A Formal Model for Weakly-structured Scientific Workflows

Zhili Zhao and Adrian Paschke

Institut for Computer Science, Freie Universität Berlin,
Königin-Luise-Str. 24/26, 14195 Berlin, Germany
{zhili,paschke}@inf.fu-berlin.de

Abstract. This paper gives a Concurrent Transaction Logic ($\mathcal{CTR}$)-based formal model for Weakly-structured Scientific Workflows (WsSWFs) and further implements it in an open source rule language Prova. Compared with the related efforts, the formal model in this paper focuses on conversation-based reactive workflow logic representation and event-driven computation of complex event patterns.

Keywords: Weakly-structured Process, Reaction Rule, Formal Model

1 Introduction

The current focus of existing solutions for scientific workflows is on the orchestrated and pre-structured execution of computational intensive and data-oriented tasks, instead of the goal-oriented and decision-centric tasks that need coordination of scientists or computer agents (aka. expert system agents) as a team supported by semi-automated Weakly-structured scientific workflows (WsSWFs).

Such WsSWFs contain tasks that are goal-oriented and that need to be executed on the basis of dynamic decisions which the agents can take in order to adapt to unforeseen uncertainties in the execution and to dynamically changed (scientific) knowledge about the scientific problems [10]. In our previous work [11, 12], we provided a declarative rule-based workflow language and implemented a distributed execution middleware that employs inference agents as execution environments. We proposed an event-driven framework, which considers data passing between agents as "events" and executes scientific processes in terms of event message-driven conversations between distributed agents [11]. Following the spirit of Subject-oriented Business Process Management (S-BPM), we proposed an agent-oriented approach to model the WsSWFs based on the event-driven framework, where each agent has an internal behavior and coordinates with other agents to complete certain goals [12].

For the purpose of reducing ambiguity and opening possibilities for verification and analysis, this paper gives a Concurrent Transaction Logic ($\mathcal{CTR}$)-based formal declarative model for the WsSWFs. Workflow formal models provide a theoretical foundation to workflow programming languages and can be used to

support the design of workflow languages and of their interpreters, compilers and optimizers as well as of debuggers, and to support the definition of verification procedures, similar to those used for verifying the correctness of complex business transactions [9].

The remainder of this paper is organized as follows: Section 2 presents a WsSWF use case. Section 3 gives a brief overview of $\mathcal{CTR}$. Section 4 explains how $\mathcal{CTR}$ is used as an underlying formalism to represent the WsSWFs. Section 5 demonstrates how the $\mathcal{CTR}$-based workflow logic can be translated into a Web rule language Prova. Section 6 introduces the related efforts and the conclusion is given in Section 7.

2 Use Case

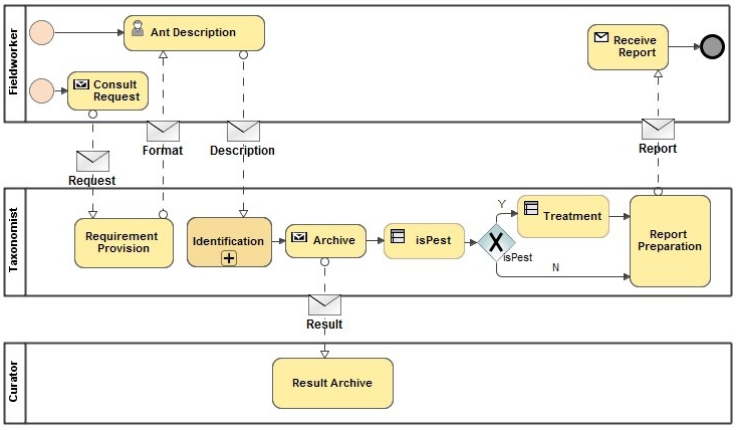

Fig. 1. Process of ant identification and treatment

This section gives a WsSWF use case taken from the EDIT (European Distributed Institute of Taxonomy)[1] to identify and treat newly discovered ants. We adapt it from [11] and make it more sophisticated for the purpose of demonstrating a logical WsSWF representation based on $\mathcal{CTR}$. The whole process is shown in Figure 1.

The process involves the collaboration of three participants: *fieldworker*, *taxonomist* and *curator*. A fieldworker who often works in the countryside firstly triggers the identification process by describing a newly discovered ant, then sends the description to a taxonomist, who has experience and expertise to perform the identification and know how to treat it. The fieldworker may consult the requirements for ant description if necessary. The identification results are

[1] http://www.e-taxonomy.eu/

usually ant scientific names. After the identification, the taxonomist sends the results to a curator who is in charge of managing the identification results. In order to automate this process, we employ agents to simulate these participants and their interaction. The intelligent agents act on their behalf by using a local knowledge base of reaction and derivation rules.

Ants differ widely in their food requirements and behaviors, some pests even can cause a serious impact on crops. In case of a pest, the corresponding treatment schemes are also needed to provide to the fieldworker. The identification task involves complicated decision logic to distinguish an ant from its homogeneous groups and is represented as a sub-process (with a "+" mark in the notation). The details of the identification process are shown in Figure 2. During the identification, there are two steps to assign the identification task to an agent that has the appropriate knowledge base: (1) narrowing down the scope of agents that can perform the identification based on the location of ant discovery; (2) finding an agent that has the best performance (e.g., success rate) in that area. Afterwards, the ant scientific name is determined in terms of a knowledge base consisting of domain rules and facts. Likewise, the tasks of identification assignments and getting pest treatment are also encoded with domain rules and facts. These knowledge-intensive decision-centric tasks are represented as rounded rectangles with small table notations in them. Moreover, if a discovered ant is extraordinary, it might involve domain experts to identify it manually.

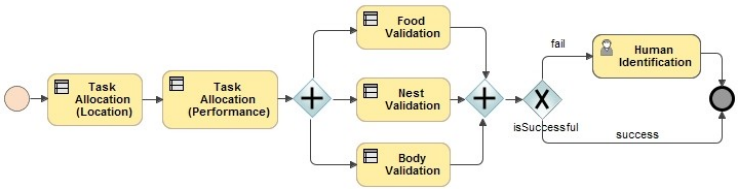

Fig. 2. Sub-process of ant identification

3 Overview of Concurrent Transaction Logic

Transaction Logic ($\mathcal{TR}$) is a general logic that accounts for state changes in database, logic programs and arbitrary logical theories in a clean and declarative way [2]. $\mathcal{TR}$ extends classic first-order formulas with a new connective of sequential composition, denoted as: $\otimes$ (aka. *serial conjunction*). $\mathcal{TR}$ accounts not only for the database update orders, but also for other important features in several areas, such as transaction and subroutine definition, deterministic and non-deterministic actions, static and dynamic constraints, hypothetical and retrospective transactions, and a wide class of tests and conditions on actions [2]. The semantics of the sequential $\mathcal{TR}$ is based on *paths*, i.e., sequences of database

states, rather than states themselves. To define truth on paths in a completely general way, each path is associated with a first-order semantic structure, which specifies those atomic formulas that are true on the path. Intuitively, all formulas, atomic or complex, that are true on a path represent actions that take place along the path.

For the purpose of allowing concurrent update operations, $\mathcal{CTR}$ further extends $\mathcal{TR}$ with a new connective, | (aka. concurrent conjunction), and one modal operator $\odot$ [3]. In summary, the intended semantic of $\mathcal{CTR}$ connectives and modal operators are as follows:

- $\phi \wedge \psi$ means both ϕ and ψ must be executed along the same path.
- $\phi \vee \psi$ means execute ϕ or ψ non-deterministically.
- $\neg\, \phi$ means execute in any way provided that this will not be a valid execution of ϕ.
- $\phi \otimes \psi$ means first execute ϕ, then execute ψ.
- $\phi \mid \psi$ means that ϕ and ψ execute concurrently.
- $\odot\ \phi$, means that the execution of ϕ should not be interleaved with other transactions.

Concurrent processes in $\mathcal{CTR}$ execute in an interleaved fashion and can communicate and synchronize themselves, thereby increasing flexibility, performance, and power of the language. In contrast to other related research efforts, $\mathcal{CTR}$ is the only deductive database language that integrates concurrency, communication and database updates in a completely logical framework, including a natural model theory and a sound-and-complete proof theory [3]. Note that since $\mathcal{CTR}$ is built upon $\mathcal{TR}$, the statements about $\mathcal{TR}$ in what follows also hold for $\mathcal{CTR}$, unless explicitly specified.

In $\mathcal{CTR}$, the elementary database operations are encapsulated as a pair of oracles. The oracles come with a set of database states, upon which they can operate. Each database state could be seen as a set of data items, which are accessed by data and transition oracles. The data oracle, $\mathcal{O}^d(\mathrm{D})$, which maps from database states to sets of first-order formulas, i.e., $\mathcal{O}^d(\mathrm{D})$ presents queries related to a particular state D. The transition oracle, $\mathcal{O}^t(\mathrm{D}_1, \mathrm{D}_2)$, which maps pairs of states to set of ground atomic formulas, i.e., these transition oracles represented by the ground atomic formulas cause database state changes. For example, if $a \in \mathcal{O}^t(\mathrm{D}_1, \mathrm{D}_2)$, then a is an elementary update that changes the state from D_1 to D_2. In other words, the oracles provide semantics for data items: a *static semantics* querying a particular state and a *dynamic semantics* changing states.

The semantics of $\mathcal{CTR}$ is based on *multi-paths* or *m-paths*, which are generalized from the notion of $\mathcal{TR}$ *paths*. Formally, an m-path is a finite sequence of paths, where each path presents a period of continuous execution. For example, if D_1, D_2, ..., D_8 are database states, then $\langle \mathrm{D}_1\mathrm{D}_2\mathrm{D}_3, \mathrm{D}_4\mathrm{D}_5, \mathrm{D}_6\mathrm{D}_7\mathrm{D}_8 \rangle$ is an *m-path*. $\mathcal{CTR}$ formulas are interpreted by an *m-path* structure, which specifies those atoms are true on the *m-path*. Intuitively, a transaction formula is true on an *m-path* represents an action that takes place along the *m-path*.

4 Process Representation Using $\mathcal{CTR}$

4.1 Workflow Modeling

A workflow in this work generally defines a process as a set of flow elements that comprise tasks and gateways. Each task has a specific objective that contributes somehow to a workflow goal. The workflow describes how tasks can interact at a message level, with a definition of the control and data flow. A gateway controls the flow of a process and it implies a mechanism that either allows or disallows passage through the gateway. In general, each gateway involves a set of condition expressions associated with the gateway's incoming or outgoing sequence flows. The data flow captures data dependencies between tasks, and some data may be shared within a task or across several different tasks. In this work, we consider the data passing between tasks as event messaging, which not only can act as "data carrier", but also can be used a basis for representing composite events, thereby implementing complex workflow patterns, especially state-based workflow patterns.

Definition 1 (Scientific Workflow). *A scientific workflow is a collection of coordinated tasks composed to achieve complex goals. In $\mathcal{CTR}$, it is represented as a $\mathcal{CTR}$ Horn goal.*

A $\mathcal{CTR}$ Horn goal is any formula with the form:

- any atomic formula is a $\mathcal{CTR}$ Horn goal;
- if ψ and ϕ are $\mathcal{CTR}$ Horn goals, then so are the expressions: $\psi \otimes \phi$, $\psi \mid \phi$, $\psi \vee \phi$;
- $\odot\, \psi$, where ψ is a $\mathcal{CTR}$ Horn goal.

In workflow management systems, the tasks can be classified into two groups: *primitive* task and *composite* task. A *primitive* task is an atomic unit in workflows, and a *composite* task defines the execution order of a set of tasks.

Definition 2 (Primitive Task). *The primitive task corresponds to an atomic activity in a workflow and it is represented in $\mathcal{CTR}$ as a predicate.*

A primitive task represented by the predicate has the following format:

$$p(arg_1, ..., arg_n).$$

Here, the predicate p denotes the name of task and $\arg_1$, ..., $\arg_n$ ($n \geqslant 1$) are simple terms. Since this work focuses on representing complex workflows using $\mathcal{CTR}$, the details of task arguments are abbreviated as $p(\overline{X})$, where $\overline{X}$ denotes all the arguments that p takes.

The states in $\mathcal{CTR}$-based workflow modeling are regarded as data sets. Each state is a set of data items that represent current workflow status. More precisely, the data oracle $\mathcal{O}^d(D)$ which corresponds to the queries to a particular state. The transition oracle is defined as a task that consumes data to achieve certain

goals. Formally, for a task has both input(s) and output(s), task($\bar{i}$, $\bar{o}$) $\in \mathcal{O}^t$(D$_1$, D$_2$) iff D$_2$ = D$_1$ $\cup$ {$\bar{o}$} - {$\bar{i}$}. Here, $\bar{i}$ and $\bar{o}$ represent the input(s) and output(s) of a task, respectively. For a task has only input(s), task($\bar{i}$) $\in \mathcal{O}^t$(D$_1$, D$_2$) iff D$_2$ = D$_1$ - {$\bar{i}$}.

Similar to $\mathcal{CTR}$ that deals with state changes in deductive databases, a primitive task can both change the workflow state and act as a query to return an answer, which we refer to as *update-tasks* and *query-tasks*, respectively in this paper. The *update task* predicates are the ones which cause state transitions from one to another. The *query task* predicates are used to query the workflow state. They are helpful during the data passing between tasks. For example, a task needs to check if it can be started, such as its precedent tasks are completed or required data for its execution are available. This paper considers data passing as event messages, more details about event-based condition representation can be found in Section 4.3.

$$isPest(Name) \otimes treatment(Name, Scheme)$$

The execution of primitive tasks may also be guarded by conditional statements, i.e., preconditions and post-conditions in workflows. For example, the above formula denotes that the treatment is only required if the ant is a pest. The atom *isPest(Name)* must return true in order for the treatment to succeed. Different with simple qualifying truth-valued conditions, this kind of condition evaluation (e.g., the pest evaluation) usually involves complex decision logic depending on the amount and quality of knowledge about a scientific domain.

Definition 3 (Composite Task). *A composite task (aka. sub-workflow or subprocess) is the composition of a set of tasks, which could be primitive or composite. The composite task is defined as a $\mathcal{CTR}$ Horn rule with a form $p \leftarrow \phi$, where* p *is an atomic formula and ϕ is a $\mathcal{CTR}$ Horn goal.*

Since they define the composition of a workflow, this work refers $\mathcal{CTR}$ Horn rules to workflow formation rules. The head of the workflow formation rule is a predicate, which corresponds to a composite task only. As the primitive task, the name of the predicate denotes the name of a composite task. The body of the workflow formation rules recursively gives the definition of the composite task. For example, the process of ant identification is represented as: $identProcess \leftarrow Assign1 \otimes Assign2 \otimes (checkFood \mid checkNest \mid checkBody)$. The identification starts with a two-step sequential task assignment. After that, there are three parallel tasks that respectively validate ant food preference, nest structure and body feature.

In this paper, the connectives: $\otimes$, $\mid$, and $\vee$ have the following semantics:

- $\phi \otimes \psi$: execute task ψ after task ϕ. Model-theoretically, $\phi \otimes \psi$ is satisfied (or is true) on an m-path τ if only if ϕ and ψ are true on some m-paths τ_1, τ_2 whose concatenation $\tau_1 \bullet \tau_2$ reduces to τ, i.e., $\tau_1 \bullet \tau_2 = \tau$.
- $\phi \mid \psi$: tasks ψ and ϕ are executed concurrently. Model-theoretically, $\phi \mid \psi$ is true on an m-path τ if only if ϕ and ψ are true on some m-paths τ_1, τ_2 whose with an interleaving $\tau_1 \parallel \tau_2$ reduces to τ.

– $\phi \vee \psi$: represents a nondeterministic task, which means "execute task ϕ or execute task ψ". Model-theoretically, $\phi \vee \psi$ is true on an m-path if τ and only if either ϕ or ψ is true over on τ.

Due to the space limitation, we do not explain operations on m-paths, such as concatenation, interleaving, and reduction. For more details of these operations, see [3]. Besides these binary connectives (i.e., $\otimes$, $|$, and $\vee$) that compose two tasks into a composite one, $\neg$ is a unary connective and the satisfaction of $\neg\phi$ is defined as: $\neg\phi$ is true on any m-path τ if only if task ϕ is not true on the m-path τ. The $\odot\phi$ means the execution of task $\odot$ must not interleave with other concurrently running tasks. The satisfaction of $\odot\phi$ is true on any m-path τ if and only if τ is a path. Although they do not compose tasks directly, they play an important role to express constraints in scientific workflows.

It is worth noticing that, the body of $\mathcal{CTR}$ Horn rules and goals do not include the classical connective $\wedge$, which is usually expressed as a constraint [2].

Definition 4 (Optional Task). *Optional tasks are the ones that may not be directly needed for building workflows (since they do not affect the result of workflows), but allow for necessary variability and make workflows more sophisticated. An optional task is presented as a composite nondeterministic task as:* $\psi \vee \boldsymbol{state}$.

Here, the **state** is a special propositional constant, which is true on paths of length 1, i.e., on database states. That means the above formula is always evaluated to true, even if the task ψ is not executed. For example, the rule $fieldworkerProcess \leftarrow (consulation \vee \mathbf{state}) \otimes antDescription \otimes rcvMsg$ defines the process managed by the fieldworker. It is satisfied on a path τ as long as *antDescription* and *rcvMsg* are true on the path τ, even if the task *consultation* is not executed.

4.2 Reactive Logic Representation

To support the WsSWF execution, we provided a declarative rule-based workflow specification by *messaging reaction rules* [11]. The *messaging reaction rules* concern the message-driven conversations between agents, i.e., sending and receiving event messages associated with a conversation identifiers.

Bonner et al. [4] generally show how active database systems can be represented as active rules in $\mathcal{TR}$. The active rules are typically defined with a global scope (global state) and react on internal events of a reactive system, such as changes (updates) in the active database. Instead of active ECA rules, this work employs messaging reaction rules to describe the composition of workflows. The messaging reactive rules which build on (composite) events, are not only capable of capturing global event occurrences that trigger immediate reactions as in the active database trigger or ECA rules, but also can be used locally in a particular context, e.g., within a particular conversation and coordination protocol (e.g., a workflow), and are more suitable to specify workflow logic. The messaging reaction rules involve both for receiving and sending messages between distributed

agents. Messaging sending can be simply represented by a sending activity embedded in the body of a $\mathcal{CTR}$ Horn rule. The sending activity can send message to itself or other agents running locally in the same process but with different threads. This section mainly introduces how to represent the reactive workflow logic with reaction rules by $\mathcal{CTR}$.

The *reaction rules* concern the actions in response to events and actionable situations. They specify the conditions under which actions must be taken and describe the effects of action executions. They are a collection of reactive rules, which allow to specify and program such reactive systems in a declarative manner. The most general form of a reaction rule consists of the following parts.

define reaction rule r_rule
on $[event]$
if $[condition]$
then $[conclusion]$
do $[action]$
after $[post-condition]$
else $[else \quad conclusion]$
elseDo $[else/alternative \quad action]$

Depending on the parts of the general syntax, the reaction rules can be specialized into different types: *derivation rules* (if-then, are often used for logical event/action calculi), *production rules* (if-do), *trigger rules* (on-do), and *ECA rules* (on-if-do). For example, as shown in Figure 2, after the ant identification is done, a curational request is sent to a curator if the identification is successful. The processes can be represented as ECA rules:

define reactive rule $identDone$
on $identDone(AntDesc, Result)$
if $isIdentSuccess(AntDesc, Result)$
do **nothing**.
define reactive rule $identDone$
on $identDone(AntDesc, Result)$
if $isIdentFaild(AntDesc, Result)$
do $humanIdent(AntDesc)$.

$\mathcal{CTR}$ provides a complete formalization for the system behavior and these ECA rules can be further represented by $\mathcal{CTR}$ as follows:

$$identification \leftarrow (checkFood \mid checkNest \mid checkBody) \otimes identDone$$
$$identDone \leftarrow isIdentSuccess(AntDesc, Result) \otimes \mathbf{state}$$
$$identDone \leftarrow isIdentFaild(AntDesc, Result) \otimes humanIdent(AntDesc)$$

4.3 Complex Event Processing

The reaction rules specify under which conditions a task can execute and these conditions determine intelligent routings at runtime. As we consider the data passing between tasks as event messaging, it is necessary to employ the CEP technology to represent composite events, thereby implementing complex workflow patterns.

A composite event is the combination of several base events. Each composite event is usually described by an event pattern, which contains event templates, relational operators and variables. However, the logic-based approaches are goal-driven and the check of a given event pattern is always performed at the time when the goal is set [1]. To overcome this limitation, Anicic et al. [1] propose an *event-driven complex event detection* approach of detecting complex event patterns based on $\mathcal{CTR}$. This paper introduces a scientific workflow representation that imposes no constraints on the reaction time with regard to the event processing, also known as a *any-time* reaction rule system. There is no need to process these events in real-time or detect particular event patterns (e.g., sequence events, concurrent events [1]), and thus we only represent conjunction and disconjuction of events based on $\mathcal{TR}$, rather than give a comprehensive complex event pattern representation involved in the *real-time* CEP.

Note that the states in $\mathcal{TR}$-based CEP need to be seen from a different perspective. They are defined as changes of base events during complex event pattern detection and what stored in the databases are ground atomic event formulas, D. More precisely, the data oracle $\mathcal{O}^d(\mathrm{D})$ which corresponds to check if the occurrence of a base event, which simply returns the event formulas. Formally, $e \in \mathcal{O}^d(\mathrm{D})$ iff the event e is in the database. For each base event e in D, the transition oracle defines two new predicates: *ins(e)* and *del(e)*, representing insertion of an event when it is detected and deletion of an event when corresponding complex event pattern is detected. Formally, $ins(e(\bar{t})) \in \mathcal{O}^t(\mathrm{D}_1, \mathrm{D}_2)$ iff $\mathrm{D}_2 = \mathrm{D}_1 \cup \{\mathrm{e}(\bar{t})\}$, and similarly $del(e(\bar{t})) \in \mathcal{O}^t(\mathrm{D}_1, \mathrm{D}_2)$ iff $\mathrm{D}_2 = \mathrm{D}_1$ - $\{\mathrm{e}(\bar{t})\}$.

Conjunction of Events An event pattern based on the conjunction of events requires all base events are detected. For example, $e(T) \leftarrow a(T_1) \wedge b(T_2) \wedge c(T_3)$ defines a composite event e occurs when the base events a, b and c are detected. Following the event-driven complex event detection approach [1], we simplify the detection of a conjunction event pattern as follows:

$a(T_1) : -ins.a(T_1) \otimes check$
$a(T_1) : -ins.a(T_1) \otimes not(check) \otimes \mathbf{state}$
$b(T_2) : -ins.b(T_2) \otimes check$
$b(T_2) : -ins.b(T_2) \otimes not(check) \otimes \mathbf{state}$
$c(T_3) : -ins.c(T_3) \otimes check$
$c(T_3) : -ins.c(T_3) \otimes not(check) \otimes \mathbf{state}$

$$check \leftarrow a(T_1) \otimes b(T_2) \otimes c(T_3) \otimes max(T_1, T_2, T_4) \otimes max(T_3, T_4, T_5) \otimes e(T_5) \otimes del.a(T_1) \otimes del.b(T_2) \otimes del.c(T_3)$$

The first six rules represent that the base events a, b and c are firstly inserted into the database when they are detected. The *check* rule checks if these base events already exist in the database. The composite event e occurs if all base events a, b and c are detected. As soon as the conjunction event pattern is detected, the base events a, b and c are removed from the database. Note the base events a, b, and c can always be successfully inserted into the database even the *check* rule fails (represented by a *Negation As Failure* inference rule *not(check)*). This is really important because that the occurred base events are

always known during a complex event pattern detection. Suppose the base events a and b are detected before c, the *check* rule fails but the base event a and b are both successfully inserted into the database. When c is detected, the third rule firstly inserts the event c into the database, then triggers the composite event e.

Disjunction of Events An event pattern based on the disjunction of events is satisfied when any base event is detected. For example, $e(T) \leftarrow a(T_1) \vee b(T_2) \vee c(T_3)$ defines the composite event e occurs when any one of the events a, b and c is detected. In this work, the detection of a disjunction event pattern is represented as follows:

$a(T_1) : -ins.a(T_1) \otimes check$
$a(T_1) : -ins.a(T_1) \otimes not(check) \otimes \mathbf{state}$
$b(T_2) : -ins.b(T_2) \otimes check$
$b(T_2) : -ins.b(T_2) \otimes not(check) \otimes \mathbf{state}$
$c(T_3) : -ins.c(T_3) \otimes check$
$c(T_3) : -ins.c(T_3) \otimes not(check) \otimes \mathbf{state}$

$check \leftarrow a(T_1) \otimes e(T_1) \otimes del.a(T_1)$
$check \leftarrow b(T_2) \otimes e(T_2) \otimes del.b(T_2)$
$check \leftarrow c(T_3) \otimes e(T_3) \otimes del.c(T_3)$

Compared with the conjunction event pattern mentioned above, there are three *check* rules that detect the disjunction of events. It means that either base event a, b or c is detected, then one *check* rule is evaluated to *true* and the composite event e occurs. Note that we assume that the events a, b and c are mutually exclusive in this work to insure the composite event e that occurs only once.

The *conjunction* and *disjunction* are standard operators to describe complex event patterns. They are often used to represent the gateways acted as join connectors in workflows. With the aforementioned approach, it is also possible to represent more complex join connectors in workflows, such as a composite event e is detected when any two of the events a, b and c occur. Moreover, since we provide a declarative logic-based workflow representation, it is possible to represent complex conditional logic in the process of complex event detection to to make more sophisticated workflows. Due to the space limitation, they are not shown in this paper.

5 Mapping $\mathcal{CTR}$-based Workflow Representation to Prova

In this section we introduce a Prova implementation of the $\mathcal{CTR}$-based logic WsSWF representation. Prova[2] is both a Semantic Web rule language and a high expressive distributed rule engine. Prova rule language combines different rule types and provides a highly expressive, hybrid, declarative and compact rule programming language, which combines the declarative programming paradigm

[2] https://prova.ws/

and object-oriented programming and the Semantic Web approach [5]. On the other hand, Prova engine supports complex reaction rule-based workflows, rule-based complex event processing, distributed inference services, rule interchange, rule-based decision logic and dynamic access to external data sources, Web-based services, and Java APIs [6].

The $\mathcal{CTR}$-based workflow logic is set of $\mathcal{CTR}$ Horn rules that define the composition of a workflow. Before the workflow execution, all these $\mathcal{CTR}$ Horn rules need to be translated into Prova rules. As mentioned in Section 4, the body of a $\mathcal{CTR}$ Horn rule represents a $\mathcal{CTR}$ Horn goal with forms of: $\psi \otimes \phi$, $\psi \mid \phi$, $\psi \vee \phi$; $\odot \psi$.

Because Prova is built upon Prolog, a serial Horn rule $p(\overline{X}) \leftarrow \psi(\overline{Y}) \otimes \phi(\overline{Z})$ can be directly translated into a Prova rule with a form $p(\overline{X}) : -\psi(\overline{Y}), \phi(\overline{Z})$. This is simply done by replacing serial connector $\otimes$ with a comma ",".

Disjunction of tasks defines a nondeterministic composite task and they can be simply implemented by a set of Prova rules imposed by different conditions. For example, $p(\overline{X}) \leftarrow \psi(\overline{Y}) \vee \phi(\overline{Z})$ can be implemented by: (1) $p(\overline{X}) \leftarrow cond1, \psi(\overline{Y})$; (2) $p(\overline{X}) \leftarrow cond2, \phi(\overline{Z})$. *cond1* and *cond2* denote the conditions under which to select branches. Similarly, it is easy to translate optional tasks, where the propositional constant *state* is translated to a condition without subsequent goals, i.e., (1) $p(\overline{X}) \leftarrow cond1, \psi(\overline{Y})$; (2) $p(\overline{X}) \leftarrow cond2$. The implementation of concurrent tasks is based on reactive event messaging, and we will introduce later in this section.

The reactive event messaging can be implemented by the following Prova constructs to send and receive one or more context-dependent multiple outbound or inbound event messages:

Listing 1.1. Prova constructs of messaging reaction rules

```
1   sendMsg(XID, Protocol, Agent, Performative, Payload|Context)
2   rcvMsg(XID, Protocol, From, Performative, Payload|Context)
3   rcvMult(XID, Protocol, From, Performative, Payload|Context)
```

Here, *XID* is the conversation identifier of a message. *Protocol* defines the communication protocol. *Agent* and *From* denote the destination and the source of the message, respectively. *Performative* describes the pragmatic context in which the message is sent. A standard nomenclature of the performative is, e.g., the FIPA Agents Communication Language (ACL)[3]. And *Payload—Context* denotes the actual content of the message.

There are two types of Prova reaction rules: *inline* and *global* reaction rules. The *inline* reaction rules are used to accept just one message, a specified number of messages. They are usually in the body of a rule and act as sub-goals of the rule. For example, the logic after the ant identification can be translated into:

Listing 1.2. Inline reaction rule example

```
1 identProcess(XID, AntDesc, Result) :-
2     ...,
3     rcvMsg(XID,async,Agent,answer,identDone(AntDesc, Result)),
```

[3] http://www.fipa.org/repository/aclspecs.html

```
4     isSuccess(Result), !.

6 identProcess(XID, AntDesc, Result) :-
7     rcvMsg(XID,async,Agent,answer,identDone(AntDesc, Result)),
8     not(isSuccess(Result)), !,
9     sendMsg(XID, async, humanAgent, ident(AntDesc)),
10    rcvMsg(XID, async, humanAgent, ident(Result)),
11    ....
```

The pattern specified by inline reaction rule *rcvMsg* (Line 3 and 7) indicates an interest in an event message of pre-defined type answer on the *async* protocol. This kind of inline reactions fire only once. The *global* reaction rules look exactly like general Prova rules but their semantics are more aligned with reactive rules rather than derivation rules. In contrast to the *inline* reaction rules, the *global* reaction rule has a rule base lifetime scope, i.e., it is active while the rule base runs on a Prova engine and it is ready to receive any number of messages as they arrive to the agent. For example, the process of waiting form external identification request can be implemented as follows:

Listing 1.3. Global reaction rule example

```
1 rcvMsg(XID,esb,Agent,request, ident(AntDesc)) :-
2     identProcess(AntDesc),
3     ....
```

With the reactive messaging, it is possible to implement the concurrent tasks in Prova. Prova has three internal protocols for reactive messaging: *self*, *task*, *async*, and *swing* (not used in this paper). The inline reaction rule *rcvMsg* itself is executed on the main thread, but it creates an inline reaction waiting for a reply according to its protocol. The *self* protocol indicates that the thread waiting for a reply is the main thread and this protocol can ensure fully sequential processing of received messages. The *task* protocol means the thread waiting for a reply is randomly taken from a task thread pool. This pool is used for running tasks achieving maximum throughput. The *async* protocol makes good use of the conversation identifier and the thread waiting for a reply is chosen in terms of the conversation identifier. This means a conversation is always mapped to one thread. In Prova, we can use both *task* and *async* protocol of the reactive messaging to implement concurrent processes. For example, the following Prova code snippet implements the ant identification that involves three tasks in parallel.

Listing 1.4. Concurrent task example

```
1 identProcess(XID, AntDesc, Result) :-
2     ...,
3     init_ident(),
4         sendMsg(XID,task,0,inform, checkFood(AntDesc, Res1)),
5         sendMsg(XID,task,0,inform, checkNest(AntDesc, Res2)),
6         sendMsg(XID,task,0,inform, checkBody(AntDesc, Res3)).

8 init_ident() :-
9     rcvMsg(XID,task,From,inform, checkFood(AntDesc, Res1)),
10    checkFood(AntDesc, Res1),
11    sendMsg(XID, async, 0, inform, checkFood(AntDesc, Res1)).

13 init_ident() :-
```

```
14      rcvMsg(XID,task,From,inform, checkNest(AntDesc, Res2)),
15      checkNest(AntDesc, Res2),
16      sendMsg(XID, async, 0, inform, checkNest(AntDesc, Res2)).

18  init_ident() :-
19      rcvMsg(XID,task,From,inform, checkBody(AntDesc, Res3)),
20      checkBody(AntDesc, Res3),
21      sendMsg(XID, async, 0, inform, checkBody(AntDesc, Res3)).
```

The predicate *init_ident()* (Line 3) creates three parallel processing streams in one agent and each inline reaction rule randomly selects a thread from the task pool in order to wait for the event to trigger the task in branch, thereby executing them in parallel.

The logic-based CEP can also be implemented by the event-driven computation of event patterns. In Prolog, the updates are not undone during backtracking. For instance, the Prolog rule "p :- assert(event(a)), fail." asserts *event(a)* into the database even though the execution fails. Although it is a limitation to implement some $\mathcal{TR}$-based applications that require to undo the updates during backtracking. But for the logic-based CEP, this is quite useful since the occurred events need to be stored in the database even the complex event pattern is not matched, i.e., the rule used to check the complex event pattern fails. Benefit from this feature, two $\mathcal{CTR}$ rules used to detect base event (see Section 4.3) can be reduced to one. Due to the space limitation, the following Prova code snippet only implements the detection of a conjunction event pattern:

Listing 1.5. Complex event pattern computation example

```
1  detect(XID) :-
2      rcvMsg(XID,async,From,inform, e(a)), assert(e(a)), check(e).
3  detect(XID) :-
4      rcvMsg(XID,async,From,inform, e(b)), assert(e(b)), check(e).
5  detect(XID) :-
6      rcvMsg(XID,async,From,inform, e(c)), assert(e(c)), check(e).

8  check(e) :-
9      e(a), e(b), e(c), sendMsg(XID, async, 0, inform, e),
10     retract(e(a)), retract(e(b)), retract(e(c)).
```

Suppose *e(a)* is detected first, it is immediately recorded as an event fact into the database. Since the base events *e(b)*, and *e(c)* have not been detected, the rule *check(e)* (Line 8-10) is evaluated to *false* at the moment. The rule *check(e)* reasoning is triggered again when another base event is detected. It can only be evaluated to *true* when all base events are detected, then the composite event e occurs and the base events are removed from the database. Note that the base events a, b and c are simplified here for clarity. In concrete applications, they can be associated with arguments to to detect different composite events.

To sum up, Table 1 shows the mappings from the $\mathcal{CTR}$-based workflow representation to Prova. For the purpose of connecting distributed Prova agents, a tool for rule-based collaboration Rule Responder[4] that is built upon Enterprise Service Bus (ESB) Mule, is often used as a communication middleware.

[4] http://responder.ruleml.org/

Table 1. Mapping $\mathcal{CTR}$-based Workflow Representation to Prova

Workflow Definition	**$\mathcal{CTR}$-based Representation**	**Prova**
Workflow	$\mathcal{CTR}$ Horn goal	Prova goal
Sub-process	$\mathcal{CTR}$ Horn rule	Prova rule
Sequential task	Serial conjunction ($\otimes$)	Comma separated
Nondeterministic task	Classic disconjunction ($\vee$)	Prova rules with conditions
Concurrent task	Concurrent conjunction ($\mid$)	Concurrent reactive messaging
Task interaction	Reactive messaging	Messaging constructs
Gateway (Join)	Logic-based CEP	Event-driven computation of event patterns

6 Related Work

Existing scientific workflow management systems (WFMSs) (such as Kepler, Triana, Taverna, Pegasus) are mainly designed for structured compute intensive or data intensive applications, and each system has its own application areas. They have proprietary workflow languages and provide limited expressiveness to describe decision logic and conditional reactive logic. The WsSWFs considered in this work not only involve complex task dependencies that cannot be easily described by imperative procedural workflow languages, but also contain knowledge-intensive decision centric steps, which need the coordination of scientists or computer agents as a team. That is why logic-based $\mathcal{CTR}$ is employed as a theoretical basis for the declarative description of the WsSWFs. Some efforts are also trying to provide a logical framework for modeling flexible workflows: Roman et al. [8] introduce an expressive logical model for process specification, contracting for services, service enactment, and reasoning. Based on $\mathcal{CTR}$, the model can model many constraints used for specifying contracts. Compared with our work, the model in [8] focuses on an expressive theoretical representation on both process specification and service contracting. Our work, however, not only explicitly considers the representation of WsSWFs, but also attempts to implement it with a Web rule language Prova. DECLARE is a prototype of a WFMS that uses a constraint-based process modeling language for the development of declarative models describing loosely-structured processes [7]. Based on Linear Temporal Logic (LTL), DECLARE provides several constraint templates to model constraints, such as "init", "1..*", "response", "responded existence" and "responded absence". Our $\mathcal{CTR}$-based workflow modeling also can represent many temporal constraints (e.g., "init", "response") with path-based constraints. Moreover, in contrast to these efforts, the logic-based representation of the WsSWFs using $\mathcal{CTR}$ in this paper focuses on representing the reactive interactions between tasks with a purpose of supporting the WsSWFs.

7 Conclusion

Currently there are many rule-based workflow languages, which support flexible service composition and model weakly-structured process logic with declara-

tive rule languages. However, many of them only provide a static syntactical process description without a precise declarative formal semantics. This paper provided a $\mathcal{CTR}$-based formal model for the WsSWFs and especially considered conversation-based reactive workflow logic representation and event-driven computation of complex event patterns.

References

1. Darko Anicic, Paul Fodor, Roland Stuhmer, and Nenad Stojanovic. Event-Driven Approach for Logic-Based Complex Event Processing. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, pages 56–63, Washington, DC, USA, 2009. IEEE Computer Society.
2. Anthony J. Bonner and Michael Kifer. An Overview of Transaction Logic. *Theor. Comput. Sci.*, 133(2):205–265, 1994.
3. Anthony J. Bonner and Michael Kifer. Concurrency and Communication in Transaction Logic. In *JICSLP*, pages 142–156, 1996.
4. Anthony J. Bonner, Michael Kifer, and Mariano Consens. Database Programming in Transaction Logic. In *In Proc. 4th Int. Workshop on Database Programming Languages*, pages 309–337, 1993.
5. Adrian Paschke. Rule Responder HCLS eScience Infrastructure. In *Proceedings of the 3rd International Conference on the Pragmatic Web: Innovating the Interactive Society*, ICPW '08, pages 59–67, New York, NY, USA, 2008. ACM.
6. Adrian Paschke and Zhili Zhao. Rule Responder: A Rule-Based Semantic eScience Service Infrastructure. In Albert Burger, M. Scott Marshall, Paolo Romano 0001, Adrian Paschke, and Andrea Splendiani, editors, *SWAT4LS*, volume 698 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.
7. M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, pages 287–. IEEE Computer Society, Washington, DC, USA, 2007.
8. Dumitru Roman, Michael Kifer, and Dieter Fensel. WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic. In Manfred Hauswirth, Manolis Koubarakis, and Sean Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, Heidelberg, June 2008. Springer Verlag.
9. Jacek Sroka, Jan Hidders, Paolo Missier, and Carole Goble. A Formal Semantics for The Taverna 2 Workflow Model. *Journal of Computer and System Sciences*, 76(6):490 – 508, 2010.
10. Zhili Zhao and Adrian Paschke. A Rule-Based Agent Framework for Weakly-Structured Scientific Workflows. In Witold Abramowicz, editor, *Business Information Systems Workshops*, volume 160 of *Lecture Notes in Business Information Processing*, pages 290–301. Springer Berlin Heidelberg, 2013.
11. Zhili Zhao and Adrian Paschke. Event-Driven Scientific Workflow Execution. In Marcello Rosa and Pnina Soffer, editors, *Business Process Management Workshops*, volume 132 of *Lecture Notes in Business Information Processing*, pages 390–401. Springer Berlin Heidelberg, 2013.
12. Zhili Zhao and Adrian Paschke. Rule Agent-Oriented Scientific Workflow Execution. In Herbert Fischer and Josef Schneeberger, editors, *S-BPM ONE - Running Processes*, volume 360 of *Communications in Computer and Information Science*, pages 109–122. Springer Berlin Heidelberg, 2013.