Process Representation Using Transaction Logic

Reza Basseda

Stony Brook University, Stony Brook, NY, 11794, USA

Abstract. Representing and answering the queries about the dynamic behavior of processes in knowledge base systems has become a challenging research area in the field of logic programming and knowledge representation systems. In this report, we are going to show how transaction logic can be used to efficiently represent dynamic behavior embedded in different domains. The ability of properly representing state changes in transaction logic enables us to express dynamic behavior of processes in different domains. The use of transaction logic to represent dynamic behavior decreases the size of knowledge bases and the query response time in comparison with other existing approaches. The efficiency of our method along with other features of transaction logic and its theoretical basis makes it an appropriate approach to represent dynamic behavior of processes in various domains.

Keywords: Process Representation, Transaction Logic

1 Introduction

In many real world applications of knowledge representations systems, effective representation of processes embedded in the domain knowledge enables the knowledge base system to answer a wide range of queries about those processes. For example, in the medical domain, physiology explains different processes by showing how different organs and parts of a human body interacts with each other while anatomy discusses the structure of the human body and its organs. A medical knowledge base system needs to represent both of the physiological and anatomical knowledge in order to be able to answer the queries about diseases and medical experiments.

Let us illustrate this concept via an example. Consider the process of myocardial infarction (MI) or acute myocardial infarction (AMI) in medical science, which is commonly known as a heart attack. Basically, myocardial infarction results from the interruption of blood supply to a part of the heart, causing heart cells to die. This is most commonly due to occlusion (blockage) of a coronary artery following the rupture of a vulnerable atherosclerotic plaque, which is an unstable collection of lipids (cholesterol and fatty acids) and white blood cells (especially macrophages) in the wall of an artery. The resulting ischemia (restriction in bloood supply) and ensuing oxygen shortage, if left untreated for a sufficient period of time, can cause damage or death (infarction) of heart muscle tissue (myocardium) [**?**]. The process starts with the step of increasing cholesterol and other lipids in the blood. This step is followed by the step of lipid dysregulation. After the step of lipid dysregulatoin, the formation of atherosclerotic plaque happens. The formation of atherosclerotic plaque causes narrowing of the coronary arteries and narrow coronary arteries leads to have to have insufficient blood supply for myocardial muscles. Finally, insufficient blood supply for myocardial muscles results in myocardial infarction. Representation of such process in a knowledge base system needs various features to exist in the system. The system needs to represent a process in terms of different steps. Each of those steps can be defined as an abstract process as well. Each process defines a set of potential dynamic changes in the system over the set of knowledge base facts. The execution of each step also depends on the various logical formulas evaluated at the different states of the knowledge base which are created during the course of the execution. It is apparent that those dynamic and static definitions of changes and terms are tightly connected to each other.

This example shows that we need to explicitly represent processes in the knowledge base systems as they are associated with some features which may be involved in query answering. For example, time duration of execution of a process or the name of a process may be queried. However, explicit representation of processes may raise other issues. For example, treating processes as first class entities of a knowledge base system may require us to express different relationships between those entities.

There are several logic programming frameworks which can be used to address the process representation problem in knowledge base systems. Situation calculus [1] provides a representation for state changes in logic. The basic concepts in the situation calculus are situations, actions, and fluents. To describe a dynamic domain in the situation calculus, one specifies a set of actions describing what changes the situations. A set of fluents is also required to describe the changing situations. Like the situation calculus, the event calculus [2] has actions, which are called events. It also has changing properties or fluents. But unlike the standard situation calculus in which an exact sequence of hypothetical actions is represented, the event calculus is based on possibly incomplete specification of a set of actual event occurrences. Different event calculus extensions addressed the frame problem in different ways [3].

A class of action languages has been developed that is independent of a specific axiomatization [4] [5] [6]. These languages try to provide high expressiveness, natural-language-like syntax, and clear formal semantics, which are important in procedural knowledge representation. [7] uses a modular action language, $\mathcal{ALM}$ in order to represent procedural knowledge. It was used to formalize of biological processes, including cell division, in $\mathcal{ALM}$. [8] also uses an action modeling scripting language to represent and reason about signaling networks. [9] is also an variation of action language $\mathcal{A}$[10] to represent procedural knowledge in biological networks. [11] also can be used to represent dynamic behavior in domain knowledge base systems.

Both of the above mentioned approaches are facing difficulties when it comes to process representation. Since situation calculus is using monotonic reasoning and scientific knowledge representation which usually involves non-monotonic reasoning, situation calculus is not suitable for process representation in scientific domain. Process representation in event calculus has several problems. This formalization of events is intended as a formal analysis of the concepts rather than as a program or even as a program specification [2]. As updates in event calculus are additive and do not delete information about events, execution of a large number of process steps may be impractical. Explicit declaration of relation between processes also requires a large number of auxiliary predicates and rules. For example, to represent a containment relation between two processes, several rules and facts may be required. Although action modeling languages can represent processes in terms of action execution sequences, they are not scalable knowledge representation languages. Since they don't support features required for efficient knowledge representation such as object orientation and higher orderness, scientific knowledge representation using this type of languages is harder and less reusable. Action and process definition syntax in action modeling languages is usually different than regular logic programming syntax. That difference makes the integration of dynamic behavior and static specification of domain knowledge difficult using action modeling languages.

Transaction Logic is an extension of classical predicate logic that accounts in a clean and declarative fashion for the phenomenon of state changes in logic programs and databases [12]. Our case study shows that $\mathcal{TR}$ eases the expression of dynamic behavior of the processes embedded in different domains. The logic of state changes provided by $\mathcal{TR}$ facilitates the inference about processes represented in $\mathcal{TR}$. That representation of state changes within the logic formulas provides non-monotonic reasoning for procedural knowledge in scientific domains. Since $\mathcal{TR}$ is a declarative formalism for specifying and executing procedures that update a logical theory, it can naturally express both the static knowledge and the dynamic behaviors in different scientific domains. We can also combine $\mathcal{TR}$ with other logical formalism such as F-logic and HiLog in order to have object oriented and higher order formalisms. Those logical formalisms simplify the representation between processes. As dynamic behavior representation in $\mathcal{TR}$ does not need to have any axiomatization in order to address the frame problem, the process representation in $\mathcal{TR}$ is more scalable in comparison with other logical formulations of processes.

$\mathcal{TR}$ includes a Horn-like fragment which supports logic programming. This logic programming framework simplifies the integration of dynamic behavior with other components of knowledge base systems. Specification of processes in the language used for specification of logical terms and rules makes the expressiveness of logical formulas and terms available for process representation. This logic programming framework also helps us to easily express a wide range of queries about the dynamic behavior of processes. $\mathcal{TR}$ is also implemented in Flora [13], which is a perfect system for knowledge representation and reasoning.

Our process representation approach using $\mathcal{TR}$ shows that other features of $\mathcal{TR}$ can also help to have a very expressive and robust process specification in a knowledge base systems. For example, we took advantage of hypothetical queries to represent the concept of fault tolerant processes in the knowledge base systems. Incremental tabling and other developments in our implementation framework also may help us to improve query answering time.

We will explain our process representation technique in the next section. Section 3 will describe our case study experiments. We will also have a brief analysis of our results in section 3, and section 4 will conclude our study.

2 Methodology

The over all representation of processes in $\mathcal{TR}$ is simple and natural. We classify processes into two groups: complex processes and primitive processes. A complex process is a sequence of complex or primitive processes and a primitive process is a single step of execution. The relationships between processes can be represented by simple logical predicates. For example, suppose process p[1] is a sequence of processes p_1, p_2, p_3. We use $complex\_process/1$ and $primitive\_process/1$ to indicate the type of process. $first\_step(p, p_1)$ says that process p_1 is the first step of process p. $next\_step(p, p_1, p_2)$ and $next\_step(p, p_2, p_3)$ show that p_1 in p is followed by p_2 and p_2 in p is followed by p_3. We do not provide the formal explanation of these concepts due to space limit.

To keep track of the execution of complex processes, we need a structure maintaining the execution status of the complex process. The current step of a process, $current\_step(P, SP)$, is an example of such a structure. A primitive process does not have internal structure.

Sequential execution of subprocesses can be defined recursively as shown below.

$$\begin{aligned} execute(P) \longleftarrow & complex\_process(P) \land current\_step(P, CS) \otimes \\ & execute(CS) \otimes advance(P, CS) \otimes execute(P). \end{aligned} \tag{1}$$

A complex process will be successfully executed if all of its subprocesses successfully complete their execution.

$$\begin{aligned} execute(P) \longleftarrow & complex\_process(P) \land \\ & current\_step(P, CS) \land \sim next\_step(P, CS, \_). \end{aligned} \tag{2}$$

$advance(P, CS)$ in (1) above refers to changing the execution status of process P. For example, it can represent the current step change as follows. Note that elementary transactions of *insert* and *delete* are defined in our transition oracle as shown in [12].

[1] In this section, capital letters denote logical variable and lower case is used to denote constant and predicate symbols

$$\begin{aligned} advance(P, CS) \longleftarrow & complex\_process(P) \wedge current\_step(P, CS) \\ & \wedge next\_step(P, CS, NS) \\ & \otimes current\_step.delete(P, CS) \otimes current\_step.insert(P, NS). \end{aligned} \tag{3}$$

Execution of primitive processes can be defined in terms of elementary transactions *insert* and *delete*. We also can extend the transition oracle and define a specific primitive process execution as a elementary transaction. For example, assume the transaction $doit(P)$ executes the elementary transaction associated to the primitive process P. We can show the *successful* and *failed* execution of P as in (4) and (5). Note that no matter $doit(P)$ finishes successfully or not, $execute(P)$ will finish successfully. However the value of $result(P, R)$ in the final state of knowledge base will depend on the execution of $doit(P)$.

$$\begin{aligned} execute(P) \longleftarrow & primitive\_process(P) \otimes doit(P) \otimes \\ & result.insert(P, success). \end{aligned} \tag{4}$$

$$\begin{aligned} execute(P) \longleftarrow & primitive\_process(P) \otimes \sim doit(P) \otimes \\ & result.insert(P, failure). \end{aligned} \tag{5}$$

Execution of primitive process may also include some conditional statements. We can use such precondition and postcondition statements to guard the execution of a primitive process. $precondition(P)$ and $postcondition(P)$ predicates can simply express those postcondition and precondition statements for a primitive process P. Now, we can show the *successful* and *failed* execution of P as in (6) and (7). In this formulation of $execute(P)$, the evaluation of this predicate will depend on the evaluation of $precondition(P)$ and $postcondition(P)$. For example, assume that the execution of process p_3 is guarded with the conjunction of g and the successful execution of process p_1 and it does not have any postcondition. This can be represented as (8) and (9).

$$\begin{aligned} execute(P) \longleftarrow & primitive\_process(P) \wedge precondition(P) \otimes doit(P) \otimes \\ & result.insert(P, success) \wedge postcondition(P). \end{aligned} \tag{6}$$

$$\begin{aligned} execute(P) \longleftarrow & primitive\_process(P) \wedge precondition(P) \otimes \sim doit(P) \otimes \\ & result.insert(P, failure) \wedge postcondition(P). \end{aligned} \tag{7}$$

$$precondition(p_3) \longleftarrow g \wedge result(p_1, success). \tag{8}$$

$$postcondition(p_3) \longleftarrow true. \tag{9}$$

Serial conjunctions used in our formulas allow sequential execution of subprocesses. Note that in (1), if transaction $execute(CS)$ fails and returns false, the transaction $execute(P)$ also fails and returns false. One can use hypothetical reasoning to have more fault-tolerant processes. For example, (10) redefines (1) such that if $execute(CS)$ fails, $failed(CS)$ will be executed instead and $execute(P)$ will be completed and return true. $\sim$ in (10) denotes default negation and term $\sim \diamond execute(CS)$ draws if the hypothetical execution of $execute(CS)$ fails. Similarly we can redefine (5) as (11). This kind of reasoning can be useful in *exception handling*.

$$\begin{aligned} execute(P) \longleftarrow & complex\_process(P) \wedge current\_step(P, CS) \otimes \\ & \sim \diamond execute(CS) \otimes failed(CS) \otimes \\ & advance(P, CS) \otimes execute(P). \end{aligned} \tag{10}$$

$$\begin{aligned} execute(P) \longleftarrow & primitive\_process(P) \otimes \sim \diamond doit(P) \otimes \\ & result.insert(P, failure). \end{aligned} \tag{11}$$

A sample implementation of this approach is available in our demo.

3 Example: A cell mitosis division process

Through a simple implementation of mitosis cell division process, we compared our $\mathcal{TR}$ process representation technique with action modeling languages. We used Flora-2 [13], an object oriented knowledge base reasoning system, to develop an abstract biological knowledge base including mitosis cell division process. We compared our $\mathcal{TR}$-based system with those obtained by a manual translation of the same knowledge base to the $\mathcal{AL}_d$ action modeling language [7]. We also compared our implementation with an implementation based on the event calculus concepts in Flora-2.

As shown in Figure 1, the comparison of systems in terms of lines of code shows that $\mathcal{TR}$ provides a more succinct representation by far. Generating auxiliary rules for inertia axioms, completeness of states, and execution possiblity, complicates $\mathcal{AL}_d$ programs. We should also mention that the $\mathcal{AL}_d$ program is including just one test query but $\mathcal{TR}$ and the event calculus based solutions responds to 7 queries. This means that for the equal test conditions, the size of $\mathcal{AL}_d$ program would be much more than 707 lines.

As shown in Figures 2 and 3, via a set of sample queries, we considered the response time of above mentioned implementations. The execution time also shows that, $\mathcal{TR}$ is much faster than $\mathcal{AL}_d$. Moving to $\mathcal{TR}$ from event calculus,

Method	Lines of Code
event calculus	1196
$\mathcal{AL}_d$	707
$\mathcal{TR}$	490

Fig. 1. Length of sample knowledge base system implemented using different methods

the overhead of transactional updates leads to decrease in the response time to test queries, which we are yet to understand. Our solution in $\mathcal{TR}$ also suffered from a bug in XSB which prevented us from taking advantages of incremental tabling. Because of that we had to refresh several tables after each transactional update. Fixing that bug will improve $\mathcal{TR}$'s response time.

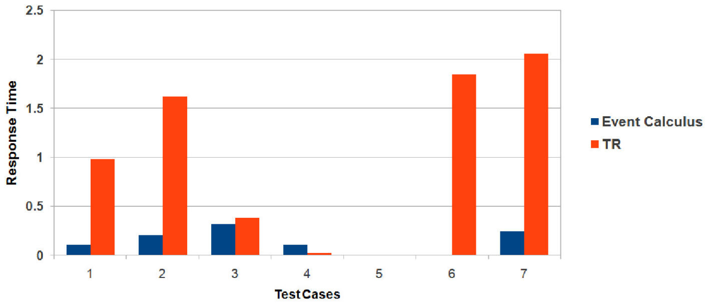

Fig. 2. Comparison of response time in event calculus and $\mathcal{TR}$ for different test cases

This example apparently shows that $\mathcal{TR}$ is promising candidate for representing processes in knowledge base systems.

There are other features in Flora-2, which we used in our development. Object orientated syntax and higher order rules are examples of these features. As those features are beyond the scope of this report, we do not consider them here.

4 Conclusion

In this paper, we discussed several methods for representing processes, which are included in knowledge representation systems as part of domain knowledge. As mentioned before, dynamic domain languages require a large number of auxiliary rules and axioms, which complicates knowledge representation. They also lack many features that facilitate knowledge representation and process specification such as higher order rules and object orientation. $\mathcal{TR}$ allows definitions of processes as first class entities. Through an experiment, we showed that, it also simplifies programs and makes them more extensible and reusable. It also apparently improves the response time in comparison with methods based on action modeling languages.

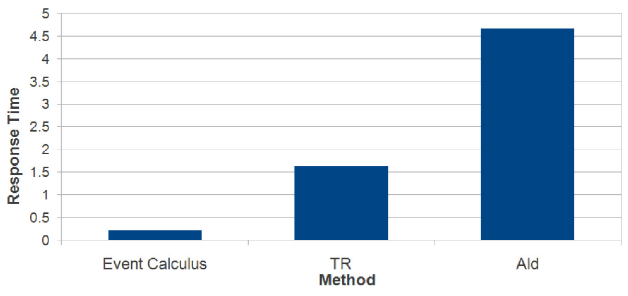

Fig. 3. Comparison of response time in different methods

We are planning to investigate $\mathcal{TR}$'s scalability in terms of size and complexity of process descriptions. Expansion of elementary updates to domain specific updates are useful. In addition, we are planning to consider other capabilities of $\mathcal{TR}$ as a process representation tool. For example, we can study how $\mathcal{TR}$ can represent concurrent behaviors. In this way, we should consider how $\mathcal{TR}$ can encode other process specification conventions such as process algebra. For example, encoding process algebra's concepts and operational structural semantics in $\mathcal{TR}$ would enable it to act as a a theorem prover engine in the process algebra's domain.

References

1. Mccarthy, J., Hayes, P.J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Machine Intelligence. Volume 4. (1969) 463–502
2. Kowalski, R., Sergot, M.J.: A logic-based calculus of events. New Gen. Comput. **4** (January 1986) 67–95
3. F. van Harmelen, V.L., Porter, B.: Event Calculus. In: Handbook of Knowledge Representation. Elsevier (2007)
4. Baral, C., Gelfond, M. In: Reasoning agents in dynamic domains. Kluwer Academic Publishers, Norwell, MA, USA (2000) 257–279
5. Lin, F.: Embracing causality in specifying the indirect effects of actions. In: Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2. IJCAI'95, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1995) 1985–1991
6. Gelfond, M., Inclezan, D.: Yet Another Modular Action Language. In: Proceedings of SEA-09, University of Bath Opus: Online Publications Store (2009) 64–78
7. Inclezan, D., Gelfond, M.: Representing Biological Processes in Modular Action Language ALM. In: Proceedings of the 2011 AAAI Spring Symposium on Formalizing Commonsense, AAAI Press (2011) 49–55
8. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. Bioinformatics **20**(1) (January 2004) 15–22

9. Tran, N., Baral, C.: Reasoning about non-immediate triggers in biological networks. Annals of Mathematics and Artificial Intelligence **51**(2-4) (December 2007) 267–293
10. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. Journal of Logic Programming **17** (1993) 301–322
11. Lesprance, Y., Kelley, T.G., Mylopoulos, J., Yu, E.S.K.: Modeling dynamic domains with congolog. In: In Proceedings of the Eleventh Conference on Advanced Information Systems Engineering (CAiSE99) (Lecture Notes in Computer Science, Springer (1999)
12. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science **133** (1994)
13. : Flora-2 : Users Manual