Event Representation in Knowledge Systems With Context Hierarchies

Engin Çınar Şahin
May 16, 2006

Undergraduate Senior Thesis

Advisor
Scott E. Fahlman

Table of Contents

1 INTRODUCTION **3**

2 GOALS **3**

3 THE SCONE KNOWLEDGE BASE SYSTEM **3**

4 EVENT REPRESENTATION **5**

4.1 SIMPLE EVENTS 5
4.2 COMPOUND EVENTS 6
4.2.1 Sequential Events *7*
4.2.2 Alternative Events *8*
4.3 EXAMPLES OF EVENT ADDITION 9
4.3.1 The `new-event` Function *9*
4.3.2 Role Forms *10*
4.3.3 Compound Forms *10*
4.3.4 Basic Cooking Knowledge Base *11*

5 EVENT QUERIES **13**

6 FUTURE WORK **14**

7 CONCLUSION **14**

8 REFERENCES **15**

1 Introduction

What can we infer from the sentence "I made a pasta dinner"? What objects were used, who made the pasta dinner, what is meant by a pasta dinner? A powerful event representation together with the background knowledge is necessary to represent and reason about the meaning of such sentences.

Event and action representations have been explored in the planning field since the 1960s. Their motivation was to create representations that improve the performance and power of planning. Most representations were designed using the situation calculus.

The main goal of this thesis is to develop an event representation for the Scone knowledge system. Unlike common knowledge systems Scone is not a general theorem-prover, but a semantic network utilizing pseudo-parallel marking passing algorithms for efficient inference (Fahlman, 2006b). My goals in this project have been to design, implement and test an event representation that agrees with the precondition and effect axioms and frame axioms of the situation calculus (Brachman, 2004).

The representation is also aimed to be used by applications such as natural language processors and observation systems. The design decisions are influenced by the possible uses of the representation; however they are not bound to one specific application.

This paper covers a basic overview of the Scone knowledge base system, the event representation formalism, the implementation of the representation, a comprehensive example and possible extensions to the thesis.

2 Goals

The goals of this thesis are to:

- Design a formal representation of events, actions and plans for the Scone Knowledge Representation System
- Implement functions to easily add and query events in the knowledge base
- Implement various event knowledge bases to test and demonstrate the representation capabilities.

3 The Scone Knowledge Base System

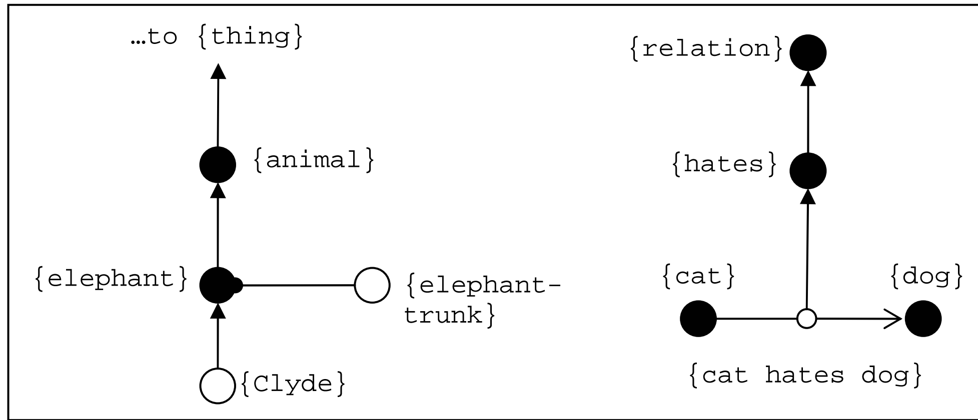
Scone (Fahlman, 2006a) is a high-performance knowledge base system consisting of a representation language and an inference engine. It is a multiple inheritance network with a powerful context mechanism (Section 1.1.) Only the basics of Scone will be introduced in this paper, avoiding the low level details of the system.

Scone represents knowledge as a semantic graph or network. Nodes represent entities (either individual or a typical member of a class). Every node in the system has a unique internal name which is surrounded by curly braces. We adopt the same syntax in this paper. Nodes are connected to other nodes via *wires.* There are parent, context, a, b and c wires for each node. Every node has a connected parent and context wire, which means that every node belongs to the type hierarchy and is true in some context (a state of the knowledge base). Some nodes represent statements between entities. Such nodes are called links and have their a, b and possibly c-wires connected (c-wires are for tertiary

relations.) For example, a "hates" link would have its a-wire connected to the hater, and the b-wire to the hated.

Figure 1, adapted from (Tribble, 2005) shows the Scone representations of "Clyde the elephant" and "Cats hate dogs".

Figure 1: Scone representation of "Clyde is an elephant" and "Cats hate dogs"



Since every node has a connected parent wire, every entity in Scone belongs to the type hierarchy. A child node inherits all knowledge from its parents including statements and roles, however explicit exceptions are allowed.

All entities and links exist in contexts. Contexts represent different states of the knowledge base; hence they may be used to represent hypotheticals, counter-factuals, opinions, or different world states, and play a crucial role in event representation. Contexts are represented as nodes in the network, so they belong to the type hierarchy as well. A child context inherits all knowledge from its parents, and alterations to the knowledge base in a context do not affect the knowledge in superior contexts. Such context hierarchies allow efficient representations of different knowledge base states.

For example, if we wanted to represent the Harry Potter World, we would make it a child of the general context, since most of general knowledge is still true in the HPW. We would then make necessary alterations to the knowledge base within the HPW such as adding wizards and witches.

Figures play an important role in explaining representations, and it's important to differentiate between the types of nodes in the network. I will be using the following patterns in the figures.

Internal Name	{iname}
Type Node	●
Individual Node	○
Node (type or individual)	▨
Context Node	△

Child to Parent Link	**Role to Owner Link**	**Equality Link**	**Statement S(A,B)**
▨ ——▶ ▨	▨ ——● ▨	▨ ◀——▶ ▨	▨ —S→ ▨

4 Event Representation

Historically event representations have been most commonly used in the planning field. The early representations usually focused on preconditions and effects of an event (Fikes, 1971). Later representations explored hierarchies of events (Sacerdoti 1974, Sacerdoti 1975, Bobrow, 1977). It must be noted that Scone is not a planner, but a general knowledge base system that may be used by other applications. The main goal of the representation is to include as much useful knowledge as possible so that planners and other software systems can use Scone as their knowledge representation system.

In Scone, the context mechanism may be used to represent snapshots of the world which may then be temporally related to represent the flow of time between them. Context hierarchies allow an event representation agreeing with both the precondition and effects axioms and the frame axioms of the situation calculus. The Scone framework also allows us to have hierarchies of events.

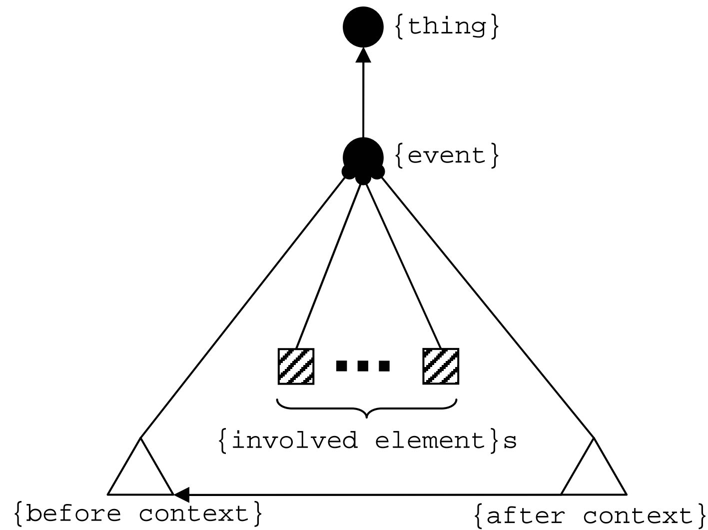
4.1 Simple Events

At a basic level an event is a thing that changes the state of the world. Events have a {duration}, a {start time}, an {end time}, a {location}, a {before context} and an {after context}. These are standard roles that each event has, and it is worth noting that these roles represent the properties of an event rather than the entities through which the event affects the world. It may be desirable to differentiate these two types of roles of events for certain applications, so we will make this distinction in the representation. The {involved element} role represents all entities through which the event affects the world. For example, {agent} and {food} roles would be {involved element}s of the {eat} event. These roles will be represented as square nodes in the figures.

The {before context} and {after context} roles represent the before and after states of the world of the event. All knowledge in the {before context} is true at the {start time} of the event, and all knowledge in the {after context} is true at the {end time} of the event. Since the {after context} comes after the {before context}, {after context} inherits all knowledge from the {before context}. In a planner point-of-view the {before context} contains the preconditions of the event, whereas the {after context} contains the effects. To create a temporal ordering between the {before context} and {after context}, there is a transitive {happens after} statement between them. Figure 3 shows the simple event representation.

Actions are special kinds of events. Every action has an {agent} role which represents the agent who causes the action to take place. So there's a {causes} statement between the {agent} and {action}. For example {speak} is an {action} where the {agent} of {speak} is an individual {human}. The {agent} of {speak} is causing the action, hence the {causes} statement makes sense.

Figure 2: Simple Event Representation



It is possible to automatically convert simple events into a STIPS-like syntax, where the preconditions are the statements in the {before context}, the add list contains newly introduced statements in the {after context}and the delete list contains cancelled statements in the {after context}. (Scone doesn't remove elements from the knowledge base, but uses cancellation to make knowledge elements untrue.)

Consider a {cook} event where there are two roles {agent} and {food to cook}. The {agent} is the agent which is doing the cooking, and {food to cook} is the food that is being cooked. The {before context} of {cook} would contain a statement saying that {food to cook} is raw (and therefore not cooked), whereas the {after context} would cancel that statement and contain a statement stating that {food to cook} is now cooked (Section 4.3.4.) This simple representation doesn't say anything about how the event took place, it only tells us what happened to what, when, and possibly who caused the event to happen. To represent the how knowledge we introduce compound events.

4.2 Compound Events

Events that decompose into subevents are called compound events. The decomposition of the compound event is a temporal structure of subevents. We shall refer to the overall structure of subevents as an expansion. A plan or a procedure can be represented as an expansion, and the compound owning the expansion would be the event that achieves the goals of the plan.

Regardless of the type of expansion, two aspects of the expansion must be coherent with the simple event representation. The first involves the roles of the subevents and the second involves the before and after contexts.

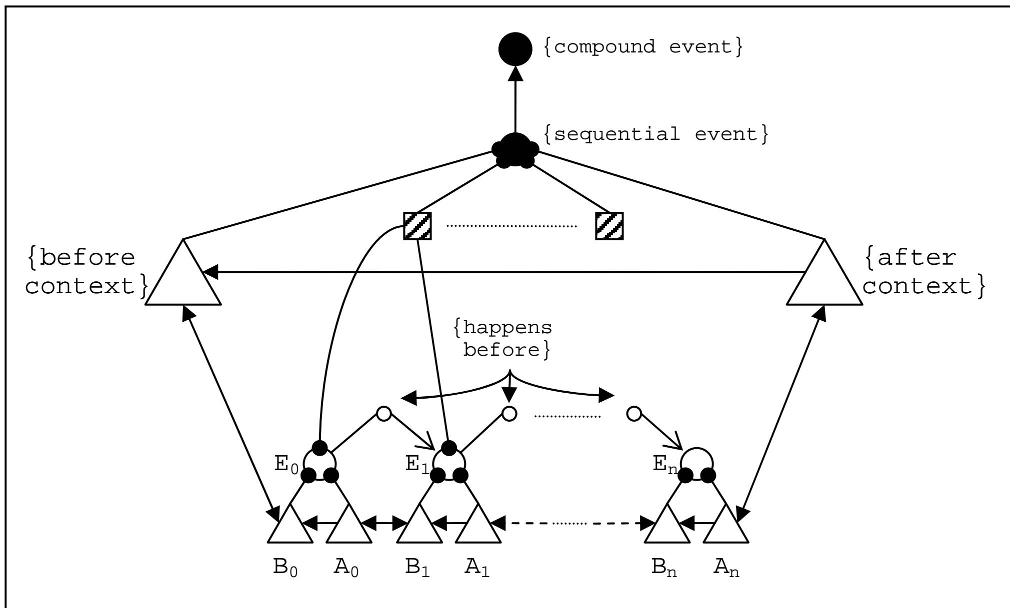
The compound event representation must allow certain {involved element} roles of the subevents to be unified in order to form a coherent representation. In other words, if we have the two subevents {peel potato} and {dice potato} of the compound event {make potato salad}, we would like them to refer to the same potato. It is also important to note that, it is not the representation's task to find possible unifications, but only to represent them. Hence the framework described here requires user input for role unification.

The second important piece in compound event representation is how the before and after contexts of a subevent relate to the contexts of the other subevents and to the ones of the compound event. It should be possible to infer the before and after contexts of the compound event given the subevents and also determine if the subevents are consistent with the compound event. How the representation handles the contexts depends on the kind of compound event. For the purposes of this thesis, only two types of compound events will be covered. Most common plans are partially ordered, where sequential segments may be branched to represent options or alternatives. The two types of compound events included in the thesis are sequential and alternative events.

4.2.1 Sequential Events

Sequential events have expansions that form a linear sequence. The event is accomplished by the happening of each of the subevents in order. Figure 3 shows the Scone representation of sequential events.

Figure 3: Sequential event representation



To make the expansion coherent with the sequential event, the representation must make sure what to link to the before and after contexts of the sequential event.

First, each subevent's after context is equated to the next subevent's before context. Note that due to inheritance the last subevent's after context (A_n) will inherit all knowledge from each context, hence it will contain the final state of the world. A_n is equated to the after context of the sequential event.

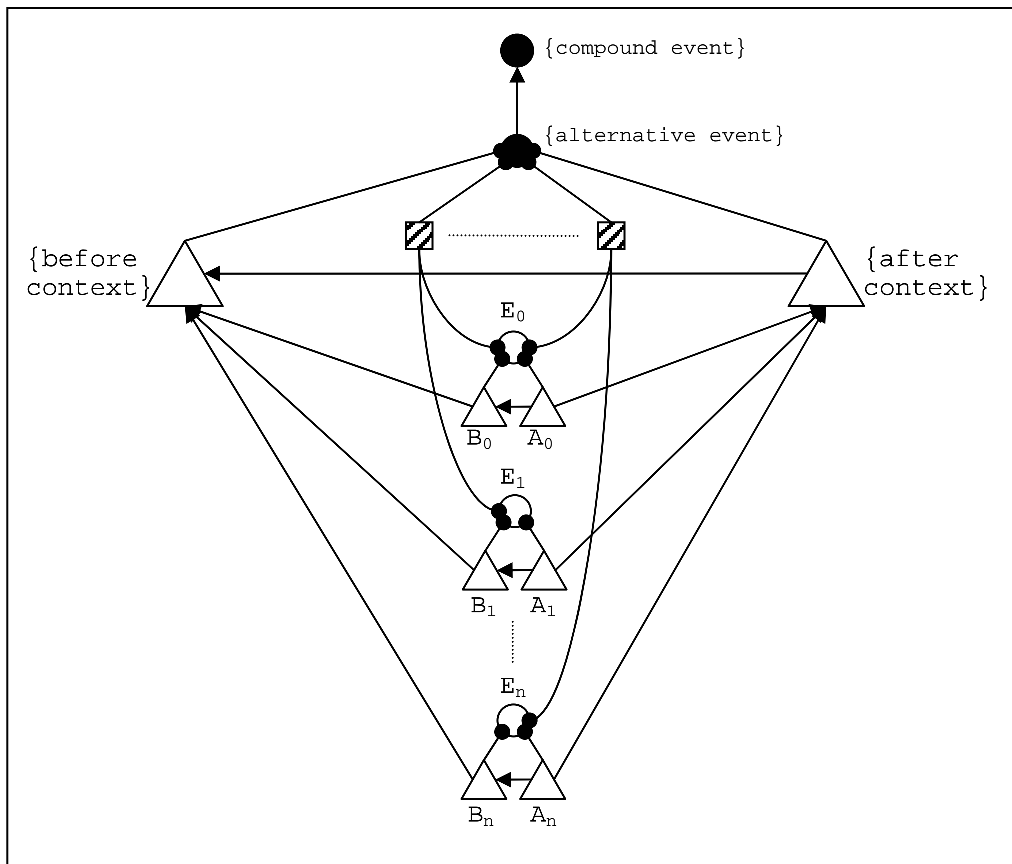
There's more inference needed to be done for the before context, however. The before context of the first subevent (B_0) only contains the preconditions of the first subevent. If any other subevent has a statement in its before context, that was not introduced by a preceding subevent, that statement should appear in B_0 as a precondition to the whole expansion. Imagine having a recipe where one slices a tomato (E_0) and then

puts the slices in a hot pan (E_1). Assume that E_1 requires the pan to be hot and it is not caused by E_0. In this case the pan being hot must be a precondition of the whole sequence and not just E_1. Therefore, during the addition of sequential events, the context contents are compared to find equivalent and canceling knowledge elements. The preconditions for the sequence are collected into B_0, which is then equated to the before context of the sequential event.

4.2.2 Alternative Events

Alternative events have their subevents as alternatives. The occurrence of any of the subevents accomplishes the main event. Alternative events allow branching when used in combination with sequential events. Such compound events may be used to represent partial order plans and procedures.

Figure 4: Alternative event representation



Since the subevents in alternative events do not rely on each other, no context weaving is necessary. The only required linking is making the before/after contexts of the subevents a type of before/after context of the alternative event. However note that this linking doesn't put anything in the before/after context of the alternative event. Therefore it is also necessary to intersect the before/after contexts of the subevents to find which statements are common preconditions and effects, and put them in the before and after contexts of the alternative event.

4.3 Examples of Event Addition

The previous section explained the semantics behind simple and compound events. To make the representation feasible, users should be able to easily create such events.

In order to make this task easy for the users a simple syntax was designed to make role unification and expansion declaration easy. Below the `new-event` function syntax, role forms and compound forms are described.

Figure 5 `new-event` syntax and usage

```
Function NEW-EVENT
Syntax:
new-event iname parent-list
          &key english type generic roles
               throughout before after expansion
=>
element

Arguments and Values:
iname       - an internal name for the event
parent-list - a list containing parents of the event
English     - a list of English names to be registered for event
type        - generalized boolean. Default: t
generic     - generalized boolean. Default: nil
roles       - a list of role-forms
throughout  - a list of forms
before      - a list of forms
after       - a list of forms
expansion   - a compound form

Description:
A new element is created with the given internal name (iname),
English names (english), and parents (parent-list). If type is
non-nil, the created element will be a type node. Otherwise, if
generic is non-nil, it will be a generic individual or if
generic is nil, it will be a proper individual.

Each role-form in roles is evaluated. All roles created with a
role-form becomes an {involved element} of the event. The
throughout forms are evaluated to add any knowledge about the
event or the roles of the event that hold throughout the event.
Then the before forms are evaluated in the {before context}.
Then the after forms are evaluated in the {after context}.

If compound is non-nil, the compound-form is evaluated and the
given expansion is added for the event.

Finally the node representing the event is returned.
```

4.3.1 The `new-event` Function

Adding new events to the knowledge base is done through the new-event function. It handles all of the following:

- the creation of the event and its roles
- the creation and population of the contexts
- the creation of subevents and weaving of subevent contexts
- role unification
- collecting of preconditions for sequential events

There are two convenience macros `new-event-type` and `new-event-indv` that supply the `:type` keyword argument with values of `t` and `nil`, respectively.

4.3.2 Role Forms

Role forms are designed to easily create roles for an event and to unify them with subevents' roles. (In EBNF form.)

(:indv *role-iname* *parent* [**:bind** *group*]**)**

This form creates a new individual role element for the event. The role is of type *parent*. If `:bind group` is supplied, the role is unified with the rest of roles in the same binding group.

(:type *role-iname* *parent* [**:bind** *group*]**)**

This form creates a new type role element for the event. The role is of type *parent*. If `:bind group` is supplied, the role is unified with the rest of roles in the same binding group.

(:rename *role-iname* *new-iname* **[:bind** *group***]***)*

This form sets the internal name of an inherited role. If `:bind group` is supplied, the role is unified with the rest of roles in the same binding group.

4.3.3 Compound Forms

Compound forms are designed to easily state an expansion together with role unification information. (In EBNF form.)

```
(:seq {subevent-iname
       | (subevent-iname {:bind role-iname group}*)}*)
```

This form is used to create a sequential expansion. If only *subevent-iname* is supplied a new generic individual event of type *subevent-iname* is created and included in the expansion. If any number of `(:bind role-iname group)` is supplied the *role-iname* of the newly created subevent is unified with the rest of roles in the same binding group.

```
(:or {subevent-iname
       | (subevent-iname {:bind role-iname group}*)}*)
```

This form is used to create an alternative expansion. If only *subevent-iname* is supplied a new generic individual event of type *subevent-iname* is created and included in the expansion. If any number of `(:bind role-iname group)` is supplied the *role-iname* of the newly created subevent is unified with the rest of roles in the same binding group.

Next section describes the creationg of a basic event knowledge base that demonstrates the use of `new-event, new-event-indv` and `new-event-type.`

4.3.4 Basic Cooking Knowledge Base

We would like to create a simple event hierarchy that contains cooking actions. We assume there is sufficient background knowledge in the cooking domain. The desired representation is shown in Figure 6a, 6b and 6c. The code necessary to built the representation is shown in Figure 7, 8.

Figure 6a: Example Event Hierarchy

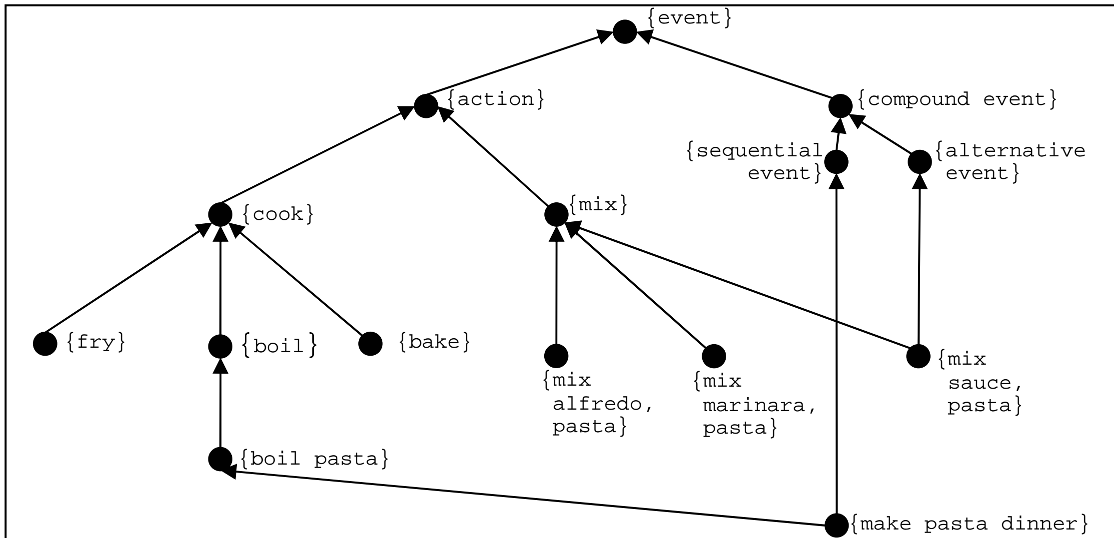


Figure 6b: {`make pasta dinner`} representation

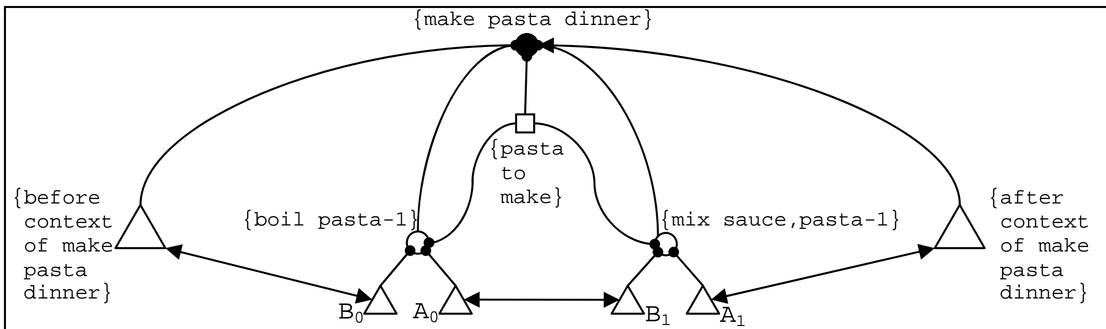


Figure 6c: {mix sauce,pasta} representation

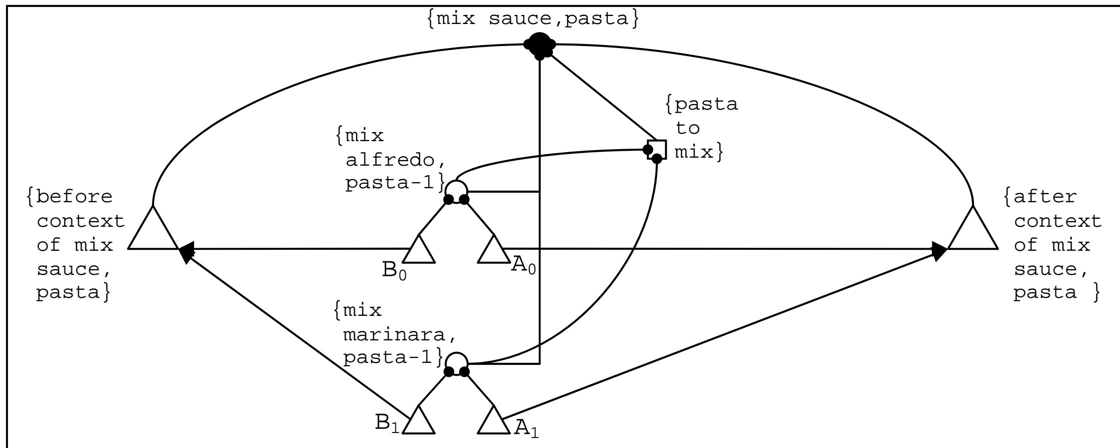


Figure 7: Example code for event addition (1)

```
(new-event-type {cook} '({action})
      :roles
      ((:rename {agent} {cooker})
       (:indv {food to cook} {food}))
      :before
      ((new-is-a {food to cook} {raw food}))
      :after
      ((new-not-is-a {food to cook} {raw food})
       (new-is-a {food to cook} {cooked food})))

(new-event-type {fry} '({cook})
      :roles
      ((:rename {agent} {frier})
       (:rename {food to cook} {food to fry}))
      :after
      ((new-is-a {food to fry} {fried food})))

(new-event-type {bake} '({cook})
      :roles
      ((:rename {agent} {baker})
       (:rename {food to cook} {food to bake}))
      :after
      ((new-is-a {food to bake} {baked food})))

(new-event-type {boil} '({cook})
      :roles
      ((:rename {agent} {baker})
       (:rename {food to cook} {food to boil}))
      :after
      ((new-is-a {food to boil} {boiled food})))

(new-event-type {boil pasta} '({boil})
      :roles
      ((:rename {food to cook} {pasta to boil}))
      :throughout
      ((new-is-a {pasta to boil} {pasta})))
```

Figure 8: Example code for event addition (2)

```
(new-event-type {mix} `({action})
      :roles
      ((:indv {edible 1} {edible})
       (:indv {edible 2} {edible}))
      :throughout
      ((new-not-eq {edible 1} {edible 2}))
      :before
      ((new-statement {edible 1} {not mixed with} {edible 2}))
      :after
      ((new-not-statement {edible 1} {not mixed with} {edible 2})
       (new-statement {edible 1} {mixed with} {edible 2})))

(new-event-type {mix alfredo,pasta} `({mix})
      :roles
      ((:rename {edible 1} {alfredo to mix})
       (:rename {edible 2} {pasta to mix}))
      :throughout
      ((new-is-a {alfredo to mix} {alfredo sauce})
       (new-is-a {pasta to mix} {pasta})
       (new-is-a {pasta to mix} {boiled food})))

(new-event-type {mix marinara,pasta} `({mix})
      :roles
      ((:rename {edible 1} {marinara to mix})
       (:rename {edible 2} {pasta to mix}))
      :throughout
      ((new-is-a {marinara to mix} {marinara sauce})
       (new-is-a {pasta to mix} {pasta})
       (new-is-a {pasta to mix} {boiled food})))

(new-event-type {mix sauce,pasta} `({mix})
      :roles
      ((:rename {edible 1} {sauce to mix})
       (:rename {edible 2} {pasta to mix} :bind 1))
      :throughout
      ((new-is-a {sauce to mix} {pasta sauce})
       (new-is-a {pasta to mix} {pasta})
       (new-is-a {pasta to mix} {boiled food})))
      :expansion `(:or ({mix marinara,pasta}
                        (:bind {pasta to mix} 1))
                       ({mix alfredo,pasta}
                        (:bind {pasta to mix} 1))))
```

5 Event Queries

Now we have the ability to add event knowledge into the knowledge base, the next step is to make sure that there are functional queries to make the representation useful. Events may be queried according to their types, roles, context contents and subevents. Type queries are already implemented in Scone on a general level. The queries I implemented are the following:

- **`(list-events-involving x)`**
  ```
  x       - internal name or an element
  returns - list containing events that have a role of type x
  ```

- **(list -events-requiring *s*)**
  s - internal name of a statement
  returns – list containing events that have a statement equivalent[1] to *s* in their {before context}
- **(list-events-causing *s*)**
  s - internal name of a statement
  returns – list containing events that have a statement equivalent[1] to *s* in their {after context}
- **(list-events-with-subevent *e*)**
  s - internal name of an event
  returns – list containing events that have a subevent of type *e*
- **(list-before *e*)**
  e - internal name of an event
  returns – list containing all knowledge elements in the {before context} of *e*
- **(list-after *e*)**
  e - internal name of an event
  returns – list containing all knowledge elements in the {after context} of *e*

6 Future Work

The thesis only covered two compound event types. One may investigate the possible representations of iterative and concurrent events. Another possibility is to have a {during context} that may contain knowledge elements that are true during the event. This would give more representation power especially to represent concurrency.

One may also investigate the possibility of interfacing Scone with a planner. Scone may be used to store and retrieve plans, make intelligent suggestions or observations with the extra background knowledge. It is also possible to interface Scone with an observation system to recognize events possibly from subevents and roles.

7 Conclusion

The goals of this thesis were to design, implement and test an event representation for the Scone knowledge system. Keeping in mind the possible applications that may use Scone; an event representation was designed based on the situation calculus.

I was able to implement simple events and compound events that exist in type hierarchies. I was also able to implement query functions for events using the marker mechanism. These representations may be used to represent plans and procedures allowing Scone to be used as a knowledge base for planners, natural language processors and observation systems.

This representation allows building event and plan hierarchies, however it is far from complete. More compound events may be included in the representation, and extensive event knowledge bases may be written to test the efficiency of Scone and the representation framework.

8 References

Bobrow, D. G., Winograd, T. (1977). An overview of KRL, a Knowledge Representation Language. *Cognitive Science, 1:1*.

Brachman, R. J., Levesque, H. J. (2004). *Knowledge representation and reasoning.* San Francisco: Elsevier.

Fahlman, S. E. (2006a). Scone user's guide. May, 2006, from http://www.cs.cmu.edu/~sef/scone/Scone-User.htm

Fahlman, S. E. (2006b). Marker-passing inference in the Scone knowledge-base system. *To be published in the proceedings of Knowledge Science, Engineering and Management, 2006.*

Fikes, R., Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, pp. 189-203.

Sacerdoti, E. D. (1974) Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 5,* 115-135.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior,* New York: Elsevier.

Tribble, A. (2005). A proposal for knowledge-based labeling of semantic relationships in English. *Thesis proposal.*