, \overline{v}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow \neg \blacktriangledown q(\overline{u}) \Leftarrow r(\overline{v}) \otimes \mathtt{path} \end{array} \tag{23}$$

– $\mathtt{right\_after}(p(\bar{t}) \rightarrow q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\bar{t}) \;\; \exists\, vars(\overline{u}) \setminus vars(\bar{t}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow q(\overline{u}) \otimes \mathtt{path} \end{array} \tag{24}$$

– $\mathtt{right\_before}(p(\bar{t}) \leftarrow q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\overline{u}) \;\; \exists\, vars(\bar{t}) \setminus vars(\overline{u}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Leftarrow q(\overline{u}) \otimes \mathtt{path} \end{array} \tag{25}$$

– $\mathtt{next\_right\_before}(q(\overline{u}') \leftsquigarrow p(\bar{t}) \leftarrow q(\overline{u}))$:

$$\begin{array}{l} \mathtt{right\_before}(p(\bar{t}) \leftarrow q(\overline{u})) \;\bigvee \\ \quad (\exists\, vars(\overline{u})\; q(\overline{u})) \otimes \mathtt{right\_before}(p(\bar{t}) \leftarrow q(\overline{u})) \end{array} \tag{26}$$

– $\mathtt{not\_right\_after}(p(\bar{t}) \nrightarrow q(\overline{u}))$:

$$\begin{array}{l} \forall\, vars(\bar{t}, \overline{u}) \\ \quad \mathtt{path} \otimes p(\bar{t}) \Rightarrow (\mathtt{arc} \wedge \neg \blacktriangledown q(\overline{u})) \Leftarrow q(\overline{u}) \otimes \mathtt{path} \end{array} \tag{27}$$

where $\mathtt{arc}$ is a CTR proposition such that $\mathbf{s}_1 \xrightarrow{\mathtt{arc}} \mathbf{s}_2$ is true for any pair of states $\mathbf{s}_1, \mathbf{s}_2$ and $\mathtt{arc}$ is not true on any other path.

– $\mathtt{force}(p(\bar{t}))$:

$$\exists\; vars(\bar{t}) \;\; p(\bar{t}) \otimes \mathtt{path} \tag{28}$$

– $\mathtt{suspend}(p(\bar{t}))$:

$$\forall\, vars(\bar{t}) \;\; (\mathtt{arc} \wedge \neg p(\bar{t})) \otimes \mathtt{path} \tag{29}$$

– **Composite constraints**: If C_1, $C_2 \in \mathcal{C}onstr$ then so are $C_1 \wedge C_2$ (a conjunctive constraint) and $C_1 \vee C_2$ (a disjunctive constraint).

References

[1] S. Abiteboul, P. Bourhis, and V. Vianu. Comparing workflow specification languages: a matter of views. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 78–89, New York, NY, USA, 2011. ACM.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Expressing and verifying business contracts with abductive logic programming. In *Normative Multi-agent Systems*, number 07122 in Dagstuhl Seminar Proceedings, 2007.

[3] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *Int. J. Softw. Tools Technol. Transf.*, 8(6):485–516, 2006.

[4] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Modeling bitcoin contracts by timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 7–22. Springer, 2014.

[5] R. Basseda, M. Kifer, and A. J. Bonner. Planning with transaction logic. In *Web Reasoning and Rule Systems, 8th International Conference (RR 2014)*, volume 8741 of *Lecture Notes in Computer Science*, Athens, Greece, September 2014. Springer.

[6] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3):20:1–20:28, July 2010.

[7] A. Bonner. Workflow, transactions, and datalog. In *ACM Symposium on Principles of Database Systems*, pages 294–305, Philadelphia, PA, May/June 1999.

[8] A. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Comput. Sci.*, 133:205–265, 1994.

[9] A. Bonner and M. Kifer. Transaction Logic Programming (or A Logic of Declarative and Procedural Knowledge). Technical Report CSRI-323, University of Toronto, November 1995.

[10] A. Bonner and M. Kifer. A Logic for Programming Database Transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer, 1998.

[11] A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Joint International Conference and Symposium on Logic Programming*, 1996.

[12] D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data-aware process analysis: A database theory perspective. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 1–12. ACM, 2013.

[13] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. In *WS-FM*, pages 148–162, 2006.

[14] M. Comuzzi and B. Pernici. A framework for qos-based web service contracting. *ACM Trans. Web*, 3:10:1–10:52, July 2009.

[15] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Transactions on Database Systems*, 37(3):1–36, September 2012.

[16] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *PODS*, pages 25–33, 1998.

[17] H. Davulcu, M. Kifer, and I. Ramakrishnan. CTR–S: A Logic for Specifying Contracts in Semantic Web Services. In *WWW2004*, pages 144+, 2004.

[18] A. Deutsch, R. Hull, and V. Vianu. Automatic verification of database-centric systems. *ACM SIGMOD Record*, 43(3):5–17, 2014.

[19] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113 – 137, 2002.

[20] P. Fodor and M. Kifer. Tabling for transaction logic. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 199–208, New York, NY, USA, 2010. ACM.

[21] P. Fodor and M. Kifer. Transaction logic with defaults and argumentation theories. In *ICLP (Technical Communications)*, pages 162–174, 2011.

[22] T. Fr uwirth. A devil's advocate against termination of direct recursion. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, July 2015.

[23] A. S. Gomes and J. J. Alferes. Extending transaction logic with external actions. *TPLP*, 13(4-5-Online-Supplement), 2013.

[24] G. Governatori, Z. Milosevic, S. Sadiq, and M. Orlowska. On compliance of business processes with business contracts. Technical report, File System Repository [http://search.arrow.edu.au/apps/ArrowUI/OAIHandler] (Australia), 2007.

[25] G. Governatori and S. Sadiq. The journey to business process compliance. In *Handbook of Research on BPM*, pages 426–454. IGI Global, 2008.

[26] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management. In *ICEBE*, pages 535–540, 2005.

[27] M. Hepp and D. Roman. An ontology framework for semantic business process management. In *Wirtschaftsinformatik (1)*, pages 423–440, 2007.

[28] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. *On the Move to Meaningful Internet Systems: OTM 2008*, pages 1152–1163, 2008.

[29] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel. A logical framework for web service discovery. In *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, volume 119. Hiroshima, J apan, 2004.

[30] S. Liang and M. Kifer. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming*, 13:705–719, September 2013.

[31] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In J. Cardoso and A. Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[32] M. Montali, F. Chesani, P. Mello, and F. M. Maggi. Towards data-aware constraints in declare. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1391–1396. ACM, 2013.

[33] M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM Trans. Web*, 4:3:1–3:62, January 2010.

[34] S. Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[35] M. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12, Dec 2003.

[36] M. Rezk and M. Kifer. Transaction logic with partially defined actions. *J. Data Semantics*, 1(2):99–131, 2012.

[37] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[38] D. Roman and M. Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *VLDB*, pages 627–638, 2007.

[39] D. Roman and M. Kifer. Semantic web service choreography: Contracting and enactment. In *International Semantic Web Conference*, pages 550–566, 2008.

[40] D. Roman, J. Kopeckỳ, T. Vitvar, J. Domingue, and D. Fensel. Wsmo-lite and hrests: Lightweight semantic annotations for web services and restful apis. *Web Semantics: Science, Services and Agents on the World Wide Web*, 31:39–58, 2015.

[41] P. G. S. Angelov. B2B E-Contracting: A Survey of Existing Projects and Standards. Report I/RS/2003/119, Telematica Instituut, 2003.

[42] P. Senkul, M. Kifer, and I. Toroslu. A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In *VLDB 2002*, pages 694–705, 2002.

[43] J. Spohrer and W. Murphy. Service science. In S. Gass and M. Fu, editors, *Encyclopedia of Operations Research and Management Science*, pages 1385–1392. Springer US, 2013.

[44] H. Staffler. A system for modeling and reasoning about constrained processes. Master's thesis, Univeristy of Innsbruck, September 2009.

[45] W. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23(2):99–113, 2009.

[46] L. Vasiliu, S. Harand, and E. Cimpian. The DIP project: Enabling systems & solutions for processing digital content with semantic web services. In *EWIMT*, 2004.

[47] V. Vianu. Automatic verification of database-driven systems: a new frontier. In *Proceedings of the 12th International Conference on Database Theory*, pages 1–13. ACM, 2009.

[48] M. Westergaard. Cpn tools 4: Multi-formalism and extensibility. In *Application and Theory of Petri Nets and Concurrency*, pages 400–409. Springer, 2013.

[49] M. Westergaard and T. Slaats. Mixing paradigms for more comprehensible models. In *Business Process Management*, pages 283–290. Springer, 2013.