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 [36], and newer approaches such as artifact-centric [29] and data-aware systems [13], microservices [35], 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 [20], OWL-S [32], WSMO [40], SWSL, DIP [46], and SUPER [28,27], and more recently [41]. Nevertheless, many core issues remained largely 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

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://sp-super.org/

∗Corresponding author, e-mail: dumitru.roman@sintef.no
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 aspects of a service, such as 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 [17,42,18,38,39]. 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 [7,37,24,8,22]. 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 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.

\( ^6\)http://www.omg.org/spec/BPMN/2.0/
\( ^7\)http://www.wsmo.org/TR/d14/
\( ^8\)http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/
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.

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).\footnote{This non-determinism has an XOR flavor.} 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 \((\text{Order#},\text{Price})\), meaning that, to execute handle_payment, Order# and Price must be provided.\footnote{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.} 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.\footnote{ServLog is independent of the choice of such a shared space.} 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 \(\text{Order#} \in \text{process_order_items(Order#)}\) is the same as the data identified by \(\text{Order#} \in \text{place_order(Order#,Price)}\), 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 sub-tasks 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.
Fig. 2. Service process example: A hierarchical control and data flow graph.

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 \texttt{select\_item}(\texttt{Order\#},\texttt{Item}) needs to pass \texttt{Order\#} and \texttt{Item} to \texttt{book\_shipper}(\texttt{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: \texttt{select\_item} can store \texttt{Order\#} and \texttt{Item}, and \texttt{book\_shipper} can read them later. This is depicted by the dashed arc going from \texttt{select\_item} to \texttt{book\_shipper}. The label on the arc represents the data items that are being passed. In our case, \texttt{Order\#} is being passed as \texttt{Ord\#} and \texttt{Item} as \texttt{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, \texttt{place\_order} produces \texttt{Order\#}-items, and that item is then shared with \texttt{process\_order\_items}, \texttt{handle\_shippers}, and \texttt{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, the data items that we employ for passing data through a shared database are consumable.

Another aspect of the service process graphs concerns transition conditions on arcs. For example, \texttt{producer(Item,Producer)} in the definition of \texttt{process\_item} and \texttt{itemAvailable(Order\#,Producer,Item)} or \texttt{itemUnavailable(Order\#,Producer,Item)} in the definition of \texttt{inform\_client}.\footnote{Such conditions represent relations queried by service tasks, but which the tasks do not modify (and thus are not used for data passing). That data is not being consumed by the service tasks.} Another example of transition condition is a test of the form \texttt{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, \texttt{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 \texttt{ServLog}'s reasoning system. It does not mean that the work performed by the task is trivial. For example, \texttt{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 6 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 re-

---

**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.
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 $\lor, \land, \neg, \forall$, and $\exists$, CTR has two additional infix connectives, $\otimes$ (serial conjunction) and $\oplus$ (concurrent conjunction), and a modal operator $\odot$ (isolated execution). For instance,

$$\odot(p(X) \otimes q(X)) \mid (\forall Y (r(Y) \lor s(X,Y)))$$

is a well-formed formula in CTR, while Figure 4 contains an example of a well-formed formula that represents the top-level composite task of Figure 2.

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 $s_1, s_2, \ldots, s_n$ are state identifiers, then $(s_1), (s_1, s_2),$ and $(s_1, s_2, \ldots, s_n)$ 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, $\phi$, is true over a path $(s_1, s_2, \ldots, s_n)$, it means that $\phi$ can execute starting at state $s_1$. During the execution, the current state will change to $s_2, s_3, \ldots$ etc., and the execution terminates at state $s_n$. In such a case we will also call the path $(s_1, \ldots, s_n)$ an execution of $\phi$.

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 $\phi$ 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 $\phi$ 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 $\phi$ execute not on paths but on multi-paths, i.e., on sequences of paths. Execution of one part of $\phi$ 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 \( (\pi_1, \ldots, \pi_k) \) of paths. If \( \mu = (\pi_1, \ldots, \pi_k) \) and \( \mu' = (\pi'_1, \ldots, \pi'_n) \) are two m-paths, their concatenation, \( \mu \cdot \mu' \), is the m-path \( (\pi_1, \ldots, \pi_k, \pi'_1, \ldots, \pi'_n) \) and their interleaving, \( \mu \parallel \mu' \), is an m-path of the form \( (\kappa_1, \ldots, \kappa_{k+n}) \) such that it is a topological sort of the two sequences \( \mu \) and \( \mu' \). For example, one interleaving of \( (\{s_1, s_2, s_3\}, \{s_4, s_5\}) \) and of \( (\{s_1, s_2\}, \{s_3, s_4, s_5, s_6\}) \) is \( (\{s_1, s_2, s_3, s_4, s_5, s_6\}) \). Also, \((\{s_1, s_2, s_3\}, \{s_4, s_5\}, \{s_6, s_7\})\) is another interleaving, meanwhile \((\{s_1, s_2, s_3\}, \{s_4, s_5\})\), \((\{s_1, s_2\}, \{s_3, s_4, s_5\})\), \((\{s_1, s_2, s_3\}, \{s_4, s_5\}, \{s_6, s_7\})\) is a degenerate interleaving.

Finally, a path \( \pi = (s_1, \ldots, s_m) \) is a merge of an m-path \( (\pi_1, \ldots, \pi_n) \) if there are integers \( 1 = i_0 \leq i_1 \leq i_2 \leq \ldots \leq i_{n-1} \leq i_n = m \) such that \( \pi_1 = (s_{i_0}, \ldots, s_{i_1}) \), \( \pi_2 = (s_{i_1}, \ldots, s_{i_2}) \), \( \ldots \), \( \pi_{n-1} = (s_{i_{n-2}}, \ldots, s_{i_{n-1}}) \), \( \pi_n = (s_{i_{n-1}}, \ldots, s_{i_n}) \). Note that for the merge to be possible, the end-state of each path \( \pi_i \) in the m-path must be the start-state of the subsequent path \( \pi_{i+1} \) for each \( 1 \leq i < n \). For instance, \((s_1, s_2, s_3, s_4, s_5, s_6)\) is a merge of the m-path \((\{s_1, s_2\}, \{s_2, s_3, s_4\}, \{s_1\}, \{s_3, s_4, s_5, s_6\})\).

A multi-path structure is a mapping that, for each multi-path \( \mu \), tells which ground atomic formulas are true on \( \mu \). Informally, this can be understood as telling which ground atomic transactions can execute along \( \mu \). 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 \( s \) is a state identifier and \( p \) is a fact that is true in the database associated with \( s \) then \( p \) is true over the path \( (s) \) (and the m-path \( (s) \)).

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

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:

- \( \phi \) is true over a path, \( \pi \), if it is true over an m-path, \( \mu \), and \( \pi \) is a merge of \( \mu \).

CTR contains a special propositional constant, \( \text{state} \), which is true only on paths of length 1, that
Definition 3.1 (Concurrent-Horn goal) A concurrent-Horn goal is either an atomic formula or has the form \( \phi \otimes \psi \), \( \phi \mid \psi \), \( \phi \lor \psi \), or \( \otimes \phi \), where \( \phi \) and \( \psi \) 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 \( \exists X \phi \), where \( \phi \) is a CTR goal and \( X \) is the set of variables that occur in \( \phi \).

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

\[
\forall X \ (\text{head} \leftarrow \text{body})
\]

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

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 \( \land \) connective thus enabling execution of constrained transactions, which are non-Horn. We also deal with a much larger class of constraints than [17,38], 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 \((s_1,s_2)\) belongs to the relation corresponding to a primitive update \(u\), it means that \(u\) can cause a transition from state \(s_1\) to state \(s_2\). We will conveniently represent this kind of situation using the following notation:

\[
s_1 \xrightarrow{u} s_2 \tag{2}
\]

Usually the binary relations that represent primitive updates are defined outside of CTR. In that case, they are called transition oracles [9,11,12,10]. 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 [37]. 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., \(\text{place}_\text{order}(\text{Order}\#\!,\text{Price})\)). It should be understood that such a predicate represents a family of related updates, one for each instantiation of the variables. Clearly, \(\text{place}_\text{order}(12365409,5123)\) and \(\text{place}_\text{order}(9865412,321)\) 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 \(\text{path}\) introduced above. For example, \(\text{path} \otimes \psi\otimes \text{path}\) is a constraint that is true on a path if \(\psi\) 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: \(\psi \leftarrow \phi \define \neg(\neg\psi \otimes \phi)\) and
\( \psi \Rightarrow \phi \overset{def}{=} \neg (\psi \land \neg \phi) \). A moment's reflection should convince the reader that \( \psi \iff \phi \) means that whenever \( \phi \) occurs then \( \psi \) must have occurred just before it and that \( \psi \Rightarrow \phi \) means that whenever \( \psi \) occurs then \( \phi \) must occur right after it. Thus, \( \text{path} \Rightarrow \psi \iff \text{path} \) constrains executions to be such that every subpath encountered in the course of the execution satisfies \( \psi \) (including subpaths of the form \( (s) \), where \( s \) 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, \( \phi \) is a CTR formula and \( s_0, s_1, \ldots, s_n \) is a sequence of database states. Then the statement

\[ P, s_0 \models s_1 \ldots s_n \models \phi \]  

is true if and only if \( M, (s_0, s_1, \ldots, s_n) \models \phi \) (i.e., \( \phi \) is true on the path \( (s_0, s_1, \ldots, s_n) \) in \( M \)), for every multi-structure \( M \) that satisfies \( P \).

Related to this is the statement

\[ P, s_0 \models \vdash \phi \]  

which are true if (3) is true for some sequence of database states \( s_0, s_1, \ldots, s_n \).

The aforementioned proof theory for CTR assumes that \( P \) is a set of concurrent-Horn rules and it manipulates the statements of the form \( P, s_0 \models \vdash \phi \). It is sound and complete in the sense that there is a proof of \( P, s_0 \models \vdash \phi \) if and only if \( P, s_0 \models \models \phi \) 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

\[ p(t_1, \ldots, t_n) \]  

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 \( p(\bar{T}) \), \( p(\bar{U}) \), etc., where \( \bar{T}, \bar{U}, \ldots \) 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 built-in 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
Definition 4.3 (Task occurrence) A task \( p \) occurs in a CTR goal \( \Omega \) if \( \Omega \) contains a task atom \( p(T) \) for some \( T \).

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 \( q(T) \leftarrow \Omega \) and \( p \) occurs in \( \Omega \).

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

\[
q(T) \leftarrow \Omega
\]

where \( q \) does not occur in \( \Omega \) and none of the tasks that occur in \( \Omega \) have \( q \) as a subtask.

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

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

Here are examples of iterative rules:

\[
\begin{align*}
q(\text{?}X) & \leftarrow (p(\text{?}X, \text{?}Y) \otimes q(\text{?}Y) \otimes r(\text{?}Y, \text{?}Z)) \lor s(\text{?}Y) \\
q(\text{?}X) & \leftarrow p(\text{?}X, \text{?}Y) \otimes q(\text{?}Y) \otimes qq(\text{?}Y, \text{?}Z) \\
qq(\text{?}X) & \leftarrow (pp(\text{?}X, \text{?}Y) \otimes q(\text{?}Y)) \lor t(\text{?}X)
\end{align*}
\]

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

\[
\begin{align*}
q(T) & \leftarrow \Phi \otimes q(U) \\
q(T) & \leftarrow \Psi
\end{align*}
\]

where \( \Phi \) and \( \Psi \) do not depend on \( q \).

Definition 4.8 (Service process) A service process is a pair \( (\Omega, R) \), where \( \Omega \) 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 \( \Omega \) 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 5.

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. In order 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 contact_producer(?Producer, ?Item) task. 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 5, 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 sub-tasks 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 5, 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 6 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 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 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, \( \theta \) such that \( \theta(pattern) = task_atom \). Since task_atom is ground, \( \theta \) 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 \( \theta \) 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:

\[
\text{refine}(\text{out_pattern}; \text{in_ground_task}, \text{in_pattern}) = \text{refined_pattern}
\]

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 \( \theta \) be the substitution that matches in_pattern against in_ground_task. If out_pattern has variables other than those in in_pattern, \( \theta \) can map them to anything (to some other variable or constant). Then

\[
\text{refined_pattern} = \theta(\text{out_pattern})
\]

One can verify by direct inspection that the task atom \( p(2, 5, \_f, \_f, \_f, \_f, \_f) \) matches the pattern \( p(_, 1, \_f, \_f) \), that

\[
\text{refine}(p(_, 5, \_f); p(1, 2, 3), p(_, 5, \_f)) = p(2, 5, \_f),
\]

and that
Goal:
\[ \text{place\_order}(\text{?Order\#}, \text{?Price}) \odot \] // primitive update
\[ \text{(process\_order\_items}(\text{?Order\#}) \mid \text{handle\_shippers}(\text{?Order\#}) \mid \text{handle\_payment}(\text{?Order\#}, \text{?Price})) \odot \] // composite tasks
\[ \text{end\_order}(\text{?Order\#}) \odot \] // primitive update

Rules:
\[ \text{process\_order\_items}(\text{?Order\#}) \leftarrow \] // composite task
\[ \text{select\_item}(\text{?Order\#}, \text{?Item}) \odot \] // primitive update that inserts selected_item(…) into database
\[ \text{process\_item}(\text{?Order\#}, \text{?Item}) \odot \] // database query
\[ \text{process\_order\_items}(\text{?Order\#}) \odot \] // composite task
\[ \text{process\_item}(\text{?Order\#}, \text{?Item}) \leftarrow \text{state} \] // composite task
\[ \text{producer}(\text{?Item}, \text{?Producer}) \odot \] // database query
\[ \text{contact\_producer}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{inform\_client}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{handle\_shippers}(\text{?Ord\#}) \leftarrow \text{contact\_shipper}(\text{?Ord\#}, \text{?Shipper}) \odot \] // database query
\[ \text{selected\_item}(\text{?Ord\#}, \text{?Itm}) \odot \] // database query
\[ \text{book\_shipper}(\text{?Ord\#}, \text{?Shipper}, \text{?Itm}) \odot \] // database query
\[ \text{deliver}(\text{?Ord\#}) \odot \] // database query
\[ \text{handle\_payment}(\text{?Order\#}, \text{?Price}) \leftarrow \text{pay}(\text{?Order\#}) \] // database query
\[ \text{handle\_payment}(\text{?Order\#}, \text{?Price}) \leftarrow \text{payment\_guarantee}(\text{?Order\#}, \text{?Credit\#}) \] // database query
\[ \text{inform\_client}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \leftarrow \text{item\Additional{Available}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{inform\_availability}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{confirm\_producer}(\text{?Order\#}, \text{?Producer}) \odot \] // database query
\[ \text{inform\_client}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \leftarrow \text{item\Additional{Unavailable}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{inform\_unavailability}(\text{?Order\#}, \text{?Producer}, \text{?Item}) \odot \] // database query
\[ \text{pay}(\text{?Ord\#}) \leftarrow \text{full\_payment}(\text{?Ord\#}) \odot \text{pay\_per\_item}(\text{?Ord\#}) \odot \text{pay\_per\_item}(\text{?Ord\#}) \odot \text{pay\_per\_item}(\text{?Ord\#}) \odot \text{state} \]

Fig. 5. ServLog representation of the service process from Figure 2.

\[ \text{refine}(q(_f, 5, _f, _h); (1, 2, 2, 3), p(_g, _f, _f, _3)) = q(2, 5, 2, _h). \]

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 \( \bar{T}, \bar{p}, \text{etc.} \), to represent tuples that include placeholders as some of the arguments in task patterns. The uppercase symbols \( \bar{T}, \bar{U}, \text{etc.} \) 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 \( \text{CONSTRAINTS} \) 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:**
   
   \( - \text{atleast}_n(p(\bar{T})): \text{task } p \text{ must execute at least } n \text{ times } (n \geq 1). \)

   Formally, an execution \( \langle s_1, \ldots, s_m \rangle \) satisfies this constraint if and only if there are ground
task atoms \( p(T_1), \ldots, p(T_n) \) that executed at some states \( s_1, \ldots, s_n \) (i.e., \( s_i \xrightarrow{p(T_i)} s_{i+1}, \ldots, s_n \xrightarrow{p(T_n)} s_{n+1} \) such that \( p(T) \) matches \( p(T_1) \), \( p(T_2), \ldots, p(T_n) \).\(^{13}\)

- \( \text{absence}(p(T)) \): task \( p \) must not execute.

Formally, an execution \( \langle s_1, \ldots, s_m \rangle \) satisfies this constraint if and only if whenever there is a state \( s_i \) in that execution such that \( s_i \xrightarrow{p(T)} s_{i+1} \) and \( p(T) \) matches \( p(T) \).

- \( \text{exactly}_n(p(T)) \): task \( p \) must execute exactly \( n \) times \( (n \geq 1) \).

An execution \( \langle s_1, \ldots, s_m \rangle \) satisfies this constraint if and only if it satisfies \( \text{atleast}_n(p(T)) \) but not \( \text{atleast}_{n+1}(p(T)) \).

2. Serial constraints:

- \( \text{after}(p(T) \rightarrow q(\Pi)) \): 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 \( \langle s_1, \ldots, s_m \rangle \) satisfies this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{q(\Pi)} s_{i+1} \) and \( q(\Pi) \) matches \( q(\Pi) \), there must be a state \( s_j \) in that same execution such that \( j \geq i + 1 \), \( s_j \xrightarrow{p(T)} s_{j+1} \), and \( \text{refine}(q(\Pi); p(T); p(T)) \) matches \( q(\Pi) \).

For instance, if the above constraint has the form \( \text{after}(p(_\text{foo},.), \rightarrow q(_\text{bar},.)) \) then the sequence \( p(a, 1), q(2, a) \) is a valid execution, but \( p(a, 1), q(2, b) \) is not.

- \( \text{before}(p(T) \leftarrow q(\Pi)) \): 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 \( \langle s_1, \ldots, s_m \rangle \) is said to satisfy this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{q(\Pi)} s_{i+1} \) and \( q(\Pi) \) matches \( q(\Pi) \), there must be a state \( s_j \) in that same execution such that \( j \leq i - 1 \), \( s_j \xrightarrow{p(T)} s_{j+1} \), and \( \text{refine}(p(T); q(\Pi), q(\Pi)) \) matches \( p(T) \).

For instance, if the constraint is

\[ \text{before}(p(_\text{foo},.) \leftarrow q(_\text{bar},.)) \]

then the sequence \( p(a, 1), q(2, a) \) is a valid execution, but \( p(a, 1), q(2, b) \) is not.

\(^{13}\)The notation \( s \xrightarrow{p} s' \) was introduced in (2) in Section 3. Recall that since \( p(T_1), \ldots, p(T_n) \) cause state transitions, they are primitive update tasks.
3. Blocks: \( \text{blocks}(p(\overline{t}) \rightarrow q(\overline{u})): \) if task \( p \) executes then task \( q \) cannot execute in the future.

Formally, an execution \( \langle s_1, ..., s_m \rangle \) satisfies this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{p(\overline{t})} s_{i+1} \) and \( p(\overline{t}) \) matches \( p(\overline{T}) \), there is no state \( s_j \) in that execution such that \( j > i \), \( s_j \xrightarrow{q(\overline{u})} s_{j+1} \), and \( \text{refine}(q(\overline{u});p(\overline{U}),p(\overline{t})) \) matches \( q(\overline{U}) \).

- Between: \( \text{between}(p(\overline{t}) \rightarrow q(\overline{u}) \iff r(\overline{v})): \) 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 \( \langle s_1, ..., s_m \rangle \) satisfies this constraint if and only if whenever there are states \( s_k \), \( s_k \ (k > i + 1) \) such that \( s_i \xrightarrow{p(\overline{T})} s_{i+1} \), \( s_k \xrightarrow{r(\overline{V})} s_{k+1} \), \( p(\overline{t}) \) matches \( p(\overline{T}) \), and \( r(\overline{v}) \) matches \( r(\overline{V}) \), then there must be a state \( s_j \) such that \( i < j < k \), \( s_j \xrightarrow{q(\overline{u})} s_{j+1} \), and both \( \text{refine}(q(\overline{u});p(\overline{T}),p(\overline{t})) \) and \( \text{refine}(q(\overline{u});r(\overline{V}),r(\overline{v})) \) match \( q(\overline{U}) \).

- Not between: \( \text{not_between}(p(\overline{t}) \rightarrow q(\overline{u}) \iff r(\overline{v})): \) 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 \( \langle s_1, ..., s_m \rangle \) satisfies this constraint if and only if whenever there are states \( s_k \), \( s_k \ (k > i + 1) \) such that \( s_i \xrightarrow{p(\overline{T})} s_{i+1} \), \( s_k \xrightarrow{r(\overline{V})} s_{k+1} \), \( p(\overline{t}) \) matches \( p(\overline{T}) \), and \( r(\overline{v}) \) matches \( r(\overline{V}) \), then there is no state \( s_j \) such that \( i < j < k \), \( s_j \xrightarrow{q(\overline{u})} s_{j+1} \), and both \( \text{refine}(q(\overline{u});p(\overline{T}),p(\overline{t})) \) and \( \text{refine}(q(\overline{u});r(\overline{V}),r(\overline{v})) \) match \( q(\overline{U}) \).

3. Immediate serial constraints:

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

Formally, an execution \( \langle s_1, ..., s_m \rangle \) satisfies this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{p(\overline{T})} s_{i+1} \) and \( p(\overline{t}) \) matches \( p(\overline{T}) \), then \( s_{i+1} \xrightarrow{q(\overline{u})} s_{i+2} \) must hold and \( \text{refine}(q(\overline{u});p(\overline{T}),p(\overline{t})) \) must match \( q(\overline{U}) \).

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

Formally, an execution \( \langle s_1, ..., s_m \rangle \) satisfies this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{p(\overline{T})} s_{i+1} \) and \( q(\overline{u}) \) matches \( q(\overline{U}) \), then \( s_{i-1} \xrightarrow{p(\overline{T})} s_i \) must hold and \( \text{refine}(p(\overline{t});q(\overline{U}),q(\overline{u})) \) must match \( p(\overline{T}) \).

- Not right after: \( \text{not_right_after}(p(\overline{t}) \rightarrow q(\overline{u})): \) 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 \( \langle s_1, ..., s_m \rangle \) is said to satisfy this constraint if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{p(\overline{T})} s_{i+1} \), where \( p(\overline{t}) \) matches \( p(\overline{T}) \), and \( s_{i+1} \xrightarrow{r(\overline{U})} s_{i+2} \) for some \( r(\overline{v}) \), then \( \text{refine}(q(\overline{u});p(\overline{T}),p(\overline{t})) \) must not match \( r(\overline{U}) \).

4. Composite constraints: If \( C_1, C_2 \in \text{Constr} \) then so are \( C_1 \land C_2 \) (a conjunctive constraint) and \( C_1 \lor C_2 \) (a disjunctive constraint).

Nothing else is in \( \text{Constr} \).

\[ \square \]

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:

\[ \text{not_right_before}(p(\overline{t}) \leftarrow q(\overline{u})) \equiv \text{not_right_after}(p(\overline{t}) \rightarrow q(\overline{u})). \]
The following is an example of a legal constraint in $\text{Constr}$:

\[
\begin{align*}
\text{atleast}_2(p(_X, 1, _X)) & \land \text{exactly}_3(q(_-, _-)) \\
& \land \text{right}_-(p(_X, _-) \rightarrow r(_-, _X))
\end{align*}
\]

(7)

The constraint $\text{atleast}_2(p(_X, 1, _X))$ 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 the constraint $\text{right}_-(p(_X, _-) \rightarrow r(_-, _X))$, 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, as depicted in:

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

- $\text{atmost}_n(p(_-, 1))$ — 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.
- $\text{absence}(p(_-, 4)) \lor \text{atleast}_1(q(_-, 2, 3))$ — 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.
- $(\text{absence}(p()) \lor \text{atleast}_1(q())) \\
& \land (\text{absence}(q()) \lor \text{atleast}_1(p()))$ — if $p$ is executed, then $q$ must also be executed, and vice versa.
- $\text{after}(p(_X) \rightarrow q(_X)) \\
& \land \text{before}(p(_X) \leftarrow q(_X))$ — 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 $q$ before every occurrence of $p$ and their arguments must be the same.
- $\text{absence}(p()) \\
& \lor \text{between}(p(_X) \rightarrow q(_X) \leftarrow p(_))$ — if task $p$ is executed then $q$ must execute after it, with the same argument, and before that $q$ there can be no other $p$.
- $\text{absence}(q()) \lor (\text{before}(p() \leftarrow q(_)) \land \text{between}(q() \rightarrow p() \leftarrow q())$ — 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 of $p$.
- $\text{between}(p(_X) \rightarrow q(_X) \leftarrow p(_X)) \\
& \land \text{between}(q(_X) \rightarrow p(_X) \leftarrow q(_X))$ — tasks $p$ and $q$ must alternate when they execute with the same argument.
- $\text{right}_-(p() \rightarrow q()) \\
& \land \text{right}_-(p() \leftarrow q())$ — executions of $p$ and $q$ must be next to each other with no intervening tasks in-between.
- $\text{absence}(p()) \lor \text{absence}(q())$ — it is not possible for $p$ and $q$ to execute in the same service process run.
- $\text{not}_-(p(_X) \rightarrow q(_X) \leftarrow p(_X)) \land \text{not}_-(q(_X) \rightarrow p(_X) \leftarrow q(_X))$ — $q$ 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 $C$ satisfy the service contract assumption if and only if the set of constraints $C$ 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 com-
positional 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 a” mean exactly? Should a 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-iterative composite task of the form $p \leftarrow \Omega$ can be changed to:

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

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

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

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

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

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

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

An important benefit of the use of bounding start- and end-tasks is that they enable easy specification of pre- and post-conditions for any task in a service and even for the entire service. For instance, if $\text{serv}_{\text{start}}$ and $\text{serv}_{\text{end}}$ are bounding primitive tasks for the entire service and $\text{precond}$ is a query then $\text{right}_{\text{before}}(\text{precond} \leftarrow \text{serv}_{\text{start}})$ establishes $\text{precond}$ as a precondition for the entire service. Likewise, $\text{right}_{\text{after}}(\text{serv}_{\text{end}} \rightarrow \text{postcond})$ establishes $\text{postcond}$ as a post-condition for the service. A service can have several pre- and post-conditions and some of these constraints can belong to client requirements while others can be specified as part of the service policy.

**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 \( \text{token}(t, 0) \) on the first execution and then \( \text{token}(t, 1), \text{token}(t, 2), \text{etc.} \), 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 \( s \overset{p}{\rightarrow} s' \) and \( s \overset{q}{\rightarrow} s' \) hold. Suppose that we are now trying to execute \( p \) at state \( s \). In the presence of constraints such as \( \text{before}(r \leftarrow q) \), \( \text{absence}(q) \), 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:

\[
\alpha \otimes (B | C) \otimes (g \lor h)
\]  

Rules:

\[
B \leftarrow d \lor e \\
C \leftarrow (f \otimes C) \lor \text{state}
\]

Client contract requirements:

\[
\text{absence}(e) \land \text{atleast}_2(f)
\]  

Service policy:

\[
\text{after}(d \rightarrow f) \land \text{absence}(g)
\]

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 \( \text{after}(d \rightarrow f) \). 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 $((d \lor e) \land ((f \land C) \lor \text{state})) \land (g \lor \text{h})$ (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 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 $\text{absence}(e)$; 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 $\text{after}(d \rightarrow f)$ with $\text{atleast}_1(f)$. This is possible because after the execution of $d$ the constraint $\text{after}(d \rightarrow f)$ will be satisfied if $f$ is executed at some point in the future, whence $\text{atleast}_1(f)$. The system now checks whether an execution of the remaining process $((f \land C) \lor \text{state}) \land (g \lor \text{h})$ exists under the updated set of constraints $\text{absence}(e) \land \text{atleast}_2(f) \land \text{absence}(g)$ (since we have two constraints $\text{atleast}_2(f)$ and $\text{atleast}_1(f)$, the last one can be removed).

3. For the remainder of the process, $((f \land C) \lor \text{state}) \land (g \lor \text{h})$, only $f$ and state can be chosen. Let us assume that the internal action state is picked for execution. The system now checks whether an execution of the remaining part of the process, $(g \lor \text{h})$, exists given the updated set of constraints $\text{absence}(e) \land \text{atleast}_2(f) \land \text{absence}(g)$. It is easy to see that $(g \lor \text{h})$ cannot execute to satisfy $\text{atleast}_2(f)$ because there is no $f$ in $(g \lor \text{h})$. Therefore, 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 $\text{atleast}_2(f)$ with $\text{atleast}_1(f)$, since executing $f$ means that the number of required occurrences of $f$ decreases by 1. The remaining part of the process, $C \lor (g \lor \text{h})$ has a legal execution $\{f, h\}$ with respect to the updated set of constraints $\text{absence}(e) \land \text{atleast}_1(f) \land \text{absence}(g)$, so $f$ is accepted.

4. To proceed, we need to expand $C$ using the rule in (8), which gives us $((f \land C) \lor \text{state}) \land (g \lor \text{h})$ — the same process as in the previous step. Although the set of constraints has now changed, because of $\text{atleast}_1(f)$ 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 $\text{absence}(e) \land \text{absence}(g)$.

5. Now the remainder of the process is $(C \land \text{state}) \land (g \lor \text{h})$ and state can be successfully picked for execution and the remainder of the process becomes $g \lor \text{h}$. Either $g$ can now be attempted by the client or $h$ by the service. However, the constraint $\text{absence}(g)$ 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 $a, f, d, f, h$.

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 \( C \in \text{CONST} \) be a constraint (which can be composite), where \( \text{CONST} \) includes both the service policy and the client contract requirements. Let \( G \) be a service process and \( G \) and \( C \) 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 \( G \wedge C \). Formally, contracting aims to determine if there is a path on which \( G \wedge C \) 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 \( G \wedge C \) 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 \( G \wedge C \). Each such proof provides a way to execute the process without violating any of the constraints in \( C \).

The rest of this section develops a proof theory for formulas of the form \( G \wedge C \), where \( G \) is a service process and \( C \in \text{CONST} \).

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

\[ p \leftarrow q \otimes (r \lor s) \otimes t \]

can be eliminated by transforming this rule into

\[ p \leftarrow q \otimes \text{newpred} \otimes t \\ \text{newpred} \leftarrow r \\ \text{newpred} \leftarrow s \]

**Hot components.** We recall the notion of hot components of a CTR goal from [11]: \( \text{hot}(\psi) \) is a set of subformulas of \( \psi \) 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. \( \text{hot}(\epsilon) = \{\} \), where \( \epsilon \) is the empty goal
2. \( \text{hot}(\psi) = \{\psi\} \), if \( \psi \) is an atomic formula
3. \( \text{hot}(\psi_1 \otimes \ldots \otimes \psi_n) = \text{hot}(\psi_1) \)  
4. \( \text{hot}(\psi_1 | \ldots | \psi_n) = \text{hot}(\psi_1) \cup \ldots \cup \text{hot}(\psi_n) \)  
5. \( \text{hot}(\neg \psi) = \{\neg \psi\} \)

**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: \( \text{force}(p(T)) \), \( \text{suspend}(p(T)) \), and \( \text{next_right_before}(q(T') \leftarrow p(T) \leftarrow q(T)) \), 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 \( \text{force}(p(T)) \), where \( p(T) \) is a task pattern for a primitive update, is that the very next task to be executed must match \( p(T) \). More precisely, an execution \( (s_1, s_2, ..., s_n) \) satisfies \( \text{force}(p(T)) \) if \( s_1 \xrightarrow{p(T)} s_2 \) holds and \( p(T) \) matches \( p(T) \).

The constraint \( \text{suspend}(p(T)) \) means that no task that matches \( p(T) \) can execute at the current state. Formally, an execution \( (s_1, s_2, ..., s_n) \), such that \( s_1 \xrightarrow{r(V)} s_2 \) holds for some task atom \( r(V) \), satisfies \( \text{suspend}(p(T)) \) if \( r(V) \) does not match \( p(T) \).

The constraint \( \text{next_right_before}(q(T') \leftarrow p(T) \leftarrow q(T)) \) says that \( q(T') \) must be immediately preceded by \( p(T) \) unless it is the first task to be executed. That first task can be a \( q \)-task, if it matches \( q(T') \). Formally, an execution \( (s_1, s_2, ..., s_n) \) satisfies this constraint if and only if either \( (s_1, s_2, ..., s_n) \) satisfies \( \text{right_before}(p(T) \leftarrow q(T)) \) or \( s_1 \xrightarrow{q(T)} s_2 \) holds, where \( q(V) \) matches \( q(T') \), and \( (s_2, ..., s_n) \) satisfies \( \text{right_before}(p(T) \leftarrow q(T)) \).

The generalization of before has the following syntax:

\[
\text{before}(p(T) \leftarrow q(T) \setminus \{q(T_1), ..., q(T_m)\})
\]

where \( p(T), q(T), q(T_1), ..., q(T_m), n \geq 0 \) \((n = 0 \text{ means that the sets of exceptions are empty})\), are task patterns. The use of “\(-\) here indicates that \( \text{before}(p(T) \leftarrow q(T)) \) must hold, except for the tasks that match one of the exceptions \( q(T_1), ..., q(T_m) \).
Formally, an execution \( (s_1, \ldots, s_n) \) satisfies the constraint (11) if and only if whenever there is a state \( s_i \) in this execution such that \( s_i \xrightarrow{q(T)} s_{i+1} \) and \( q(U) \) matches \( q(\pi) \) but none of the \( q(\pi_j) \)'s then there is \( j, j < i \), such that \( s_j \xrightarrow{p(T)} s_{j+1} \) holds and \( p(T) \) matches \( \text{refine}(p(U); q(\pi), q(\pi_j)) \).

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 \( \sigma \) is a substitution and \( \psi \) is a service process or a term then we write \( \psi\sigma \) for the result of applying the substitution \( \sigma \) to \( \psi \). We call \( \psi\sigma \) an instance of \( \psi \). If \( \psi\sigma \) has no variables left, we say that \( \psi\sigma \) is a ground instance and that \( \sigma \) is a ground substitution.

**Sequents.** Let \( P \) be a set of composite task definitions. The proof theory manipulates expressions of the form \( P, s \rightarrow \psi \land C \), called sequents, where \( P \) is a set of task definitions, \( s \) is an identifier for the underlying database state, \( \psi \) is a CTR goal, and \( C \) is a (possibly composite) constraint, which may include the constraints in \( \text{CONSTRAINT} \) as well as the new constraints (\( \text{force, suspend, etc.} \)) introduced just above. Informally, a sequent is a statement that \( (\exists) \psi \land C \), which is defined by the rules in \( P \), can execute along some path that starts at state \( s \) so that all the constraints in \( C \) 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 \( C \) is a CTR tautology (patch). 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 \( c \in C \) meaning that \( c \) is a conjunct in \( C \).

**The notion of a proof.** A proof of a sequent \( s \rightarrow \psi \land C \) is a sequence of sequents, \( seq_0, seq_1, \ldots, seq_n-1, seq_n \), where \( seq_0 = s \rightarrow \psi \land C \) or each \( seq_j \) 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 \( P, s \rightarrow (\psi \land C) \), where \( s \) is a database state identifier and \( C \) is a conjunctive constraint that does not contain constraints of the form \( \text{force}(\pi), \text{atleast}_k(\pi), \text{exactly}_k(\pi) \), where \( k \geq 1 \).

**Inference Rules.** In Rules 1-7 below, \( \sigma \) denotes a substitution, \( \psi \) and \( \psi' \) are service processes, \( C \) and \( C' \) are constraints, \( s, s_1, s_2 \) are database state identifiers, and \( p \) is a task.

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

\[
\frac{P, s \rightarrow (\exists) \psi \land C'}{P, s \rightarrow (\exists) \psi \land C}
\]

Note that \( C' \) is a conjunction of existence and serial constraints.

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

\[
\frac{P, s \rightarrow (\exists) \chi \land C}{P, s \rightarrow (\exists) \chi \land C}
\]

3. **Commutativity with respect to builtin tests:** Let \( \chi \) be a conjunction of builtin test tasks and \( \psi \) a CTR goal. Then

\[
\frac{P, s \rightarrow (\exists) (\psi \land \chi) \land C}{P, s \rightarrow (\exists) (\psi \land \chi) \land C}
\]

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 \( \psi \). If \( p \) and \( r \) unify with the most general unifier \( \sigma \) then

\[
\frac{P, s \rightarrow (\exists) \psi' \land C}{P, s \rightarrow (\exists) \psi' \land C}
\]

where \( \psi' \) is obtained from \( \psi \) by replacing a hot occurrence of \( p \) with \( \beta \).
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) $p = \text{state}$ and $\sigma$ is the identity substitution. Then

$$
\frac{\mathbf{P}, s \longrightarrow (\exists) \psi'\sigma \land C}{\mathbf{P}, s \longrightarrow (\exists) \psi \land C}
$$

where $\psi'$ is obtained from $\psi$ by deleting a hot occurrence of $p$.

6. Executing primitive update tasks: Let $p\sigma$ be a primitive update task such that $s_1 \xrightarrow{p\sigma} s_2$, $p \in \text{hot}(\psi)$, and $C$ 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:

- absence($\bar{p}$)
- suspend($\bar{p}$)
- force($\bar{r}$) such that $p\sigma$ does not match $\bar{r}$
- before($\bar{r} \leftarrow \bar{p} \setminus \{\bar{p}_1, ..., \bar{p}_k\}$) $k \geq 0$ 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($\bar{r}' \leftarrow \bar{p} \leftarrow \bar{p}'$), where $p\sigma$ does not match $\bar{p}'$.

Then the inference rule has the following form:

$$
\frac{\mathbf{P}, s_2 \longrightarrow (\exists) \psi'\sigma \land C'}{\mathbf{P}, s_1 \longrightarrow (\exists) \psi \land C'}
$$

where $C$ is a conjunctive constraint, $\psi'$ is obtained from $\psi$ by deleting the hot component $p$, and $C'$ is constructed out of $C$ as follows.

Initially, $C'$ is empty (a tautology, 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 $\text{atleast}_n(\bar{p}) \in C$, where $n > 1$, add the following to $C'$:

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

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

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

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

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

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

- $\text{after}(\bar{p} \rightarrow \bar{r})$
- $\text{atleast}_{1}(\bar{r}')$
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})$

(e) If $\text{blocks}(\bar{p} \dashv \bar{r}) \in C$, add the following to $C'$:

- $\text{blocks}(\bar{p} \dashv \bar{r})$
- absence($\bar{r}'$)
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})$

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

- right_after($\bar{p} \rightarrow \bar{r})$
- force($\bar{r}'$)
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})$

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

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

(h) If $\text{not_right_after}(\bar{p} \rightarrow \bar{r}) \in C$, add the following to $C'$:

- not_right_after($\bar{p} \rightarrow \bar{r})$
- suspend($\bar{r}'$)
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})$

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

- right_before($\bar{p}' \leftarrow \bar{p} \leftarrow \bar{r}$), where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})$

(j) If $\text{next_right_before}(\bar{r}' \leftarrow \bar{p} \leftarrow \bar{s}) \in C$, add the following to $C'$:

- $\text{next_right_before}(\bar{r}' \leftarrow \bar{p} \leftarrow \bar{s})$
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p}))$

(k) If $\text{between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s}) \in C$, add the following to $C'$:

- $\text{between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s})$
- $\text{before}(\bar{r}' \leftarrow \bar{s}')$
  
  where $\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p}))$ and $\bar{s}' = \text{refine}(\bar{s}; p\sigma, \bar{p})$

(l) If $\text{not_between}(\bar{p} \dashv \bar{r} \dashv \bar{s}) \in C$, add the following to $C'$:

- $\text{not_between}(\bar{p} \dashv \bar{r} \dashv \bar{s})$
the execution of a primitive task with the argument \( p \) and in the case of a database, as follows:

\[
P,\{\} \vdash (\exists ?x ((p(?x) \land q) \mid r(?x)))
\land \text{atleast}_2(s(\_1,\_1))
\land \text{before}(p(\_x) \leftarrow s(\_x, 1, \_))
\]

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:

\[
P,\{\} \vdash (\exists ?x, ?y, ?z (p(?x) \land q)
\land (xyz(?y, ?z) \land s(?x, ?y, ?z) \mid r(?y)))
\land \text{atleast}_2(s(\_1,\_1))
\]

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

\[
P,\{\} \vdash (\exists ?x, ?y, ?z q
\land (xyz(?y, ?z) \land s(1, ?y, ?z)
\mid r(?y)))
\land \text{atleast}_2(s(\_1,\_1))
\]

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

\[
P,\{\} \vdash q \mid (s(1, 3, 7) \land r(3))
\land \text{atleast}_2(s(\_1,\_1))
\]

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:

\[
P,\{\} \vdash (q \mid r(3))
\land \text{atleast}_1(s(\_1,\_1))
\]
Hot components: \{q, r(3)\}. By inference rule 6 applied to the primitive task \( q \) (which, as mentioned above, deletes \( xyz(3, 7) \)):

\[
P, \{xyz(2, 7)\} \rightarrow \neg r(3) \land \text{atleast}_1(s(\_,\_,\_))
\]

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

\[
P, \{xyz(2, 7)\} \rightarrow (\exists \ y, \ z \ (xyz(\ y, \ z) \\
\quad \land s(3, \ y, \ z) \land r(\ y)))
\]

\[
\land \text{atleast}_1(s(\_,\_,\_))
\]

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

\[
P, \{xyz(2, 7)\} \rightarrow (s(3, 2, 7) \land r(2))
\]

\[
\land \text{atleast}_1(s(\_,\_,\_))
\]

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:

\[
P, \{xyz(2, 7), abc(1)\} \rightarrow r(2)
\]

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

\[
P, \{xyz(2, 7), abc(1)\} \rightarrow \text{state}
\]

Hot components: \{\text{state}\}. By rule 5 applied to \text{state} we derive:

\[
P, \{xyz(2, 7), abc(1)\} \rightarrow (\_)
\]

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 \( C \), which required \( s(\_,\_,\_) \) to be executed at least twice.

5.3. 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 \text{ServLog}. In [8], various subsets of CTR were investigated for their decidability and complexity properties. One important restriction studied there is called full boundedness. In terms of \text{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 [8] (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 \text{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 [31,23]. 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 \text{ServLog}—see the discussion of artifact systems [16] in Section 6.

6. Related Work

\text{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 $\land$ connective thus enabling execution of constrained transactions, which are non-Horn. We also deal with a much larger class of constraints than [17,38], 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 $\text{Constr}$ 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, [34] 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 [29, 19,16]. 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:

\[
\text{serv}(?X) \leftarrow \text{termination\_condition}(?X) \\
\text{serv}(?X) \leftarrow \text{precond}_1(?X) \land \text{task}_1(?X, ?Y) \land \text{postcond}_1(?X, ?Y) \land \text{serv}(?Y) \\
\text{serv}(?X) \leftarrow \text{precond}_2(?X) \land \text{task}_2(?X, ?Y) \land \text{postcond}_2(?X, ?Y) \land \text{serv}(?Y) \\
\ldots
\]

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 [19] carry over from artifact systems to the special case of the ServLog service processes described above and thus give us a decidable...
subset of Transaction Logic that is different from the one mentioned in Section 5.3.

An emerging area related to our work is that of compliance checking between business processes and business contracts. For example, in [26,25] 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 [5,2,3,14], 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, [18]). Moreover, as shown in [17], 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 [9,10,11,37,6].

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)\textsuperscript{14} 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., [15] 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 [21]. 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

\textsuperscript{14}http://sourceforge.net/projects/gecop/
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. [30]) 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\(^{15}\) in the context of cryptocurrencies.\(^{16}\)

Here, ServLog could prove to be a useful approach for modeling and reasoning about distributed cryptocurrency contracts.\(^{17}\)

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.

References


---

\(^{15}\)For a relevant related idea, see [4], where the authors proposed timed automata for formalizing Bitcoin contracts.

\(^{16}\)See for example Bitcoin contracts at https://en.bitcoin.it/wiki/Contracts.

\(^{17}\)http://szabo.best.vwh.net/smart_contracts_idea.html


Appendix

A. Soundness of the Inference System

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

$$\text{If } P, s \longrightarrow \psi \land C \text{ then } P, s \longrightarrow \psi \land C$$

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 $\sigma r$ be a primitive update task such that $s_1 \xrightarrow{\sigma r} s_2$ holds, $\rho \in \text{hot}(\psi)$, and $C$ has no constraint of either of the forms below. We assume that $\tilde{p}$ is a task pattern such that $\sigma r$ matches $\tilde{p}$ and that $\tilde{t}$ denotes an arbitrary task pattern:

$$- \text{ absence}(\tilde{p})$$
- suspend(\(\bar{p}\))
- force(\(\bar{r}\)) such that \(p\sigma\) does not match \(\bar{r}\)
- before(\(\bar{r} \leftarrow \bar{p} \setminus \{\bar{p}_1, ..., \bar{p}_k\}\)) \(k \geq 0\) 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(\(\bar{r}' \leftarrow \bar{p}\)), where \(p\sigma\) does not match \(\bar{p}'\).

Then the inference rule has the following form:

\[
\frac{P, s_2 \ldots \vdash (\exists) \psi' \sigma \land C'}{P, s_1 \ldots \vdash (\exists) \psi \land \overline{\sigma} \land C'}
\]

where \(\psi'\) is obtained from \(\psi\) by deleting the hot component \(p\) and \(C'\) is constructed out of \(C\) as follows.

Initially, \(C'\) 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 \(\text{atleast}_n(\bar{p}) \in C\), where \(n > 1\), add the following to \(C'\):
- \(\text{atleast}_{n-1}(\bar{p})\);

(b) If \(\text{exactly}_n(\bar{p}) \in C\), where \(n > 1\), add the following to \(C'\):
- \(\text{exactly}_{n-1}(\bar{p})\);

(c) If \(\text{exactly}_1(\bar{p}) \in C\), add the following to \(C'\):
- \(\text{absence}(\bar{p})\);

(d) If \(\text{after}(\bar{p} \Rightarrow \bar{r}) \in C\), add the following to \(C'\):
- \(\text{after}(\bar{p} \Rightarrow \bar{r})\)
- \(\text{atleast}_1(\bar{r}')\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\)

(e) If \(\text{blocks}(\bar{p} \Rightarrow \bar{r}) \in C\), add the following to \(C'\):
- \(\text{blocks}(\bar{p} \Rightarrow \bar{r})\)
- \(\text{absence}(\bar{r}')\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\)

(f) If \(\text{right_after}(\bar{p} \rightarrow \bar{r}) \in C\), add the following to \(C'\):
- \(\text{right_after}(\bar{p} \rightarrow \bar{r})\)
- \(\text{force}(\bar{r}')\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\).

(g) If \(\text{before}(\bar{p} \leftarrow \bar{r} \setminus \{\bar{r}_1, ..., \bar{r}_k\}) \in C\), add the following to \(C'\):
- \(\text{before}(\bar{p} \leftarrow \bar{r} \setminus \{\bar{r}', \bar{r}_1, ..., \bar{r}_k\})\) where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\);

(h) If \(\text{not_right_after}(\bar{p} \rightarrow \bar{r}) \in C\), add the following to \(C'\):
- \(\text{not_right_after}(\bar{p} \rightarrow \bar{r})\)
- \(\text{suspend}(\bar{r}')\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\)

(i) If \(\text{right_before}(\bar{p} \leftarrow \bar{r}) \in C\), add the following to \(C'\):
- \(\text{next_right_before}(\bar{r}' \leftarrow \bar{p})\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\);

(j) If \(\text{next_right_before}(\bar{r}' \leftarrow \bar{s}) \in C\), add the following to \(C'\):
- \(\text{right_before}(\bar{s} \leftarrow \bar{r})\), where \(\bar{s}, \bar{r}\) are arbitrary task patterns.

(k) If \(\text{between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s}) \in C\), add the following to \(C'\):
- \(\text{between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s})\)
- \(\text{blocks}(\bar{r}' \rightarrow \bar{s}')\), where \(\bar{r}' = \text{refine}(\bar{r}; p\sigma, \bar{p})\)
and \(\bar{s}' = \text{refine}(\bar{s}; p\sigma, \bar{p})\)

(l) If \(\text{not_between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s}) \in C\), add the following to \(C'\):
- \(\text{not_between}(\bar{p} \rightarrow \bar{r} \leftarrow \bar{s})\)
- \(\text{force}(\bar{p})\)
- \(\text{suspend}(\bar{r})\), for any task pattern \(\bar{r}\)

\textbf{Proof:} Suppose \(P, s_2 \ldots s_n \vdash (\exists) \psi' \sigma \land C'\). By soundness of the CTR proof theory, it follows that \(P, s_1 \ldots s_n \vdash (\exists) \psi\). So, it remains to prove that \(P, s_1 \ldots s_n \vdash (\exists) \psi \land C\) and, as a special case, that:

\[
P, (s_1, ..., s_n) \vdash C
\]

(12)

In the proof, we will rely on the premise

\[
P, (s_2, ..., s_n) \vdash C'
\]

(13)

which is a special case of the assumption \(P, s_2 \ldots s_n \vdash (\exists) \psi' \sigma \land C'\).

\textbf{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 \(C\) to \(C'\) via Case (m), we need to show that it holds over the execution \(\{s_1 ... s_n\}\).
Case (a): In this case, \( C \) has at least \( n(p) \) and \( C' \) has at least \( n-1(p) \). By (13), it follows that \( P, s_2 \ldots s_n \models \text{at least}_n(p) \).

If \( P, s_3 \ldots s_n \models \text{at least}_n(p) \) then since \( s_1 \rho \sigma \), \( s_2 \) and \( \rho \sigma \) matches \( p \), it follows that \( P, s_1 \ldots s_n \models \text{at least}_n(p) \) holds.

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

Case (c): In this instance, \( C' \) contains \( \text{absence}(p) \), so \( s_i \overset{r}{\longrightarrow} s_{i+1}, 1 < i < n \), is not possible for any \( r \) that matches \( p \). Therefore, \( \langle s_1, \ldots, s_n \rangle \) satisfies exactly_1(p), i.e., \( P, s_1 \ldots s_n \models \text{exactly}_1(p) \) holds.

Case (d): In this case, \( C' \) contains \( \text{after}(p \rightarrow r) \) and at least_1(r'). and \( C \) contains \( \text{after}(p \rightarrow r) \). By (13), it follows that \( P, s_2 \ldots s_n \models \text{at least}_1(r') \).

Suppose now that after \( \rho \sigma \) does not hold on \( \langle s_1, s_2, \ldots, s_n \rangle \). Since the transition from \( s_1 \) to \( s_2 \) can be made only by \( \rho \sigma \) (by the primitive update task independence assumption), it follows that \( \text{absence}(r') \) must hold on \( \langle s_2, \ldots, s_n \rangle \). But this contradicts the fact that \( \text{at least}_1(r') \) must hold on the same path. Thus, \( \text{after}(p \rightarrow r) \) must hold on \( \langle s_1, \ldots, s_n \rangle \), i.e., \( P, s_1 \ldots s_n \models \text{after}(p \rightarrow r) \).

Case (e): Here \( C' \) contains \( \text{blocks}(p \rightarrow r) \) and \( \text{absence}(r') \). and \( C \) contains \( \text{blocks}(p \rightarrow r) \).

By (13), \( P, s_2 \ldots s_n \models \text{absence}(r') \). Suppose that \( \text{blocks}(p \rightarrow r) \) does not hold on \( \langle s_1, s_2, \ldots, s_n \rangle \). Since the transition from \( s_1 \) to \( s_2 \) can be made only by \( \rho \sigma \) (by the primitive update task independence assumption), it follows that \( \text{at least}_1(r') \) must hold on \( \langle s_2, \ldots, s_n \rangle \). But this contradicts \( P, s_2 \ldots s_n \models \text{blocks}(p \rightarrow r) \). Thus \( P, s_1 \ldots s_n \models \text{blocks}(p \rightarrow r) \).

Case (f): Here, we have that \( C' \) contains both \( \text{right after}(p \rightarrow r) \) and \( \text{force}(r') \), while \( C \) contains \( \text{right after}(p \rightarrow r) \).

By (13), it follows that \( P, s_2 \ldots s_n \models \text{force}(r') \), and thus \( s_2 \overset{r}{\rightarrow} s_3 \) must hold for some \( r' \) that matches \( r \), a refinement pattern of \( r \). Since \( s_1 \overset{p}{\rightarrow} s_2 \) and \( \rho \sigma \) matches \( p \), it follows that \( P, s_1 \ldots s_n \models \text{right after}(p \rightarrow r) \).

Case (g): In this instance, we have that \( C' \) contains \( \text{before}(p \leftarrow r \setminus \{r, r_1, \ldots, r_k\}) \) while \( C \) contains \( \text{before}(p \leftarrow r \setminus \{r, r_1, \ldots, r_k\}) \). By (13), it follows that \( P, s_2 \ldots s_n \models \text{before}(p \leftarrow r \setminus \{r, r_1, \ldots, r_k\}) \).

Since \( s_1 \overset{p}{\rightarrow} s_2 \) holds, \( \rho \sigma \) matches \( p \), and \( r' \) \( = \text{refine}(R; \rho \sigma, p) \), it follows that \( \text{before}(p \leftarrow r') \) (and, as a special case, before \( p \leftarrow r' \)) is satisfied by \( \langle s_1, s_2, \ldots, s_n \rangle \). Since \( \langle s_2, \ldots, s_n \rangle \) satisfies \( \text{before}(p \leftarrow r \setminus \{r, r_1, \ldots, r_k\}) \), we conclude that it also satisfies \( \text{before}(p \leftarrow r \setminus \{r_1, \ldots, r_k\}) \). But the transition from \( s_1 \) to \( s_2 \) was made by \( \rho \sigma \), which matches \( p \), so it holds that \( \langle s_1, s_2, \ldots, s_n \rangle \) cannot violate \( \text{before}(p \leftarrow r \setminus \{r_1, \ldots, r_k\}) \).

Case (h): In this case, we have that \( C' \) contains \( \text{not right after}(p \rightarrow r) \) and \( \text{suspend}(r') \), and \( C \) contains \( \text{not right after}(p \rightarrow r) \). By (13), it follows that \( P, s_2 \ldots s_n \models \text{suspend}(r') \) and \( \text{not right after}(p \rightarrow r) \). Suppose, we have that \( \text{not right after}(p \rightarrow r) \) does not hold on \( \langle s_1, s_2, \ldots, s_n \rangle \). Since the transition from \( s_1 \) to \( s_2 \) can be made only by \( \rho \sigma \) (by the primitive update task independence assumption), it follows that the only way to violate \( \text{not right after}(p \rightarrow r) \) over \( \langle s_1, s_2, \ldots, s_n \rangle \) is to cause a transition \( s_2 \overset{r'}{\rightarrow} s_3 \) using some \( r' \) that matches \( r' \) \( = \text{refine}(R; \rho \sigma, p) \). But this contradicts the assumption that \( P, s_2 \ldots s_n \models \text{suspend}(r') \). Therefore, it must be the case that \( P, s_1 \ldots s_n \models \text{not right after}(p \rightarrow r) \).

Case (i): In this case, \( \text{next right before}(r' \mapsto p \leftarrow r) \in C' \), where \( r' \mapsto \text{refine}(R; \rho \sigma, p) \), and \( \text{right before}(p \leftarrow r) \in C \). Since the path \( \langle s_2, \ldots, s_n \rangle \) satisfies the constraint \( \text{next right before}(r' \mapsto p \leftarrow r) \), we have that either (i) \( s_2 \overset{p}{\rightarrow} s_3 \) holds and \( \langle s_2, \ldots, s_n \rangle \) satisfies right before \( p \leftarrow r \); or (ii) \( s_2, \ldots, s_n \rangle \) satisfies right before \( p \leftarrow r \). In either case, since \( s_1 \overset{p}{\rightarrow} s_2 \) holds by the assumption, it follows that \( \text{right before}(p \leftarrow r) \) holds on the entire execution path \( \langle s_1, \ldots, s_n \rangle \).

Case (j): Here \( \text{right before}(s \leftarrow r) \in C' \) and \( \text{next right before}(r' \mapsto s \leftarrow r) \in C \), where \( s, r \) are arbitrary task patterns. We thus have that \( P, \langle s, s \ldots, s_n \rangle \models \text{right before}(s \leftarrow r) \) and \( s_1 \overset{p}{\rightarrow} s_2 \) both hold. By the premise of rule 6, either

- \( \rho \sigma \) does not match \( r' \); or
- \( \rho \sigma \) matches \( r' \).

Since the constraint \( \text{right before}(s \leftarrow r) \) is satisfied by \( \langle s, s \ldots, s_n \rangle \), in either of these cases the constraint \( \text{next right before}(r' \mapsto s \leftarrow r) \) holds on \( \langle s_1, \ldots, s_n \rangle \).

Case (k): In this case, we have that \( C' \) contains \( \text{between}(p \rightarrow r \mapsto s) \) and \( \text{before}(r' \leftarrow s') \), where \( r' \mapsto \text{refine}(R; \rho \sigma, p) \) and \( s' \mapsto \text{refine}(S; \rho \sigma, p) \), while \( C \) contains \( \text{between}(p \rightarrow r \mapsto s) \) instead. In addition, \( s_1 \overset{p}{\rightarrow} s_2 \) holds. Since \( \langle s_2, \ldots, s_n \rangle \) satisfies both \( \text{before}(r' \leftarrow s') \) and \( \text{between}(p \rightarrow r \mapsto s) \), it follows that \( \langle s_1, s_2, \ldots, s_n \rangle \) satisfies the constraint \( \text{between}(\rho \sigma \rightarrow r' \leftarrow s') \). Therefore, it must be that \( \text{between}(p \rightarrow r \mapsto s) \) cannot be violated by \( \langle s_1, s_2, \ldots, s_n \rangle \).
Case (l): In this case, we have that $C'$ contains not_between($\bar{p}$, $\bar{s}$) and blocks($\bar{r}$, $\bar{s'}$), while $C$ contains not_between($\bar{p}$, $\bar{r}$). In addition, $s_1 \overset{p}{\rightarrow} s_2$ holds. Since $(s_2, ..., s_n)$ satisfies blocks($\bar{r}$, $\bar{s'}$) and not_between($\bar{p}$, $\bar{r}$), it follows that $(s_1, s_2, ..., s_n)$ satisfies the constraint not_between($p \rightarrow r$, $\bar{s'}$). Therefore, we have that not_between($\bar{p}$, $\bar{r}$) cannot be violated by $(s_1, s_2, ..., s_n)$.

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

- at least 1($\bar{p}$): In this case, $C$ has at least 1($\bar{p}$) while $C'$ does not. Recall that the initial transition was $s_1 \overset{p}{\rightarrow} s_2$ and $p \sigma$ matches $\bar{p}$. Therefore, $P, s_1 ... s_n \models$ at least 1($\bar{p}$) holds.

- force($\bar{p}$): In this case $C$ has force($\bar{p}$) while $C'$ does not. Since the initial transition was $s_1 \overset{p}{\rightarrow} s_2$ and $p \sigma$ matches $\bar{p}$, we conclude that $P, s_1 ... s_n \models$ force($\bar{p}$) holds. (Note that the case of force($\bar{r}$) $\in C$, where $p \sigma$ does not match $\bar{r}$ is precluded by the precondition to rule 6, so we are not missing cases by considering force($\bar{p}$) only.)

- suspend($\bar{r}$), for some task pattern $\bar{r}$: Here $C$ has suspend($\bar{r}$), but $C'$ does not. Suppose $P, s_1 ... s_n \models$ suspend($\bar{r}$) does not hold. This means that the initial transition from $s_1$ to $s_2$ was caused by a task that matches $\bar{r}$. At the same time, the initial transition was $s_1 \overset{p}{\rightarrow} s_2$, 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 suspend constraint in $C$, including $\bar{r}$. Thus $P, s_1 ... s_n \models$ suspend($\bar{r}$) holds.  

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

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 $C$ 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 $C$ contains:

- absence($\bar{p}$)
- force($\bar{r}$) such that $p \sigma$ does not match $\bar{r}$
- before($\bar{r} \leftarrow \bar{p}$ $\setminus \{\bar{p}_1, ..., \bar{p}_k\}$) $k \geq 0$ and $p \sigma$ does not match any of the $\bar{p}$’s
- right_before($\bar{r} \leftarrow \bar{p}$)
- next_right_before($\bar{p} \leftarrow \bar{r} \leftarrow \bar{p}$), 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 absence($\bar{p}$), or before($\bar{r} \leftarrow \bar{p}$ $\setminus \{\bar{p}_1, ..., \bar{p}_k\}$), right_before($\bar{r} \leftarrow \bar{p}$), or force($\bar{r}$), or next_right_before($\bar{r} \leftarrow \bar{p}$) are not satisfied on any path of the form $(s_1, s_2, ..., s_n)$. 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 $P, s \models (\psi \wedge C)$ but cannot declare success because the axiom does not apply due to the fact that $C$ contains a constraint of the form $\text{force}(\bar{r})$, $\text{atleast}_k(\bar{r})$, or $\text{exactly}_k(\bar{r})$, where $k \geq 1$. But then, if the eliminated execution path satisfies the original constraint $C$ then, by soundness of the inference rules, the path $(s)$ would have to satisfy $C$ and thus also one of the aforementioned constraints: $\text{force}$, $\text{atleast}$, or $\text{exactly}$, 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.

B. Completeness of the Inference System

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

$$P, s \models \rightarrow (\psi \wedge C) \quad \text{implies} \quad P, s \models \rightarrow (\psi \wedge C)$$

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

- The original CTR proof theory is complete.
C. Representing Constraints of ServLog as CTR Formulas

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

The followings are CTR representations for the constraints in 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 \textit{vars}(\overline{t}) to represent the set of variables appearing in \overline{t},\textsuperscript{18} and \(\nabla p(\overline{t})\) denotes\textsuperscript{19}

\[
\text{path} \otimes p(\overline{t}) \otimes \text{path}
\]  

(14)

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

- \texttt{atleast}_n(p(\overline{t})):  
  \[  
  \exists \text{vars}(u^n_{i=1}t_i) \ (\nabla p(t_1) \otimes ... \otimes \nabla p(t_n))  
  \]  
  (15)

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

- \texttt{absence}(p(\overline{t})):  
  \[  
  \forall \text{vars}(\overline{t}) \ldots \nabla p(\overline{t})  
  \]  
  (16)

- \texttt{exactly}_n(p(\overline{t})):  
  \[  
  \text{atleast}_n(p(\overline{t})) \land \neg \text{atleast}_{n+1}(p(\overline{t}))  
  \]  
  (17)

- \texttt{after}(p(\overline{t}) \rightarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}) \ \exists \text{vars}(\overline{p}) \land \neg \text{vars}(\overline{t})  
  \text{path} \otimes p(\overline{t}) \Rightarrow \nabla q(\overline{p})  
  \]  
  (18)

Here \text{vars}(\overline{p}) \land \neg \text{vars}(\overline{t}) is the set of variables in \(\overline{p}\) minus those that appear in \(\overline{t}\).

- \texttt{before}(p(\overline{t}) \leftarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{p}) \ \exists \text{vars}(\overline{t}) \land \neg \text{vars}(\overline{p})  
  \nabla p(\overline{t}) \leftarrow q(\overline{p}) \otimes \text{path}  
  \]  
  (19)

\texttt{before}(p(\overline{t}) \leftarrow q(\overline{p}) \setminus \{q(\overline{v}_1), ..., q(\overline{v}_n)\})  

\[  
\forall \text{vars}(\overline{p}) \forall \text{vars}(u_{i=1}n) \ \exists \text{vars}(\overline{t}) \land \neg \text{vars}(\overline{p})  
\forall_{i=1}^n t_i = p_i \Rightarrow \nabla q(\overline{p}) \otimes \text{path}  
\]  

(20)

- \texttt{blocks}(p(\overline{t}) \rightarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}, \overline{p}) \text{path} \otimes p(\overline{t}) \Rightarrow \nabla q(\overline{p})  
  \]  
  (21)

- \texttt{between}(p(\overline{t}) \rightarrow q(\overline{p}) \leftrightarrow r(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}, \overline{p}) \ \exists \text{vars}(\overline{p}) \land \neg \text{vars}(\overline{t})  
  \text{path} \otimes p(\overline{t}) \Rightarrow \nabla q(\overline{p}) \leftrightarrow r(\overline{p}) \otimes \text{path}  
  \]  
  (22)

- \texttt{not_between}(p(\overline{t}) \rightarrow q(\overline{p}) \leftrightarrow r(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}, \overline{p}, \overline{r}) \text{path} \otimes p(\overline{t}) \Rightarrow \nabla q(\overline{p}) \leftrightarrow r(\overline{p}) \otimes \text{path}  
  \]  
  (23)

- \texttt{right_after}(p(\overline{t}) \rightarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}) \ \exists \text{vars}(\overline{p}) \land \neg \text{vars}(\overline{t})  
  \text{path} \otimes p(\overline{t}) \Rightarrow q(\overline{p}) \otimes \text{path}  
  \]  
  (24)

- \texttt{right_before}(p(\overline{t}) \leftarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{p}) \ \exists \text{vars}(\overline{t}) \land \neg \text{vars}(\overline{p})  
  \text{path} \otimes p(\overline{t}) \leftarrow q(\overline{p}) \otimes \text{path}  
  \]  
  (25)

- \texttt{next_right_before}(q(\overline{p}) \leftarrow p(\overline{t}) \leftarrow q(\overline{p})):  
  \[  
  \text{right_before}(p(\overline{t}) \leftarrow q(\overline{p})) \lor \text{right_before}(p(\overline{t}) \leftarrow q(\overline{p}))  
  \]  
  (26)

- \texttt{not_right_after}(p(\overline{t}) \rightarrow q(\overline{p})):  
  \[  
  \forall \text{vars}(\overline{t}, \overline{p}) \text{path} \otimes p(\overline{t}) \Rightarrow (\text{arc} \land \nabla q(\overline{p}) \otimes \text{path})  
  \]  
  (27)

where \texttt{arc} is a CTR proposition such that \(s_1 \rightarrow_s \texttt{arc} \rightarrow_s s_2\) is true for any pair of states \(s_1, s_2\) and \texttt{arc} is not true on any other path.

\textsuperscript{18}Note that all underscores (unnamed placeholders) are replaced with new variables, e.g. \texttt{vars}(p(_{\ldots})) = \{X, Y\}.

\textsuperscript{19}Informally, \(\nabla p(\overline{t})\) means that \(p(\overline{t})\) eventually executes.
- force\( (p(\overline{t})) \):

\[ \exists \ text{vars}(\overline{t}) \ p(\overline{t}) \otimes \text{path} \quad (28) \]

- suspend\( (p(\overline{t})) \):

\[ \forall \ text{vars}(\overline{t}) \ (\text{arc} \land \neg p(\overline{t})) \otimes \text{path} \quad (29) \]

- **Composite constraints:** If \( C_1, C_2 \in \text{CONST} \) then so are \( C_1 \land C_2 \) (a conjunctive constraint) and \( C_1 \lor C_2 \) (a disjunctive constraint).