

Episodic Memory Representation in a Knowledge Base, with Application to Event Monitoring and Event Detection

Master's Thesis

Engin Çınar Şahin

Committee Members

Scott E. Fahlman

Eric Nyberg

Stephen F. Smith

Language Technologies Institute

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA, 15213

Abstract

The thesis explores the use of a knowledge-based AI system to assist people in executing procedures and detecting events in an observed world. We extend the Scone knowledge-base system with open-ended facilities for representing time and events. Then we use this episodic knowledge representation as the foundation for our event monitoring and event detection algorithms. This approach lets us represent and reason about three fundamental aspects of the observed events:

1. their ontological character and what entities take part in these events (e.g. *buying* is a kind of transaction that involves an agent, a seller, money and goods)
2. how events change the world over time (e.g. after a *buy* event the agent has the goods rather than the money)
3. how events may be composed of other subevents (i.e. a *buy* event may be composed of giving money and receiving goods)

We illustrate knowledge-based solutions to the event monitoring problem in the conference organization domain and to the event detection problem in the national security domain.

Acknowledgements

First, I would like to thank my advisor Scott Fahlman for letting me learn so much from him. His invaluable experience and constant support helped me go through the ups and downs of research. Working with him, his passion and excitement for knowledge representation and reasoning has passed on to me. I am grateful to Eric and Steve for helping bring depth and perspective to the thesis. I would like to thank everyone in the Language Technologies Institute for providing a unique and lovely academic environment. Also, I was lucky to have Mark Stehlik as my undergraduate academic advisor. He has been a friend as well as a great mentor, and I am forever indebted to him.

I am grateful to my family and grandparents for their unconditional love and support. I would like to thank my girlfriend Duygu for her encouragement and endless patience. Finally, I am indebted to my great friends, Ajay Surie, Adam Wolbach and Zhi Qiao, for making life fun and for being there for me when I needed them.

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract numbers NBCHD030010 and FA8750-07-D-0185. Additional support for Scone development has been provided by generous research grants from Cisco Systems Inc. and from Google Inc.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of DARPA or our other sponsors.

Table of Contents

1	Introduction	1
1.1	Thesis Overview	2
1.2	Problem Descriptions	3
1.2.1	Event Monitoring	3
1.2.2	Event Detection	4
1.3	Approach	7
1.4	Relevant Work	8
1.4.1	Time, Events and Procedures	8
1.4.2	Event Monitoring and Detection	12
2	Fundamental Knowledge and Reasoning	15
2.1	Standard Scone Functionalities	15
2.1.1	Scone Descriptions and Contexts	18
2.2	Time and Temporal World-States	20
2.3	Events	28
2.4	Future Extensions	35
3	Algorithms and Implementation	39
3.1	Event Monitoring	40
3.2	Event Detection	47
4	Experiments	51
4.1	Monitoring Experiment	54
4.2	Detection Experiment	61
4.3	Analysis	64
5	Conclusion and Future Work	67
5.1	Extending the Time Knowledge Base	68

5.2	Causes, Intentions and Goals	69
5.3	Executable Actions and Planning	70
5.4	Information Acquisition	72
5.5	User Interaction	73
5.6	Learning	74
A	Event Monitoring and Event Detection Helper Functions	79

Chapter 1

Introduction

The thesis explores the use of a knowledge-based AI system to assist people in executing procedures and detecting events in an observed world. We refer to these tasks as event monitoring and event detection, respectively. We extend the Scone knowledge-base system with open-ended facilities for representing time and events. Then we use this episodic knowledge representation as the foundation for our event monitoring and event detection algorithms.

In event monitoring, the system is responsible for monitoring the actions of an agent who has pledged to complete a procedure that is composed of subevents. For instance the pledged procedure may be *ordering lunch*, which may be composed of choosing what to order, entering a delivery address and submitting a payment. In turn, each of these steps may be composed of other subevents. In event detection, the system is responsible for recognizing what procedure is taking place in a world using the observations of the actions of an agent. For example, the system may observe a transaction of nuclear materials and the hiring of nuclear scientists which should lead the system to believe that a nuclear reactor may be under construction.

1.1 Thesis Overview

The thesis is contingent on the observation that an expressive and efficient knowledge-base system can drive the solutions to both problems and possibly other related ones. A complex AI system that is deployed in the real world will consist of many modules, each with its own knowledge requirements. An expressive and efficient knowledge-base system, such as Scone, can provide service to all of these modules to make the architecture more coherent and powerful. A module may be responsible for monitoring and detecting events, while others are responsible for information acquisition, learning and human-computer interaction. Each module may make use of whatever representational aspects they require using the common knowledge bases.

Following this observation, we extended the Scone knowledge-base with open-ended facilities for representing and reasoning about time and events so that Scone can act as a central component in a complex AI system. Specifically, the added facilities let us represent and reason about three fundamental aspects of events:

1. their ontological character and what entities take part in these events (e.g. *buying* is a kind of transaction that involves an agent, a seller, money and goods)
2. how events change the world over time (e.g. after a *buy* event the agent has the goods rather than the money)
3. how events may be composed of other subevents (i.e. a *buy* event may be composed of giving money and receiving goods)

These facilities are task-independent and can be used by any of the modules in a large knowledge-based architecture. The event monitoring and event detection problems are thought to be the responsibilities of a single module within such an architecture. We present knowledge-based solutions to these problems in the conference organization and national security domains, respectively.

1.2 Problem Descriptions

The problems of interest are best described using example use cases. We will assume the following scenarios and explore how a knowledge-based AI system can help the people involved. We will consider two scenarios: a person organizing a conference held at a university and a person trying to notice a sequence of events that are deemed dangerous. The conference organization domain was brought to our attention by the DARPA PAL program and the Radar project at Carnegie Mellon University.

1.2.1 Event Monitoring

Scenario 1: Organizing a Conference

There will be a conference held at a university, and Andrew is in charge of organizing a particular day of the conference. He will have to allocate rooms, equipment and order flowers, food and beverages for the guests. There are many steps of this task that an AI system can help with. A system can optimally schedule events so the minimum number of resources have to be allocated, or a system can automatically do simple but repetitive and time consuming tasks such as updating the conference website when the schedule changes. These systems no doubt will make Andrew more efficient, *if* Andrew knows about things that can go wrong outside those systems' responsibilities. If a certain conference room floods every time it rains or if the university equipment allocation system is unreliable and needs to be checked manually, the benefits of automation and optimization will diminish. Good organizers are distinguished from others by knowing more about their environment, which help them both make better decisions and anticipate problems. An AI system that is equipped with this kind of knowledge can benefit both experienced and inexperienced users. Experienced organizers may forget steps, where as inexperienced organizers have to learn all the steps either by making mistakes or from their co-workers. In either case, an AI system can assist the organizer by helping him/her engage that accumulated real-world knowledge. Assistants that aim to do so will undoubtedly need a flexible and efficient knowledge-base component. It has to represent and reason

about time, world-states, tasks, events and actions.

One capability of such an assistant is monitoring the execution of a procedure, which we will refer to as *event monitoring*. In event monitoring, Andrew first declares what procedure he is going to follow to accomplish a certain task. The procedure is composed of other actions, which may have their own procedures themselves. When the assistant starts observing Andrew's actions, it makes sure that the procedure is being followed correctly. If Andrew makes a mistake, the assistant should notice the problem immediately.

For example, Andrew may declare that he is about to order food and beverages for the conference by using the "order food" procedure. The assistant then starts observing Andrew's actions. While observing, the assistant should recognize when Andrew misses a step, or spends too much time at a step. Let's say, that Andrew starts ordering food before he knows how many vegetarians and vegans are attending the conference. The assistant should recognize that he missed this step in the procedure. If Andrew is taking too long filling out the order-form, the assistant should recognize the delay. Note that the assistant is only responsible for recognizing inconsistencies between the pledged procedure and the execution of it. Warning Andrew about the inconsistencies is the ultimate goal of the assistant; however, when and how to effectively remind the user are important user-interface questions that are beyond the scope of this thesis.

1.2.2 Event Detection

Scenario 2: Detecting A Threat

An intelligence agency is responsible for recognizing possible threats to the public. The agency has many highly trained field agents who gather intelligence and send them to the headquarters. Andrew is an analyst located at the agency headquarters and his task is to look over the gathered intelligence and recognize dangerous procedures. For concreteness let's assume he's responsible for recognizing bomb building procedures. The amount of intelligence that is flowing from field agents to the headquarters can be huge, so AI systems may be used to automatically detect potential threats. Statistical systems can analyze the data to find statistically dangerous events; however, our

focus is on deploying the knowledge a human uses in an AI system. For example, if the agency learns about a new procedure being used to build bombs, statistical systems will not be immediately effective. It would be desirable if the AI system could be told about the procedure in a seamless way. However, in order to be told about procedures, the system should be able to represent and reason about the world, materials, objects, time and events. It should also be scalable and efficient to be able to handle massive amounts of intelligence. Finally, it should be able to interface with natural language easily so that it can communicate with humans in an effective manner.

The capability we have been describing here is recognizing known procedures given observations of events in an environment. We will refer to this task as *event detection*. In event detection, the assistant receives basic, *relevant* information about events in the world. These actions are assumed to be done by the same agent as part of a single complex procedure. The event detection module is responsible for identifying the procedure the agent is executing as it receives information. For example, if the assistant was told that a government obtained nuclear coolant material, the assistant should keep track of procedures that involve that step. Later, if it receives information that the same government is hiring nuclear scientists, it should prefer procedures that involve hiring nuclear scientists. Andrew can make various queries to the assistant about the procedures that are likely to be happening in the world. If Andrew asks the assistant what may be taking place in the observed world, the assistant should bring up procedures that involve building nuclear reactors (which involves obtaining nuclear coolant and hiring nuclear scientists) among other possible procedures. There may be many such procedures such as “building nuclear weapons”, “building nuclear powerplants” or “building nuclear vehicles” and Andrew can ask the assistant what future actions to expect for each of these possibilities.

It is important to reiterate the limitations and assumptions about the information received by the event detection module. First, we limit the responsibility of the event detection module to observe only one agent (a group of actors working together can be treated as a single agent.) We assume that the observed agent is doing a single procedure. Allowing the detection of more than one procedure hurts the complexity of the detection algorithms, since each action of the agent can potentially

be the beginning of a new procedure or the continuation of older ones. This assumption lets us focus on the thesis, which is to illustrate a knowledge-based solution to the problem, rather than focusing on complexity issues. Finally, we assume a different algorithm with access to the very same knowledge facilities filters irrelevant actions. For example, if the observed agent eats lunch while building a bomb, the eating action is never sent to the event detection module. Depending on how event relevancy is judged (discrete or continuous) the event detection problem may become a binary selection problem or a ranking problem. Since we assume irrelevant actions are filtered out, we approach the event detection problem as a selection task rather than a ranking task.

With these limitations and assumptions, event detection and event monitoring become closely related problems. We can best illustrate these relations in the conference organization scenario.

Scenario 1: Organizing a Conference, Continued

It is unrealistic to assume that Andrew will always declare what precise procedure he is about to execute. This assumption may be reasonable for important, lengthy and complicated procedures, but Andrew will probably do smaller routine tasks without bothering to tell the assistant about it. Event detection can be used to initially guess what Andrew may be doing, and when a few possible procedures are left (or when the assistant is confident enough in the possibilities) the assistant can start the monitoring process and make notes of Andrew's mistakes.

Even if Andrew announced the procedure he is going to do, the pledged procedure may involve ambiguous steps. For example, if the 'order-food' procedure involves making a payment, and making a payment can be accomplished by a number of subprocedures (i.e. 'pay by credit card', 'pay by check' etc.) the assistant should be able to detect which subprocedure is followed. This is a version of event detection that is necessary for proper event monitoring. However, unlike event detection the assistant should be careful about when to dismiss a procedure. Andrew may change his mind in the middle of a certain branch (let's say 'pay by credit card') and switch to another one (say, 'pay by check'.) This change may not cause a problem in the overall procedure ('order food') and the assistant should be tolerant for these situations.

In both event monitoring and detection we assume that the assistant is being fed information about a world. Although we don't focus on how the information is acquired, it is fundamentally important to acknowledge and address this problem. We look into information acquisition and how it could be integrated with the presented approach in Section 5.4.

1.3 Approach

Event monitoring and event detection are two problems that require a high-level understanding of time and events. Also, solutions to these problems can only be realized when deployed in the real world as part of a larger architecture. Other modules in the architecture will have similar knowledge requirements. For these reasons, our approach focuses on the knowledge aspects of these problems.

The central knowledge component of a complex AI architecture must be expressive so that it can provide for the different needs of each module in the architecture. For example, in order to accurately represent the effects of events on the world, the knowledge-base system must be able to represent and reason about world states. Furthermore, the knowledge-base system must be efficient to handle large amounts of knowledge and queries made by the architecture. Unfortunately these requirements force us to give up on logical soundness and completeness on complex reasoning tasks. We prevent this from becoming a problem by making the knowledge-base system responsible for simple reasoning tasks. Scone is an expressive and efficient knowledge representation and reasoning system [13]. The requirements mentioned above make Scone a good candidate for a complex AI architecture. Scone provides simple and efficient reasoning mechanisms using marker-propagation algorithms [12].

We extended Scone with open-ended and task-independent knowledge bases that facilitate the representation and reasoning about time and events. We implemented solutions to the event monitoring and event detection problems that illustrate the use of the extended Scone knowledge-base system as a central library for knowledge requirements. We demonstrate the use of these solutions in simulated conference organization and national security situations.

Our report is structured around our approach. We first present relevant work on representing and reasoning about time, events and procedures. We then present relevant work on various versions of the event monitoring and event detection problems. We then describe the Scone knowledge-base system and the facilities we extended it with. We present the algorithms for event monitoring and event detection that use Scone as a central component. We conclude the thesis with the demonstrations of these algorithms in the conference organization and national security domains and the analysis of these demonstrations.

1.4 Relevant Work

We review the relevant work in the literature in two parts. First, we review work on representing and reasoning about time, events and procedures, followed by work on versions of the event monitoring and event detection problems.

1.4.1 Time, Events and Procedures

Representing and reasoning about time is a very old problem dating back at least to Aristotle. Over the years many different schemes have been developed that address this problem. One of the most influential of these schemes is the Interval Algebra [1]. Allen's work formed as the foundation for his work on time and events, and defines fundamental relations between intervals of time. The KANI time ontology [15] is a flexible and intuitive time ontology that builds on Allen's work. We were strongly influenced by the KANI time ontology; however, the time knowledge-base developed makes no attempt at mapping time references to precise numbers. Time references are kept completely symbolic and all comparisons and conversions are done symbolically.

Also, various temporal logics have also been developed including Linear Temporal Logic and Computational Tree Logic [26]. These logics use propositional formulae that are attached to modal operators, which act like temporal quantifiers. Rather than representing time (such as representing a certain day), these logics can represent a temporally dynamic world. Our knowledge-base is aimed

to provide a flexible representation for time itself (in terms of points and intervals) and builds on that to provide a comprehensive representation for temporal world-states.

Representing events and actions are very closely tied to representing time. Almost any attempt at representing time has been due to the goal of representing events. Just like time, events have been studied by philosophers, linguists and computer scientists, each focusing in different aspects of events. We present relevant work for representing and reasoning about events in three parts: *philosophy*, *linguistics and natural language processing* and finally *problem solving*. We have borrowed and adapted many ideas from each field to form a flexible and comprehensive representation.

Although the questions tackled in philosophy might appear to be too abstract for knowledge representation, the distinctions made by philosophers are invaluable and the terminology developed are useful. Philosophers make a distinction between static (*standing, waiting*) and dynamic (*moving, eating*) events [10]. Such a distinction has not been made in the computer science field. Both kinds of events can be modeled in a similar way without loss of generality. Another non-critical distinction is between mental and physical events [36]. This distinction was made by philosophers because of its ramifications to the philosophy of mind. In terms of the fundamental representation of these events, we do not consider them to be different; however, we note that in order to represent mental events, belief states need to be accurately modeled. Modeling belief states in Scone have been explored by Chen and Fahlman [6].

Actions, namely events that animate objects do, are considered as a subclass of events, and they have been closely analyzed by philosophers [2, 7, 39]; primarily because of the intentionality and causality of actions. The intentionality and causality of an action has effects on the action's meaning, and for this reason we pay special attention to their representation in Section 2.4; however, we leave that as future work since representing intentionalities is not critical to the event monitoring and detection problems.

Case grammar [17] is possibly the most influential work related to representing events for natural language processing. Case grammar introduced the concept of semantic roles for verbs and later led to frame semantics [18]. A frame is a representation for a certain situation including partici-

pants, props and conceptual roles. FrameNet [3] is an ongoing project that currently holds hundreds of such frames. Representing verbs and their semantic roles is not enough to describe the changes to world caused by the event. Using Scone's fundamental capabilities, one can create an ontology of verbs with semantic roles. Our goal in developing an events KB is to capture the deeper semantics of events: their temporal nature and how they affect a world.

Fillmore's concept of frames is different than Minsky's frame [31]. Minsky includes the pre-conditions and effects of an event in a frame. This information allows a lot more inferences to be made, and therefore is more complete. Our work on representing events is most related to Minsky's frames; however, there are certain differences in approach. Minsky's frame is a "data-structure for representing a stereotyped situation." Instead of using the word situation, we will use the phrase world-state to be precise that we mean a static description of a world. Scone can efficiently represent stereotypical world-states as special concepts called contexts. Contexts do play a central role in representing how events typically change the world. We describe in detail how the context mechanism works and how it overcomes 'the frame problem' in Chapter 2.

Other efforts involving verbs and events in natural language processing include PropBank [32] and VerbNet [38]. PropBank is a set of annotated documents where the annotations are verbal propositions and their arguments. An annotated verb only has arguments that exist syntactically, therefore a single annotation for a verb may have missing arguments. VerbNet is project that builds on WordNet [14] and PropBank to include more information on verb class memberships. All linguistic efforts that aim to provide semantic information about verbs do not attempt to represent knowledge or world-states in any manner. These projects are lexical resources that contain some semantic information; however, they are not knowledge representation systems, hence they do not provide any reasoning mechanisms that can be used elsewhere.

The problem-solving domain defines actions and events as world-state changing operators. The STRIPS language [16], perhaps the most commonly used language to define world-models for problem solving, was engineered to describe operators that changed the world-state. The original STRIPS language is a form of predicate logic, hence is not very expressive. The Action Description

Language (ADL) [33] and the Planning Domain Definition Language (PDDL) [30] were introduced to standardize problem-solving and planning definition languages. ADL extends STRIPS by allowing negative literals and disjunctives. PDDL contains ADL (and therefore STRIPS) and adds various features from restricted temporal logic. These languages remain in the first-order realm. ABSTRIPS [37] builds on the STRIPS planner (not the language) by using abstraction hierarchies in order to deal with more complex problem domains efficiently. Given a problem, ABSTRIPS solves a simplified version of the problem and deals with the details incrementally. This abstraction of the problem implicitly builds a hierarchy of events which is what simplifies the problem in the first place. For example, if the problem is to go from one room to the other with multiple closed doors on the way, the simplest solution will be to simply move from the current location to the goal, ignoring the doors. These ignored details are then addressed, at more specific levels. So, as the plan is generated a hierarchy of events is built. The ABSTRIPS work introduced event hierarchies (or plan hierarchies, but we don't make that distinction); however, the hierarchies are only built during the planning process and are not represented explicitly. The presented events KB explicitly represents and makes use of event hierarchies while providing higher-order constructs.

Another influential model of events is the situation calculus [29]. The representation is based on descriptions of the world at certain instants of time. Situation calculus is much richer than the STRIPS language. Events are represented as a set of preconditions that will hold before the event and effects that will hold in the final situation. Event calculus [25] differs from situation calculus in that situation calculus focuses on histories of world-states (i.e. situations) whereas event calculus focuses on events happening at certain times. Both are logical models and therefore are designed to be used in theorem-proving reasoning mechanisms. This inherently limits the expressiveness or the efficiency of the overall knowledge-based system. Also, both models do not focus on complex temporal reasoning, but only on queries about particular times points that are abstractly represented. They rely on separate representation of time references.

1.4.2 Event Monitoring and Detection

Unfortunately, event detection is an overloaded phrase, referring to slightly different tasks. One can encounter the phrase in the database, data mining and natural language processing fields. What is common over these fields is that the task is to find some pattern in time-series data.

Event detection and monitoring in the data mining field usually involves time series data with event occurrences at specific times [20]. This data is usually collected via sensors or system logs. Detecting frequent sequences of events [28] and decomposing a sequence of events into independent event subsequences [27] in such data are common tasks in this field. However, this kind of data has no information about what the events actually are or what they involve. For example, a series may have two *buy* events; one involving a soda, and one involving nuclear material. Unless these events are typed differently (such as *buy-soda* and *buy-nuclear-material*) no distinction can be made between these events. For real-world tasks, there can be thousands of different kinds of objects that may fill such roles and creating an event type for each possible filler is infeasible. Furthermore, only sequential processes are considered in these tasks, and other more interesting temporal relations are not considered.

Detecting event patterns in active databases has also been studied under the name of event detection [5]. This work is probably the most similar to the proposed thesis. The events that are monitored by an active database system can be composed into composite events. However, like the data mining field, here the data is usually in the form of event-type and time tuples. The omission of the roles of the events is not possible for real-world event detection.

Pedro [34] explored how information from multiple sources could be easily correlated and reasoned about in a coherent way. The HOUND system received information about the world in a language similar to propositional logic. The reasoning performed were in the form of production rules written for the domain of interest. A variant of the RETE algorithm was used to match and execute productions to reach conclusions. This work is similar to the event detection task; however, the information that flows from the environment to the system may include state information (e.g. “The glass is on the table.”) and productions may fire when such state information arrive. We’ve

limited our task to receiving event information to demonstrate how event detection can be easily done using generic knowledge-bases. Extending the event detection task to recognize general situations would involve more work on pattern recognition outside of the knowledge-based system. The episodic knowledge facilities presented in this thesis would provide the knowledge aspect of the pattern recognition problem, but the recognition algorithms would have to change considerably.

SHINE [24] is an expert system that can do detection and monitoring for NASA's Deep Space Network. Like most expert systems, the knowledge components used are task-specific and are not real-world oriented. For these reasons, they cannot be easily used by other systems. The presented work in this thesis can be viewed as a general framework that helps develop expert systems for real-world tasks. The developed knowledge-bases are task-unspecific and the algorithms implemented are domain-independent. Task specific knowledge-bases come into play to describe the domain of interest and the processes to be monitored or detected.

Chapter 2

Fundamental Knowledge and Reasoning

2.1 Standard Scone Functionalities

Scone is a semantic network composed of *elements* that are connected via *wires*. Elements are descriptions of actual concepts, objects, relations and statements. Wires connect the elements and allow markers to propagate between elements. The way wires are used to connect different element types is not very relevant to the thesis, so we will stay at a slightly higher level of abstraction. For details on how wires connect elements see [11, 12]. In this document we will use the `typewriter` face to indicate elements in Scone. Generally speaking, there are two kinds of elements in Scone: nodes and links. A node is an element that represents a standalone concept or object, such as ‘elephant’ or ‘Clyde’. We will use the lowercase (possibly capitalized) `typewriter` face to indicate Scone nodes. A link is an element that represents relations or statements between other elements, such as “Clyde is an elephant”, “Bertha is Clyde’s mother” or “Clyde loves Bertha”. We will use the uppercase `TYPEWRITER` face to indicate Scone links.

The most fundamental kinds of links are the `IS-A` link, the `CONTEXT` link and the `MAP` link. The `IS-A` link states that an element is a virtual copy (see Fahlman [11]) of another element. For example, if “Clyde `IS-A` elephant”, then Clyde is a virtual copy of the typical elephant, and therefore inherits all the properties of the typical elephant. Every element in Scone has at least

one IS-A link to another element. Cycles are permitted with the requirement that there must be an IS-A path from any element to the `thing` node, which is the root of the IS-A hierarchy. The concept `thing` has a single IS-A link to itself, fulfilling the IS-A hierarchy requirement. A symmetric IS-A link is called an EQ link, which semantically equates two elements.

The second kind of fundamental link is the CONTEXT link, which states that an element only exists because of another element. For example, if every `elephant` has an `elephant-trunk` (we will use a phrase like “`elephant HAS elephant-trunk`” for statements like this), then the concept of an `elephant-trunk` exists only because of the `elephant` concept. Every element in Scone has at least one CONTEXT link. Again, cycles are allowed with the requirement that there must be a CONTEXT path from any element to the `universe` node, which is the root of the CONTEXT hierarchy. The concept `universe` has a single CONTEXT link to itself fulfilling the CONTEXT hierarchy requirement.

If `Clyde IS-A elephant`, then Clyde inherits all the properties attached to `elephant`. This includes elements that exist only because of `elephant`, such as `elephant-trunk`. This means that Clyde will also have an `elephant-trunk`, but it will have *its own* `elephant-trunk`. In order to create Clyde’s elephant trunk concept, we create an element representing Clyde’s elephant trunk and then create a MAP link between `Clyde’s-elephant-trunk`, `elephant-trunk` and Clyde, simply stating “`Clyde’s-elephant-trunk is the elephant-trunk of Clyde`”. We can attach further properties to `Clyde’s-elephant-trunk` without affecting the original `elephant-trunk` concept.

Apart from these fundamental statements, if a user wants to state a statement like “Clyde loves Bertha”, s/he must first define the LOVES relation as a new link as well as the domain and range of this link. For instance, we can define LOVES as a relation between `elephant` instances and `elephant` instances. If we then want to state that Clyde LOVES Bertha, we create a new instance of that relation with Clyde on one side and Bertha on the other. If we want to create a negated statement, we simply do the same but flag the statement as a negation. We will use the term *statement* to refer to instances of relations.

Using SPLIT links, it is also possible to state mutually exclusive sets, like **African-elephants** and **Indian-elephants**. Finally, it is possible for any element to cancel any property about itself and state otherwise using CANCEL links. For example, if “the typical elephant HATES the typical mouse”, but Clyde doesn’t, then Clyde can cancel that statement without affecting other elements using a CANCEL link.

What makes Scone expressive is its basic and indiscriminating treatment of elements. An element may represent anything – an object, a statement, time, a world-state or a belief-state; and anything can be stated about that element. Therefore, Scone can represent and reason with higher-order constructs. Also, one can cancel any inherited property and state the negation of the property in Scone. This makes the representation non-monotonic. It has been shown that non-monotonic higher-order logics are not only provably intractable but provably undecidable [19]. Therefore, it is necessary to have special mechanisms that allow simple and fast reasoning. The reasoning mechanisms are described in more detail in Fahlman [11, 12].

The kinds of available reasoning mechanisms are:

- querying the IS-A superiors and inferiors of an element
 - “Is Clyde an elephant?”
 - “Show me all African elephants.”
 - “Can Clyde be an octopus?”
- querying the CONTEXT superiors and inferiors of an element
 - “Does Clyde have an elephant trunk?”
 - “How many legs does Clyde have?”
 - “Who is Clyde’s mother’s husband?”
 - “What elements have elephant trunks?”
- querying statements

- “Does Clyde love Bertha?”
- “Who loves Bertha?”
- “Who does Clyde love?”

This description of Scone is a very broad overview of how basic knowledge can be represented in Scone. In the following sections, we will describe how complex representations of world-states, time, temporal world-states and events can be built. Even when dealing with these complicated knowledge structures, we will show how the simple reasoning mechanisms will suffice for quite complicated tasks.

2.1.1 Scone Descriptions and Contexts

One important aspect of Scone is its context mechanism. Among other uses contexts may be used for representing world states at a particular time. The time knowledge base that we describe in the next section uses this mechanism for this purpose, so it is important to describe what contexts are and how the mechanism works.

Contexts are elements that represent sets of knowledge that are *active* together. By an active element, we mean an element that can partake in the reasoning mechanism and legitimately pass markers to other *active* elements. For example, if we want to reason about Clyde’s-trunk, the concept Clyde must be active. This is quite intuitive, if Clyde didn’t exist, his trunk would never exist, or if Clyde’s trunk exists, then Clyde must exist. The CONTEXT hierarchy in Scone describes the dependencies for being active between elements.

In understanding the active element idea, it is important to note the difference between semantic and physical existence. An element may or may not physically exist, but if it exists semantically it must be active in order to be reasoned about. For example, even if Clyde was born without a trunk, as an elephant he conceptually would have a trunk. The only significance of his trunk would be its physical non-existence. So the following statement “Clyde’s trunk never grew” should be assertable. We should also be able to ask about his trunk: “Can you see Clyde’s trunk?” So

regardless of physical existence, if an element in Scone is not cancelled, then that element requires its *context element* to be active in order for it to be active. Clyde's-trunk requires its context element, Clyde, to be active. So what is Clyde's context-element?

When we traverse the CONTEXT hierarchy upwards we notice we reach semantically isolated and self-coherent concepts and individuals such as Clyde (as opposed to Clyde's-trunk, or Clyde's-trunk's-nostrils.) Clyde exists by himself, but he exists in some location during some time or perhaps in someone's imagination. Let's assume that Clyde the elephant really exists in the real-world and you can sometimes find him standing in Scott's office. Then we can assert that Clyde exists in the real-world context. In turn, we can assert that the real-world exists in the universe context. Since the universe always exists (as it is the root of the CONTEXT hierarchy) we will have completed the activation procedure and now can reason about Clyde.

An important distinction that was not made in the previous section is between concepts such as Clyde and concepts such as real-world or universe. We suddenly jumped from standalone descriptions of real entities or concepts to areas of knowledge and truth. We will refer to the former as *descriptions* and to the latter as *contexts*.

It is important to note that elements that are connected to contexts via CONTEXT links are interpreted as *contents*. On the contrary, elements that are connected to descriptions are usually interpreted as *properties, roles or parts* of the description. So contexts inherit contents of other contexts through virtual copy links, while descriptions inherit roles and properties. For instance, if we assert that the Harry-Potter-world is a virtual copy of the real-England context, then everything true in the real-England context becomes true in the Harry-Potter-world. So, in the Harry-Potter-world the typical broom would be a cleaning-artifact by default. We could change this by asserting that the typical broom IS-A air-vehicle in the Harry-Potter-world. The virtual copy mechanism when used in conjunction with contexts creates a very efficient method for representing and reasoning about world-states, since it seamlessly copies the contents of contexts. Knowledge engineers should be careful when using this behavior. If some elements of a context are supposed to be interpreted as roles or properties, those concepts will also be inherited by vir-

tual copies of the context. For example, if we attached a time property to the `year-2006-context` describing the time the context held true, the time property would be inherited by any other context that was a virtual copy of the `year-2006-context`. So, if we wanted the `year-2007-context` to inherit everything from the `year-2006-context`, the time property would also be inherited. This property would have to be cancelled and re-asserted for the `year-2007-context`. Instead of cancelling and re-asserting properties and roles of contexts, we've decided never to assign properties or roles in the time knowledge bases. So instead of contexts having a time property, references to time will have a context role representing the world state at that time.

2.2 Time and Temporal World-States

Representing and reasoning about time is a very well studied field in KR [1, 23, 26]. The time knowledge base we describe here is a selection and adaptation of many ideas from this literature. The goals of the developed Scone time knowledge-base are as follows:

- referencing time as points or intervals, interchangeably
- referencing time at arbitrary levels of precision
- comparing any two time references, allowing for ambiguity and differences in precision
- using different time standards and calendars
- allowing ambiguous units of time, such as month or year

The time knowledge base, as designed, tries to be as comprehensive as possible without restricting the user too much. It is loosely modeled after the KANI time ontology [15], which in turn has its roots in Allen's work [1]. The knowledge base is also compatible with the DAML time ontology, which was developed for the semantic web [23]. Scone, in general, aims to have a powerful natural language interface and the time KB is designed to easily integrate with temporal natural language

interfaces like the one presented by Han and Lavie [21]. Many design choices were made with the goal to make the KB more compatible with linguistic phenomena related to time.

We will first describe the fundamental concepts and the temporal relations defined in the time KB. Then we will go over how calendar and clock standards are implemented, and how these can be used to make actual references to time (such as dates and times of day.) We will finish this section by describing how temporal world-states are represented and how they work together with temporal relations to inherit knowledge from each other.

The two most fundamental concepts in the time KB are `time` and `time-reference`.

To represent temporal units and measures, the units KB, which was developed by the author in earlier work, was used. The units KB allows the definition of measurable qualities, units for such qualities, conversions between units and measures of such qualities (number-unit tuples.) Using the units KB, we have modeled `time` as a `measurable-quality`, with various `time-units`, such as `hour-unit` and `second-unit`. We also defined ambiguous units like `month-unit` (which can be 28, 29, 30 or 31 days) and `year-unit` (which can be a normal or a leap year.) For non-ambiguous units we have defined conversion relations. Figure 2.1 shows the defined units in a hierarchy. It is important to note that these units are not related to a particular time system, and should not be confused with defined intervals, which we will describe later. These are commonly used units for measuring durations.

A `time-reference` represents a reference to a specific time, and has two subtypes: `time-point` and `time-interval`. Any `time-reference` has two roles: `time-start` and `time-end` which are both `time-points`. A `time-point`'s `time-start` and `time-end` roles are EQ to each other, and EQ to the `time-point` itself. The `time-start` role of a `time-interval` is BEFORE the `time-end` role of that `time-interval` (the BEFORE relation is defined later on.) So, every `time-interval` has a `duration` property which is an instance of `time-measure`. Figure 2.2 shows the relevant Scone code for this part of the KB.

Time relations are defined between time references. We've adapted the Allen time relations for Scone [1]. The Allen relations are purposefully disjoint, which is somewhat restricting for the user.

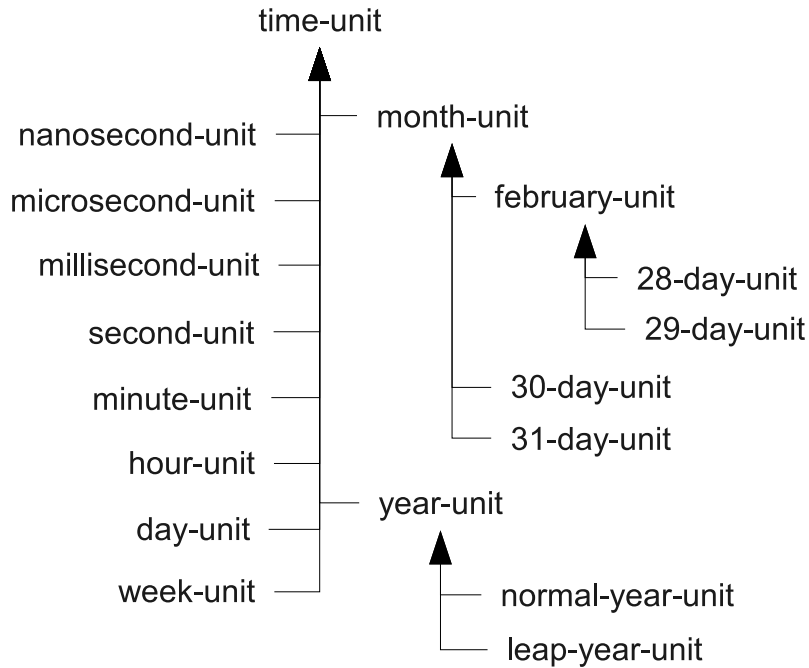


Figure 2.1: The time units defined in the time KB.

```

(time TYPE-OF measurable-quality)
(time-reference TYPE-OF intangible-thing)
(time-point , time-interval TYPE-OF time-reference)
(time-reference HAS-A time-start IS-A time-point)
(time-reference HAS-A time-end IS-A time-point)
((THE time-start OF time-point) EQ (THE time-end OF time-point))
((THE time-start OF time-point) EQ time-point)
((THE time-start OF time-interval) BEFORE (THE time-end OF time-interval))
(time-interval HAS-A duration IS-A time-measure)

```

Figure 2.2: Basics of the time KB. TYPE-OF is just like an IS-A link but additionally states that the first element is a type node not an instance.

We've defined other non-disjoint relations to form a relation hierarchy. One important reason for this is linguistic compatibility. For example, the statement "Monday is before Tuesday" corresponds to the Allen *meets* relation, i.e. Monday MEETS Tuesday, not to the Allen *before* relation, because the Allen *before* relation requires there to be a duration between the two time intervals. Also, Allen's work (the *Interval Algebra*) is only concerned about intervals and don't include points. In the time KB, if a relation naturally applies to time points as well as intervals (BEFORE, for example) then the relation is defined between time-references. Figure 2.3 shows the relevant Scone code for the definition of the time relation hierarchy. In some cases the defined relations are the inverses of the original Allen relations (CONTAINS-ALLEN, STARTED-BY, FINISHED-BY.) This is to make the relation hierarchy more interesting and structured.

The time KB is aimed to be as extensible as possible. For this reason, the fundamental concepts in the KB are not connected to any time system. However, in order to make actual references to time ("Monday the 12th") we need to define a time system: a calendar and clock standard. Once a time system is defined it may be related to the definition of other time systems so that conversions between two systems become possible. For the purposes of this thesis defining the Gregorian calendar standard with the standard 24-hour clock system was sufficient.

Every calendar system has an epoch time point that is used as the ultimate anchor point of any time reference. For the Gregorian system, the epoch point is the suspected birth year of Jesus Christ. The calendar itself is a division of time into intervals. Each of the divided intervals, which we refer to as *defined intervals*, are indexed starting from the epoch to make referring to them easy. The year 2006 C.E., for example, is the 2006th year after the epoch in the Gregorian calendar. That interval itself is divided into subintervals (months, days, weeks), each with their own indexes relative to the start time of the encompassing interval. This fractal referencing of time can indeed go indefinitely; however for the purposes of the thesis we defined intervals between the century and second levels.

A defined-interval, being a time-interval, has a duration property. The time units we have defined earlier are used to fill this property of defined intervals, and if a certain defined interval is not fully described (such as a defined year interval, but we don't know which year) ambiguous

```

; BEFORE: end(X) <= start(Y)
(RELATION (INST-OF time-reference) BEFORE (INST-OF time-reference))

; BEFORE-ALLEN: end(X) < start(Y)
((RELATION BEFORE-ALLEN) IS-A BEFORE)

; CONTAINS: start(X) <= start(Y) AND end(X) >= end(Y)
(RELATION (INST-OF time-interval) CONTAINS (INST-OF time-reference))

; CONTAINS-ALLEN: start(X) < start(Y) AND end(X) > end(Y)
(RELATION (INST-OF time-interval) CONTAINS-ALLEN (INST-OF time-reference))

; STARTED-BY: start(X) = start(Y) AND end(X) > end(Y)
((RELATION STARTED-BY) IS-A CONTAINS)

; FINISHED-BY: start(X) < start(Y) AND end(X) = end(Y)
((RELATION FINISHED-BY) IS-A CONTAINS)

; OVERLAPS: start(X) < start(Y) AND end(X) > start(Y) AND end(X) < end(Y)
(RELATION (INST-OF time-interval) OVERLAPS (INST-OF time-interval))

; MEETS: start(X) < start(Y) AND end(X) = start(Y)
(RELATION (INST-OF time-interval) MEETS (INST-OF time-interval))

```

Figure 2.3: The time relations in the time KB. RELATION asserts that a relation is being defined. The domain and range are inherited in cases like BEFORE-STRICT, where they are not explicitly defined.

```
(NEW-TIME-POINT :name Clyde-bday :year 1974 :month 9 :day 6)
(GET-NOW :name thesis-time)
(NEW-TIME-INTERVAL :name Clyde-life :start Clyde-bday :end thesis-time)
```

Figure 2.4: Convenience functions for creating time points and intervals. The GET-NOW function asks the operating system the time, and converts it to a Scone time-point.

time units can fill the duration property.

Making time references involve creating a defined interval with a certain precision level. For example, a date such as “September 6th, 1974” can be represented as defining the 1974CE defined year, the September defined month in that year and the day-6 defined day of that month. We can also define dates with missing information. For example, if we just wanted to refer to “September 6th”, we would create a generic defined year interval and then define its subintervals. In the time KB, this is precisely how one can refer to a point in time: one creates a `time-point` element and fill the defined intervals roles of that time point. The actual code for defining a time point in this way is not short, since each subinterval defined needs to be connected to its owner, and each defined interval’s index needs to be assigned as well. The time KB provides convenience functions that make defining a time point or a time interval quite easy. Figure 2.4 shows example code defining two time points and an interval between them.

When a `time-interval` has its `start-time` and `end-time` roles filled with `time-points` that are described by defined intervals, one can calculate its `duration`. For example, `Clyde-life` in the above example is such a `time-interval`, so we can calculate its `duration`. Currently, the calculations are done in Lisp, since that level of reasoning is beyond simple marker-propagation procedures; however, with the implementation of procedural-hooks (i.e. `if-needed` and `if-added` hooks similar to those of PLANNER [22]) and a simple Scone-based planner one can completely embed the calculations in Scone (see Section 2.4 for more on this topic.) Similarly query functions for temporal relations between such time references are also implemented in Lisp and not embedded into Scone.

Note that the way we have defined references to time is mutually recursive. A `time-interval` is described by two `time-points` and a `time-point` is described by a `defined-interval`, which is a `time-interval`. This mutual dependency leads to a very convenient way to switch from interpreting a reference to time as a point to interpreting it as an interval and vice versa. For example, if we want to take `Clyde-bday` and interpret it as an interval and not as a time point, we have to do only two changes. First, we have to cancel the IS-A link from `Clyde-bday` to `time-point` and create a new IS-A link to `time-interval`. Finally, we have to change the MAP link between `Clyde-bday`, `Clyde-bday's-defined-interval` and `time-point's-defined-interval` to an EQ link. This change basically follows the logic that `Clyde's-bday` now is a `time-interval`, so it doesn't have a `defined-interval` describing it. Instead, that `defined-interval` should be semantically equivalent to `Clyde-bday`. We do the same changes in reverse to convert a `time-interval` into a `time-point`. Additionally we have to take care of two more things. Since `time-point's start-time` and `end-time` are EQ to each other, we have to make sure that if these roles are filled, we have to cut the granularities of these points until we can EQ them. For example, if we wish to convert a time interval defined between “Monday, 3pm” and “Monday, 7pm”, we cancel the hour-level information of the start and end times making both reference “Monday.” The last thing we need to take care of involve temporal relations. Some temporal relations are only defined for `time-intervals` and if there is a statement of this kind involving a time interval we want to switch the interpretation of, we should attach the statement to the `defined-interval` describing the reference rather than the reference itself.

Certain representations of time don't have a time point concept (i.e. Allen's work [1]) but we included the concept and therefore had to support switching the interpretation of a reference. The reason for this is mostly linguistic compatibility. For example, one might say “The meeting was at 3pm” which suggests a point interpretation to that time frame. The same person could later say “And it lasted for 3 hours!”, which involves a switch of perspectives from a point to an interval with a duration. We also find it more intuitive to have time points. Many people think about time as a linear dimension and therefore refer to points on that line or intervals [35]. The correctness of this

picture is not relevant to the discussion here; however, the fact that many people think this way is. Including the time point concept as a kind of time reference is not more restrictive or less expressive in any way. In fact, we hope it makes the time KB more intuitive and flexible.

`time-references` and the relations described in the beginning of the section provide the framework that we used to model temporal world-states. Every `time-reference` has a `time-context` role that represents the world at or during that time. So if a statement is true in a `time-context` then that statement is true *throughout* that time. Depending on the relations between a `time-reference` and other `time-references`, the `time-contexts` will inherit knowledge from each other. There happens to be a single inheritance rule, shown in Table 2.1 which follows the order of the arguments of the relations. Some of these cases are quite intuitive. For example, if slavery is illegal in the year 2008, then slavery is illegal in any time during 2008, so for all `CONTAINS` relations, knowledge flows from the container to the contained. Some rules are for convenience only. For example, for `BEFORE` relations we would like to copy the knowledge to later times, so we don't have to reassert everything. We can; however, change certain things in the new time-context using `CANCEL` links. All of these rules are realized by creating virtual copy links from the inheritor context to the other. The virtual copy links are created only when a temporal relation is asserted or the truth of a statement is queried. So if two time points are created independently and one is indeed `BEFORE` the other, a virtual copy link will not be created between the `time-contexts` until the statement is queried or the statement is explicitly asserted. If we actively updated the temporal world-state virtual copy

Statement			Inheritance Rule (how knowledge flows)
x	BEFORE	y	
x	BEFORE-ALLEN	y	
x	CONTAINS	y	
x	CONTAINS-ALLEN	y	<code>context(y) IS-A context(x)</code>
x	STARTED-BY	y	<code>(context(x) --> context(y))</code>
x	FINISHED-BY	y	
x	OVERLAPS	y	
x	MEETS	y	

Table 2.1: Temporal world-state inheritance rules for temporal relations.

network whenever a new time reference was added to the KB, we would have to keep track of $O(n)$ time references and compare the added reference to each of these (assuming there are n time references in the KB.) The comparison of time references is linear in the defined-interval granularity, so we can assume it is constant time for the time KB, but it is a high constant factor. The temporal world-state mechanism is used most in the events representation and we can shortcut the inheritance rules by defining them at a high level of abstraction. This way the `time-contexts` will inherit from each other due to inherited virtual copy links rather than waiting for a query or an assertion of their relations. This is explained in more detail in the next section.

2.3 Events

Representing and reasoning about events are very important problems in AI and they have been studied in various fields within AI with different perspectives. Planning, scheduling, natural language processing, cognitive modeling fields have all studied the problem with various goals in mind [29, 22, 16, 37, 25, 1]. Our goals in modeling events are similar to those of the time KB: to have a comprehensive model that may be used in many different applications with different tasks. To be concrete, the goals of the events KB are:

- to model event hierarchies
- to model typed semantic roles of events (things involved in events)
- to model the temporal aspects of events
- to efficiently model processes and procedures for events

Scone's most basic capabilities already let us fulfill the first two goals. We can easily create event hierarchies using the `IS-A` hierarchy and define typed roles for events using the `CONTEXT` hierarchy. The key contributions of the events KB are modeling events as temporal entities and modeling events as processes and procedures.

Events are intangible temporal and spatial entities; in other words the predicate “anything that happens at or in some place over or at some time” can be used to define what an event is. We have avoided the spatial aspects of events in order to bound the thesis. Spatial representation and reasoning is a difficult problem by itself and should be left as future work. In Section 2.4 we go over the necessities and difficulties tied to the spatial aspects of events.

The event concept is the central concept of the events KB. It represents the typical event. Every event has a `time-interval` role called `event-time` that represents the time the event happens. The `start-time` and `end-time` of `event-time` are called the `event-start` and `event-end` for convenience. The start and end times of events play an important role in the KB, as their contexts represent the world at the very beginning and at the very end of the event. We will refer to these contexts as the `before-context` and the `after-context` of the event respectively. Because the `before-context` and `after-context` are the `time-contexts` of the start and end time of a time interval respectively, the `after-context` inherits all knowledge from the `before-context`. This way the after context needs to contain only the changes that the event causes to the world and nothing else. This is a convenient and efficient solution to the famous frame problem [31].

The `before-context` can also be interpreted as the set of preconditions of an event. Similarly, one can interpret the `after-context` as the effects of the event. These interpretations are most appropriate in a planning perspective. We consider using the events KB in a planning task in Section 2.4.

In literature events and actions are sometimes treated differently [2]. The reasons behind this are aspects such as intentions, goals, causality and executability. These aspects raise important representational problems that are beyond the scope of this thesis. We provide a simple exploration of how these aspects can be modeled and used in the events KB in Section 2.4. For the purposes of the thesis, an `action` is just a subtype of `event` that has an `agent` role.

Events involve other objects and usually these objects are used to change the world-state or they are changed themselves. For example for the `walk` action, the location of the agent changes. To conveniently define events such as this one the events KB provides functions for creation of events.

Figure 2.5 shows the syntax of `NEW-EVENT-TYPE` which allows defining a new event type. In this example, the `walk` action is being defined as a subtype of `action`. First we define the roles for the `walk` action. The `agent` role is inherited through `action` and we declare that we're going to refer to it as `walker`. Then we define two new roles `walk-from` and `walk-to`. Both of these roles are `location` instances. Now that the role variables are defined (these are the Lisp variables `walker`, `src` and `dest` that can be used in the rest of the function to easily refer to the concept nodes themselves) we can now describe the contexts. First, we say that in general the `agent` of `walk`, i.e. `walker`, IS-A `legged-animate`. This statement will hold in general and throughout the event. We also assert that before the event happens the `walker` is at the `walk-from` location of `walk`, i.e. `src`. Since the `after-context` of any event inherits all knowledge from the `before-context` of that event, we have to cancel statements that are no longer true. For the `walk` action, the `walker` is no longer at the `src` location. Rather it is at the `dest` location. This completes the definition of the `walk` action.

Now that we have defined an interesting event type, we can create instances of this event. Figure 2.6 shows the syntax of `NEW-EVENT-INDV` and definitions of three different `walk` events that Clyde did. The first one started at `Newell-Simon-atrium`, but we don't know where he walked to. The second one started somewhere unknown and ended at `Wean-8th-floor`. We also state that Clyde is tired after this walk as he went up three stories. The last walk event started somewhere unknown and ended at `Scott's-office`. Not knowing a particular role of an event doesn't have any effect on our reasoning capabilities. When asked where Clyde was after the second event, Scone should correctly respond `Wean-8th-floor` since the earlier knowledge about Clyde's location should have been canceled.

The events KB also provides a simple interface to unify roles of two events. For example, if we later learn that Clyde's second walk started where his first walk ended we need to unify those roles. Figure 2.7 shows the `UNIFY-EVENTS` function with example calls unifying the unknown roles from the earlier example. Unification is a general reasoning problem and theoretically it can be handled automatically; however it has been shown that for second-order and higher-order logics unification

```
;; Syntax
(NEW-EVENT-TYPE name parents-list
  { (var-name role-name role-parent) | (var-name role-name) }*
  [ (:ALWAYS statement* ) ]
  [ (:BEFORE statement* ) ]
  [ (:AFTER  statement* ) ] )

;; Example
(NEW-EVENT-TYPE walk (action)
  ;; Roles
  (walker  agent)
  (src      walk-from  location)
  (dest     walk-to    location)
  ;; Contexts
  (:ALWAYS
    (walker IS-A legged-animate))
  (:BEFORE
    (walker LOCATED-AT src))
  (:AFTER
    (walk CANCELS (walker LOCATED-AT src))
    (walker LOCATED-AT dest)))
```

Figure 2.5: Convenience function for creating event types.

```
;; Syntax
(NEW-EVENT-INDV name parents-list
  (var-name role-name [filler-name])*
  [ (:ALWAYS statement* ) ]
  [ (:BEFORE statement* ) ]
  [ (:AFTER statement* ) ] )

;; Examples
(NEW-EVENT-INDV walk1 (walk)
  (cl agent Clyde)
  (from walk-from Newell-Simon-atrium))
(NEW-EVENT-INDV walk2 (walk)
  (cl agent Clyde)
  (to walk-to Wean-8th-floor)
  (:AFTER (cl IS-A tired)))
(NEW-EVENT-INDV walk3 (walk)
  (cl agent Clyde)
  (to walk-to Scott's-office))
```

Figure 2.6: Convenience function for creating event instances.

is undecidable [9]. For this reason, providing a general automatic unification mechanism is not very useful. Instead, the events KB tries to make unification convenient, so other specialized applications can unify roles easily. For example, a natural language interface may process sentences and send the extracted knowledge to Scone. As it is processing sentences, it may keep track of a stack of concepts extracted, so that when an anaphora is encountered the stack can be checked for the first compatible element and unified with the anaphora. This may or may not be an effective mechanism, but the last step should be simple to do.

The temporal relations defined in the time KB are all possible relations that can hold between time references, and each relation is assertable for events as well. For example, one may assert that Clyde started whistling 5 minutes into his first walk and stopped 2 minutes before the walk ended. This statement involves relating the events' `event-start` and `event-end` times and the forming the correct virtual copy relations between the before and after contexts. The events KB provides simple functions to assert these relation for events. The syntax of these functions is shown

```
;; Syntax
(UNIFY-EVENTS  event-1 event-2 ... event-n
                (role-1  role-2  ... role-n))

;; Examples
(UNIFY-EVENTS  walk1      walk2
                (walk-to  walk-from))
(UNIFY-EVENTS  walk2      walk3
                (walk-to  walk-from))
```

Figure 2.7: Convenience function for unifying event roles.

in Figure 2.8 each with a simple example.

With these representational aspects we can model event hierarchies with typed roles as well as complex temporal relations involving events. The last goal of the events KB involves representing processes and procedures for events. An event can be expanded into many subevents. For example, a walking action can be expanded into many step-taking actions and taking a step can be expanded into a sequence of limb-movement actions and so on. The expansions can be interpreted as processes, procedures, situations or plans depending on the use-case. We may use any of the terms process, procedure, plan or expansion to refer the subevents of an event. The events KB provides a generic framework for representing and reasoning about expansions of events.

Every event may have some number of subevent roles, which are events themselves. Creating temporal relations between the subevents of an event constitute describing the expansion temporally. Unifying the event's roles and the subevents' roles with each other creates a coherent representation of the changes that happen to the world. Both temporal information and unification information must be present for the expansion to make sense.

For example, the simplest temporal structure of an expansion is a sequence. To assert a sequential expansion for an event we can use the `HAPPENS-AFTER` function for the subevents. We also need to unify the roles that are the same throughout the event. Supplying all of this information involves many Scone calls and easily clutters knowledge bases. For this reason the events KB provides a


```

;; Syntax
(HAPPENS-AFTER event1 event2 :WAIT-FOR duration)
;; Example: waited for 5 minutes between first and second walk
(HAPPENS-AFTER walk1 walk2 :WAIT-FOR 5-minutes)

;; Syntax
(HAPPENS-CONCURRENTLY event1 event2)
;; Example: whistled while walking
(HAPPENS-CONCURRENTLY walk1 whistle1)

;; Syntax
(HAPPENS-AT-START event1 event2)
(HAPPENS-AT-END event1 event2)
;; Example: whistled at the beginning/end of the first walk
(HAPPENS-AT-START whistle1 walk1)
(HAPPENS-AT-END whistle1 walk1)

;; Syntax
(HAPPENS-DURING event1 event2 :START-DELAY duration1
                        :END-DELAY duration2)
;; Example: started whistling 5 minutes into the first walk and stopped
;;           2 minutes before the walk ended
(HAPPENS-DURING whistle1 walk1 :START-DELAY 5-minutes
                        :END-DELAY 2-minutes)

;; Syntax
(HAPPENS-OVERLAPPING event1 event2 :START-DELAY duration1
                        :OVERLAP duration2
                        :END-DELAY duration3)
;; Example: started whistling 5 minutes into the first walk, whistled
;;           for 10 minutes and stopped 2 minutes after the walk ended
(HAPPENS-OVERLAPPING whistle1 walk1 :START-DELAY 5-minutes
                        :OVERLAP 10-minutes
                        :END-DELAY 2-minutes)

```

Figure 2.8: Convenience functions for temporally relating events. `whistle1` is an instance of a whistle event and the *x*-minutes elements are time-measure instances.

function to conveniently define expansions for events. Figure 2.9 shows the syntax of `NEW-PROCESS` and examples describing Clyde's walk from `Newell-Simon-atrium` to `Scott's-office`. In the examples we assume no unifications were made prior to the call.

Note that the `NEW-PROCESS` function uses a special list-syntax that only contains temporal information. A process may contain alternative branches (do `action1` or `action2`) or temporally-independent events (do `action1` and `action2` in any order). Alternative branching is handled by the `IS-A` hierarchy itself. Figure 2.10 shows an example procedure involving branching. Temporally independent events are trivial to have in expansions. The nonexistence of temporal relations between subevents implies that the subevents can be done in any order. Note that knowledge bases involving procedures should be written in a bottom-up fashion.

The temporal relations together with alternative branches allow for very complex process structures. However, there are certain processes that cannot be effectively defined with the events KB. These are iterative events that either happen many times or have a termination condition. For example, an expansion with a subevent repeating thousands of times is representable using the events KB, but it is representationally uneconomical. Thousands of subevents would have to be created even if there is nothing special to say about each of them. On the other hand iterative events with termination conditions (e.g. hit the nail until it's embedded) cannot be represented correctly at all since the number of subevents is not certain. These cases are considered in more detail in Section 2.4 as future work.

This completes the representational capabilities of the events KB. Unlike the time KB, the events KB does not have non-marker-propagation reasoning functions. So all query functions related to events are calls to standard Scone calls. This makes the events KB particularly simple and efficient.

2.4 Future Extensions

The implemented features in the events KB provide a foundational framework for many uses. There are more representational features that can be added to the KB, which will ultimately give more

```

;; Syntax
(NEW-PROCESS event
  { (:SEQ      [start-delay] {event [duration]}+ event [end-delay] ) |
    (:CONC     event+ ) |
    (:DURING   event1 event2 [start-delay end-delay] ) |
    (:OVERLAP  event1 event2 [start-delay overlap end-delay] ) }
  BODY)

;; Example: Clyde walked from the atrium to Scott's office, taking a
           break between each walk.
(NEW-PROCESS Clyde's-big-walk
  (:SEQ walk1 5-minutes walk2 10-minutes walk3)
  (UNIFY-EVENTS walk1 walk2
    (walk-to walk-from))
  (UNIFY-EVENTS walk2 walk3
    (walk-to walk-from))
  (UNIFY-EVENTS Clyde's-big-walk walk1
    (walk-from walk-from))
  (UNIFY-EVENTS Clyde's-big-walk walk3
    (walk-to walk-to)))

;; Example: Driving a car is done by steering, using the pedals and
;;           watching the road at the same time by the same agent
(NEW-PROCESS drive-car
  (:CONC steer-car use-pedals watch-road)
  (UNIFY-EVENTS drive-car steer-car use-pedals
    (vehicle-used vehicle-used vehicle-used))
  (UNIFY-EVENTS drive-car steer-car use-pedals watch-road
    (agent agent agent agent)))

;; Example: The robot should kick the ball 2 seconds after moving
(NEW-PROCESS kick-moving-2
  (:DURING kick-ball move 2-seconds nil)
  (UNIFY-EVENTS kick-moving-2 kick-ball move
    (agent agent agent))
  (UNIFY-EVENTS kick-moving-2 kick-ball
    (ball-kicked ball-kicked)))

;; Example: The robot should move forwards and start moving left
;;           overlapping with the forward move
(NEW-PROCESS left-overlapping-forwards
  (:OVERLAP move-left move-forwards 5-seconds 2-seconds 3-seconds)
  (UNIFY-EVENTS left-overlapping-forwards move-left move-forwards
    (agent agent agent)))

```

Figure 2.9: Convenience function for defining expansions.

```
(NEW-EVENT-TYPE drive-to-airport (go-to-airport))

;; We can take the highway 1
(NEW-EVENT-TYPE drive-to-airport-H1 (drive-to-airport))
(NEW-PROCESS    drive-to-airport-H1
  (:SEQ go-to-H1-onramp drive exit-at-airport))

;; We can take the highway 2
(NEW-EVENT-TYPE drive-to-airport-H2 (drive-to-airport))
(NEW-PROCESS    drive-to-airport-H2
  (:SEQ go-to-H2-onramp drive exit-at-airport))
```

Figure 2.10: Representing expansions with alternative branches. Unifications were omitted for conciseness.

reasoning capability. Below are some ideas for extending the time and events knowledge bases. We discuss these ideas in more detail in Chapter 5.

- During Contexts (Section 5.1)
- Recurrent Times and Events (Section 5.1)
- Causes, Intentions and Goals (Section 5.2)
- Executable Actions and Planning (Section 5.3)

Chapter 3

Algorithms and Implementation

Event monitoring and detection systems can be placed in a general architecture shown in Figure 3.1. The described architecture observes a world and feeds information about the world to an event monitoring and detection system. The responses of this system is then sent to a reminder/warning module which decides if and when to alert the user. Every module in this scheme (the box, the diamond and the triangle) can make use of a knowledge representation system, such as Scone. For example, the reminder/warning module may use knowledge about the preferences of the user being tracked to be most productive. If the observations of the world is text, the information extractor may use knowledge based natural language processing to extract structured knowledge.

As indicated by the double-lining in the figure, we are focusing on the event monitoring and detection module in this scheme. We assume that observations have been processed by the information acquisition module, and coherent information packets are input to the module. The module propagates the information through consistent processes in the knowledge base in order to either monitor or detect events. In order to be modular, we enforce the same form of information packets to be received for both tasks. There are two kinds of information packets allowed: event information packets and time updates. Each event information packet contains what kind of event happened in the world, when it happened and what was involved in the event. For example, if Andrew filled a form at 2.30pm with his address, the information packet would resemble (`fill-form (time 2.30pm)`

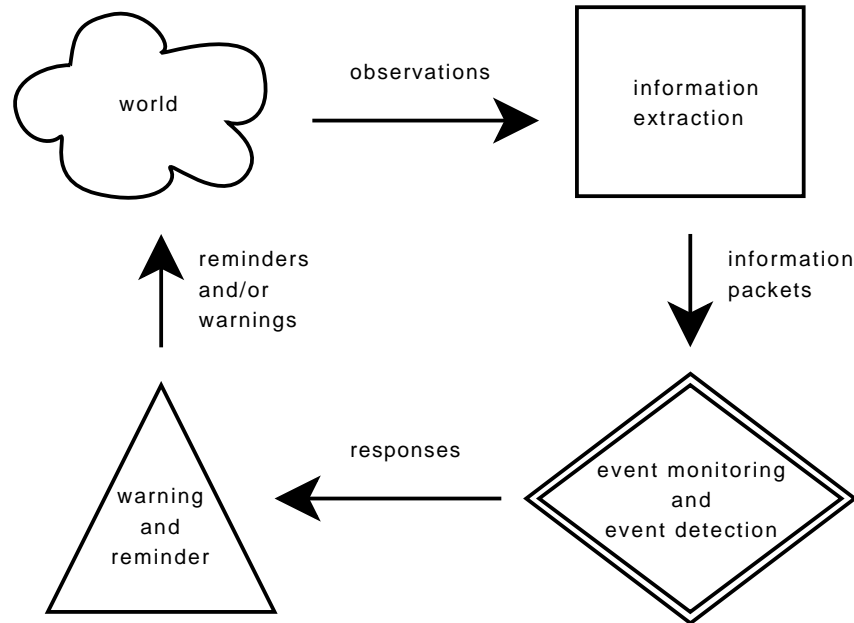


Figure 3.1: An architecture diagram for an assistant that warns and reminds the user as it observes events happen in the world.

(agent Andrew) (address-entry Andrew's-address)). Time update information packets are system triggered time updates in order to keep track of deadlines and necessary delays. These updates are in the form (time-is DD-MM-YYYY hh:mm:ss).

3.1 Event Monitoring

In the event monitoring task, there is a single agent doing a single task in the observed world. The goal of our system is to notice inconsistencies between the procedure defined for the task and the agent's execution. There are three kinds of inconsistencies we wish to capture: doing an incorrect action (Andrew pays the vendor rather than clearing the purchase with his boss), doing the correct action with incorrect roles (Andrew pays the vendor with cash rather than a credit card) and finally doing the correct action too early or too late.

We assume that the observed agent does not do two or more actions at the same time. This

means that the processes being monitored cannot include concurrent or overlapping actions. The reason for this constraint is to make the task simpler and more straightforward. The code could be extended to allow concurrent and overlapping actions and we consider how this could be done at the end of the section. We also assume all actions of the agent are observed.

The algorithm for monitoring events is quite straightforward. We are given an event which has a procedure defined for it. The steps in the procedure may recursively be expanded into their own procedures. When the expansion stops, the leaves in the procedure tree will contain events with no procedures defined for them. The first step in the event monitoring algorithm is to retrieve the expanded procedure for the event to be monitored from Scone. Then the module starts waiting for information packets, and as information packets are received, the constraints in the packet (such as when the action happened, what items were involved, etc.) are compared to the procedure's constraints. If the constraints agree, the new ones are propagated; otherwise the module makes note of the inconsistency. If the procedure is completed at any time, the module stops the monitoring process and returns a success value.

So, the described algorithm roughly has three parts: i) getting the expanded procedure for the event; ii) checking if the information is consistent or not; iii) propagating the information to future events. The events KB provides an `expand-event` function, which gets the procedure for an event and recursively expands each step. The returned list has a special form described in Figure 3.2. The second step is checking whether the given time and roles of the executed action are consistent with the knowledge base and previous constraints. Keeping track of constraints given by the user is done by the event monitoring software. A `constraint-form` is added to the `expansion-form` defined in 3.2. The constraint-form is a simple list of tuples, where each tuple is the type and value of a constraint. Alternatively, one could instantiate the actions reported by the information packets in Scone and use Scone to store the constraints. This would be especially useful when other algorithms need access to this knowledge; however, handling the constraints in Lisp is much simpler and sufficient for our needs. Finally, the third step of the algorithm involves propagating the time and role information to other events in the procedure to form new constraints. The events KB provides


```

event-form      ::= event-name | expansion-form
process-form    ::= (:OR event-form+) | (:AND event-form+) |
                   (:SEQ [duration]
                        {event-form [duration]}+ event-form
                        [duration])
expansion-form  ::= (event-name process-form)

(EXPAND-EVENT event) ==> expansion-form

;; Example
(EXPAND-EVENT order-food) ==>
  (order-food (:SEQ find-food-needs
                    choose-vendor
                    (fill-vendor-forms
                     (:AND (fill-order-form (:SEQ ... ))
                           (fill-payment-form (:OR pay-with-credit
                                                    pay-with-account))))
                    email-vendor-forms))

```

Figure 3.2: The syntax of the expansion-form returned by `expand-event`.

`get-unifications`, a function that returns all roles that are unified between two events. Since unifications are implemented by the events KB as EQ links, and since EQ links are transitive, constraint propagation is actually handled by `Scone`. The event monitoring software only copies the constraints to the right events. Function 1 shows the top-level function for event monitoring. Consistency checking and propagation of constraints are done in Function 2. `try-propagate-constraints` (Function 2) is a recursive function that follows a different propagation policy for each kind of process. For space reasons, we've separated each case into functions 3, 4 and 5.

The sequential process case (function 3) is the simplest case. The information packet is compared with the first incomplete subevent in the sequence. If this subevent has a process of its own, we recurse; otherwise the constraints are checked and copied to the subevent. If the checks failed, we note the inconsistency and return the unchanged process. If the constraints were consistent, the updated subevent (which is now marked as the last subevent done) is replaced with the old. Finally the constraints are propagated to the rest of the process from the replaced subevent using

Function 1: monitor-event

Input: *event* with a non-empty procedure**Output:** *T* if *event* is completed

```

process ← (expand-procedure event)                ;; provided by events KB
while true do
  | info ← (get-info-packet)
  | process ← (try-propagate-constraints process info)
  | if (is-complete? process) then return complete

```

Function 2: try-propagate-constraints

Input: *process*: an expansion-form with embedded constraints**Input:** *info*: an information packet**Output:** *process2*: the updated process

```

switch (expansion-type process) do                ;; provided by events KB
  | case :SEQ
  |   | (try-propagate-seq process info)                ;; see Function 3
  | case :OR
  |   | (try-propagate-or process info)                ;; see Function 4
  | case :AND
  |   | (try-propagate-and process info)                ;; see Function 5

```

the `propagate-from` function (see Appendix A). This function propagates the constraints from the given subevent to the rest of the process. The paths for the propagation are found using the `get-unifications` function supplied by the events KB.

In the branching process case (function 4), we try to propagate the information to each branch. If a branch has a process of its own, we recurse; otherwise we check the constraints and try to update them. If all branches are inconsistent, then the information is inconsistent with the process overall, so we note this inconsistency and return the original process. Otherwise, we update the process by copying each potentially updated branch (we also copy inconsistent branches to allow the user to switch processes.) Note that we do not propagate the constraints from a branch to the rest of the process, unless the branch is completed. This is to avoid running into inconsistencies when the user tries multiple branches. For example, let's say the user is trying to `make-frosting` and there are two branches: `make-chocolate-frosting` and `make-vanilla-frosting`. If the

Function 3: try-propagate-seq

```

next ← (next-event process)
if next has a subprocedure then
  | response ← (try-propagate-constraints next info)
else
  | response ← (check-update-constraints next info)
if response = inconsistent then
  | print inconsistent
  | return process
(replace-next process response)
process2 ← (propagate-from (last-updated-sub process) process)

```

user attempts to make-chocolate-frosting and if we propagate the constraints from that branch to the top-level the flavor role of make-frosting will be bound to chocolate prematurely. If the user changes his/her mind and starts following the make-vanilla-frosting branch the flavor role of make-frosting will still be bound to chocolate which will be inconsistent with make-vanilla-frosting's flavor, vanilla.

The temporally independent process case (AND process, function 5) is the most complicated of the three. First, we try to propagate the information to each incomplete subevent in the process. This step may recurse like the previous cases. If all incomplete subevents were inconsistent with the information, then the information is inconsistent with the overall process and the original process is returned. If there is a single consistent subevent, we update that subevent and propagate the constraints from that subevent to the rest of the process. If there are more than one consistent subevents we have to be careful. Since a single execution can't count for multiple actions in a process, we have to consider each match as a possible interpretation for the process. For example, let's say a process open-venthole involves four identical temporally-independent events unscrew. Each unscrew event, will involve a different screw which can be represented as negated unifications (creating negated EQ links between the roles.) If the user reports that s/he unscrewed one screw, we cannot know which of the four screw s/he unscrewed. For this reason, we have to create four interpretations of the process, where each interpretation has a different subevent completed. In order to represent these interpretations, we create a branching process for the top-level event with the same

Function 4: try-propagate-or

```

response ← [...] ;; array with a slot for each branch
foreach branchi in process do
    if branchi has a subprocedure then
        | responsei ← (try-propagate-constraints info branchi)
    else
        | responsei ← (check-update-constraints branchi info)
if all responses are inconsistent then
    | print inconsistent
    | return process
process2 ← (without-subevents process)
foreach responsei in process do
    if responsei ≠ inconsistent then
        | process2 ← (add-subevent process2 responsei)
        | if (is-complete? responsei) then
            | process2 ← (propagate-from (last-updated-sub process2) process2)
            | return process2

```

constraints, but where each branch is the differently interpreted AND process. Figure 3.3 abstractly shows how the process-forms would look. This approach uses the branching process representation to handle ambiguities, which minimizes the amount of code we write. Note that when a branch is completed in a branching process, the constraints are propagated from it to the rest of the process. Since the branch and top-level event are the same event in this approach, all constraints will be propagated up. One problem with this approach is that the branched process may grow factorially in bad cases. For instance, if the *open-venthole* event had n *unscrew* subevents, and if the user unscrewed t screws, the branching process would consist of $O\left(\binom{n}{t}\right)$ processes before the completion is recognized. Bad cases are unavoidable unless the knowledge base makes certain that there aren't too many temporally independent subevents in the same process. In the *open-venthole* example, the knowledge base may define four distinct temporally-independent subevents, such as *unscrew-topleft*, *unscrew-topright*, *unscrew-bottomleft* and *unscrew-topleft*.

There are multiple reasons why we have not implemented support for concurrent and overlapping processes. One could easily implement support for concurrent and overlapping processes, by adding more cases to the *try-propagate-constraints* function and allowing information pack-

Function 5: try-propagate-and

```

response ← [...] ;;; array with a slot for each subevent
foreach incomplete subeventi in process do
  if subeventi has a subprocedure then
    | responsei ← (try-propagate-constraints info subeventi)
  else
    | responsei ← (check-update-constraints subeventi info)
if all responses are inconsistent then
  | print inconsistent
  | return process
else if exactly one consistent response then
  | process2 ← (without-subevents process)
  | foreach responsei ∈ process do
    | process2 ← (add-subevent process2 responsei)
  | process2 ← (propagate-from (last-updated-sub process2) process2)
else ;;; more than one subevents match
  | process2 ← (empty-or-process-from process)
  | foreach consistent responsei ∈ process do
    | branch ← (without-subevents process)
    | foreach subeventj ∈ process do
      | if j ≠ i then branch ← (add-subevent branch subeventj)
      | else branch ← (add-subevent branch responsei)
    | branch ← (propagate-from (last-updated-sub branch) branch)
    | if (is-complete? branch) then return branch
    | process2 ← (add-subevent process2 branch)

```

ets to contain multiple event information. However; since the time KB only allows for precise closed time intervals, any temporal information about concurrent and overlapping processes would be unrealistic for event monitoring or event detection. For example, let's say we define an overlapping event with a process (:DURING whistle walk 30-seconds) meaning that someone started walking and 30 seconds into the walk started whistling. In order to monitor or detect this process, the 30-second duration would have to match *precisely*. We consider how the time KB can be extended to allow for more flexible duration descriptions in Section 5.1 and only with such flexible durations can concurrent and overlapping processes be accurately modeled.

```

(open-venthole
  (:AND unscrew1 unscrew2 unscrew3 unscrew4) (:CONSTRAINTS c1 c2 ...))
==>
(open-venthole (:OR (open-venthole (:AND (unscrew1 (:CONSTRAINTS done))
                                         unscrew2 unscrew3 unscrew4)
                                         (:CONSTRAINTS c1 c2 ... ))
                    (open-venthole (:AND unscrew1
                                         (unscrew2 (:CONSTRAINTS done))
                                         unscrew3 unscrew4)
                                         (:CONSTRAINTS c1 c2 ... ))
                    ...2 more branches... )
  (:CONSTRAINTS c1 c2 ... ))

```

Figure 3.3: An AND process may branch out into multiple interpretations when an information packet is consistent with multiple incomplete subevents.

3.2 Event Detection

In the event detection task, the system tries to recognize a process that is happening in an observed world. Like in event monitoring, the event detection module receives information about events that happen in the world. We've assumed for simplicity that the events are done by the same agent and that the agent is doing a single event at a time.

We've assumed that the knowledge bases containing domain knowledge may be very large. This was one of our assumptions about real-world tasks. For this reason, the event detection algorithm has to be efficient enough to handle hundreds if not thousands of complex processes. We provide three algorithms of increasing complexity to deal with huge knowledge bases.

The simplest algorithm filters out potential processes by looking up which ones have subevents that match the received information. No constraints are checked at all. There several advantages to using this algorithm. First of all it is very efficient and fast. Secondly, the domain knowledge may not be completely correct. The same process may be executed in a slightly different structure. The fact that this algorithm doesn't care about process structure allows it to potentially detect these unknown cases. The main disadvantage of this algorithm is its potential low precision. The relevant

processes will be in the set of results; however, depending on the domain knowledge the number of irrelevant processes in the result can be high. For example, if a knowledge base about bombs contains hundreds of methods to build bombs, and most involve the same basic actions as steps, this algorithm may be ineffective. What this algorithm can provide though is the initial filtering out of processes so the more complex algorithms can become computationally feasible. This algorithm is detailed in Function 6.

Function 6: detect-by-subevents

Input: *info*: an information packet

Input: *candidates*: a list of event candidates, if empty no candidates have been selected yet

Output: an updated list of event candidates

event \leftarrow (event-name *info*)

processes \leftarrow (get-events-with-subevent *event*) ;; provided by events KB

if *candidates* = \emptyset **then return** *processes*

else return *candidates* \cap *processes*

The second algorithm uses the basic process structure to find relevant events. By the process structure we mean the knowledge that a process is sequential or involves many alternative branches, etc. Time or role constraints are not used in this algorithm. Compared to the first algorithm, this method provides a deeper analysis, so it will have a higher precision measure. We believe that many processes can be eliminated by using the simple structure information, allowing the system to move on to the final algorithm. The second algorithm is detailed in Function 7.

Function 7: detect-by-structure

Input: *info*: an information packet

Input: *candidates*: a list of event candidates, if empty no candidates have been selected yet

Output: an updated list of event candidates

candidates2 \leftarrow \emptyset

event \leftarrow (event-name *info*)

if *candidates* = \emptyset **then**

candidates \leftarrow (get-events-with-subevent *event*)

foreach *candidate_i* \in *candidates* **do**

response \leftarrow (check-structure *candidate_i* *event*)

if *response* \neq *NIL* **then** *candidates2* \leftarrow *candidates2* \cup *response*

return *candidates2*

The final algorithm is very similar to the event monitoring algorithm. In event monitoring, there is a single process that the system is following and when there is an inconsistency it is only noted. In the third event detection algorithm, the same idea applies. Here we may have a number of events that we are ‘monitoring’, but when an inconsistency is detected with an event, that event is dismissed, reducing the number of possibilities. So, the only changes to the event monitoring algorithm we’ve made are dismissing events when inconsistencies occur, and creating a dummy branching event at the very beginning. The dummy branching is to trick the algorithm to believe that it is monitoring an overall process with multiple branches. These branches will eventually be pruned and we will be left with a single one, completing the detection procedure. Function 8 shows the algorithm in detail. Note that the `detect-by-all` function uses the `try-propagate-constraints-x` function, which is a version of the `try-propagate-constraints` function that returns NIL when an inconsistency is detected (we don’t reproduce these versions of `try-propagate-constraints`, `try-propagate-seq`, `try-propagate-or` and `try-propagate-and` to avoid repetition.) This algorithm uses all available knowledge about the events, so it is the most demanding of the three. The other two algorithms can be used initially to greatly reduce the number of possible processes so that this algorithm can become feasible for real-time event detection.

Function 8: `detect-by-all`

Input: *info*: an information packet

Input: *candidates*: a list of event candidates, if empty no candidates have been selected yet

Output: an updated list of event candidates

candidates2 $\leftarrow \emptyset$

event \leftarrow (`event-name` *info*)

if *candidates* = \emptyset **then**

 | *candidates* \leftarrow (`detect-by-structure` *info* *candidates*)

;; form a dummy expansion-form: (`event` (`:OR` *candidate*₁ *candidate*₂ ...))

dummy \leftarrow (`list` *event* (`cons` `:OR` *candidates*))

dummy \leftarrow (`try-propagate-constraints-x` *dummy* *info*)

return (`list-subevents` *dummy*)

In order to provide a single solution to the event detection problem, we’ve combined these three algorithms into a single module. The module selects which algorithm to use depending on the

number of possible matches. When the number is very high the simplest algorithm is used, when the number becomes tolerable the module switches to the second algorithm, and when the number is very small the final algorithm is used. All information packets are stored by the module, so that when an algorithm switch happens the same information can be rescanned by the new algorithm. The user can set the thresholds for the switches, or s/he can manually select which algorithm the module should use. The pseudocode for this module is shown in Function 9.

Function 9: detect-event

```

candidates  $\leftarrow \emptyset$ 
history  $\leftarrow \emptyset$ 
algorithmold  $\leftarrow$  detect-by-subevents
while true do
  info  $\leftarrow$  (get-info-packet)
  history  $\leftarrow$  history  $\cup$  info
  if  $|candidates| == 0 \vee |candidates| > threshold_1$  then
     $\lfloor$  algorithmnew  $\leftarrow$  detect-by-subevents
  else if  $threshold_1 \geq |candidates| > threshold_2$  then
     $\lfloor$  algorithmnew  $\leftarrow$  detect-by-structure
  else
     $\lfloor$  algorithmnew  $\leftarrow$  detect-by-all
  if algorithmold  $\neq$  algorithmnew then
    foreach infoi  $\in$  history do
       $\lfloor$  candidates  $\leftarrow$  (apply algorithm info candidates)
  else
     $\lfloor$  candidates  $\leftarrow$  (apply algorithm info candidates)
  if  $|candidates| == 0$  then
     $\lfloor$  print no candidates left.
     $\lfloor$  return NIL
   $\lfloor$  algorithmold  $\leftarrow$  algorithmnew

```

Chapter 4

Experiments

The presented work is aimed to be used in assisting users in two real-world and knowledge intensive tasks: event monitoring and event detection. In order to quantify the effectiveness of the solutions provided, we have to define metrics for evaluation. Both tasks involve information observations and responding to the information. So for each information packet received, the assistant responds to the packet by changing its internal representation of the situation. The correctness of the internal representation forms the basis of the effectiveness of the assistant.

Tables 4.1 and 4.2 describe how these information events can be used to define type-I (false positive) and type-II (false negative) errors as well as the correct cases for the event monitoring and the event detection tasks, respectively.

In event monitoring the assistant observes the execution of a procedure to catch any inconsistencies. For each information input, the assistant decides whether that piece of information indicates an inconsistency or not. This decision can be quantitatively used to measure the effectiveness of the assistant. Table 4.1 shows how each event in an event monitoring experiment can be interpreted as being correct or incorrect.

In event detection the assistant is observing a world where a procedure is taking place. The

True Positive	Information is consistent and assistant recognizes consistency
False Positive	Information is inconsistent and assistant recognizes consistency
True Negative	Information is inconsistent and assistant recognizes inconsistency
False Negative	Information is consistent and assistant recognizes inconsistency

Table 4.1: Event monitoring metrics for information input events.

assistant is supposed to identify which procedure is happening using the information it receives. At the beginning of the task all known procedures will be regarded as possible by the assistant. As information is input, the assistant should eliminate procedures that cannot be happening. We will take the decisions to eliminate procedures as the basis for evaluating the assistant.

We could also interpret the assistant's decisions as retaining rather than eliminating procedures. With this interpretation the event detection task draws a similarity with the unranked information retrieval (IR) task. In IR the task is to find the relevant set of documents given a query. Here the assistant needs to find the consistent set of procedures given information about the world. Many different IR evaluation metrics have been thoroughly studied and compared. We cannot immediately use the developed metrics as they are, since the tasks are different in nature. In event detection the information input events (which roughly correspond to queries) are incremental and furthermore the information input events monotonically decrease the number of procedures possible. For this reason, we will only define the fundamental metrics to evaluate the assistant: we will draw equivalents to precision, recall and F-measure. We will refer to the set of retrieved procedures at an information input event as R and the set of truly possible (relevant) procedures as T . Table 4.2 shows how type-I and type-II errors can be defined for the event detection task.

True Positive	Retained and relevant procedures: $R \cap T$
False Positive	Retained and irrelevant procedures: $R - T$
True Negative	Eliminated and irrelevant procedures: $R' \cap T'$
False Negative	Eliminated and relevant procedures: $R' \cap T$

Table 4.2: Event detection metrics for information input events. R' and T' denote the compliment sets of R and T .

With these we can define precision and recall in the conventional way:

$$Precision = \frac{|R \cap T|}{|R|} \quad Recall = \frac{|R \cap T|}{|T|} \quad (4.1)$$

It is important to note that since both R and T are monotonically getting smaller in the course of an experiment, both precision and recall should remain reliable metrics at every information input event. The F-measure can then be defined as a harmonic mean of precision and recall (Equation 4.2.) The F1-measure (equal weighting of precision and recall) is the most commonly used F-measure in the IR task. One could argue that for event detection recall is slightly more important than precision, especially in threat detection cases. Different tasks may find different weighting schemes more useful.

$$F_{\beta} = \frac{(1 + \beta^2)(precision \cdot recall)}{(\beta^2 \cdot precision + recall)} \quad F_1 = \frac{2 \cdot (precision \cdot recall)}{precision + recall} \quad (4.2)$$

It must be noted that regardless of what algorithms are used to monitor or detect events, the knowledge base with the domain knowledge has a great impact on the effectiveness of the assistant. These evaluation metrics don't only tell us how well an algorithm is doing, but also how precise a knowledge base is for the domain.

In this thesis, we would like to establish the feasibility of the presented approach rather than to evaluate it immediately. Once the feasibility is established, user studies can be done to evaluate the algorithms and the knowledge bases. For our experiments, we've developed domain knowledge bases for the event monitoring scenario and the event detection scenario, each. These knowledge bases encode feasible and realistic domain knowledge about conference organization and national security, respectively. We've simulated the use of the presented system with these knowledge bases in order to answer the following questions about each task.

Event Monitoring

Can the assistant follow the event when the execution is correct?

Can the assistant detect missing or incorrect steps?

Can the assistant detect time and role inconsistencies?

Is the assistant tolerant of the user trying multiple branches in a branching process?

Event Detection

How effective are each of the three detection algorithms?

Do the algorithm switches happen correctly?

4.1 Monitoring Experiment: Conference Organization

We go back to our original example, conference organization, and see how our assistant can help out our simulated organizer Andrew. Andrew's task is to organize a particular day of a conference held at a university. He will organize the day using his desktop computer which makes available to him a number of tools for allocating resources and rooms, communicating with guests and vendors and publishing the conference website. For example, using a resource allocation optimizer Andrew can submit exactly which resources to use for each event. We assume his actions are being monitored and sent to the event monitoring assistant.

The assistant has previous knowledge about people, their dietary needs, professions and hierarchies, the university buildings and rooms, the resources (projectors, tables, microphones), events, and actions that are usually involved in conference organizing (ordering food, e-mailing attendees, publishing the website, etc.) We assume that this knowledge is inherently correct.

The day in question starts with a breakfast, followed by some talks and lectures, followed by lunch and concluded with a demonstration of the latest Robo-soccer teams. Certain tasks, like setting up lectures, have sequential and linear plans. Tasks like ordering food; however, can have multiple alternative plans.

Experiment Transcript

Andrew pledges to do the `organize-day` procedure

```
* (monitor-event  organize-day)
```

```
Started monitoring [organize-day] at 13:20.
```

```
>>
```

Andrew checks the procedure associated with `organize-day` (without recursively expanding the procedure.) Then he checks what next actions he can do and he sees that he can only do a basic action of type `get-guest-info`.

```
>> (show-process :depth 1)
```

```
(organize-day (:SEQ get-guest-info-1      setup-breakfast-1 setup-keynote-1  
                  setup-presentations-1 setup-lecture-1   setup-lunch-1  
                  setup-robodemo-1))
```

```
>> (peek-at-next)
```

```
get-guest-info-1
```

Andrew executes a `get-guest-info` action. This action causes Andrew and the assistant to learn how many guests there are, and how many of them are vegetarian.

```
>> (get-guest-info (guest-population 100) (veggy-population 15)  
                  (nonveggy-population 85))
```

```
:) ok [13:21]
```

Andrew then checks the procedure and sees the propagated constraints. (The same constraints are propagated to the subprocedures of the shown steps, but we don't include them here for space reasons.)

```
>> (show-process :depth 1)
(organize-day (:SEQ (get-guest-info-1
                     (:CONSTS (start-time 13:21) (end-time 13:21)
                               (guest-population 100)
                               (veggy-population 15)
                               (nonveggy-population 85)
                               (done)))
              (setup-breakfast-1
                (:CONSTS (veggy-orders 15) (nonveggy-orders 85)))
              (setup-keynote-1
                (:CONSTS (keynote-population 100)))
              (setup-presentations-1
                (:CONSTS (presentation-population 100)))
              (setup-lecture-1
                (:CONSTS (lecture-population 100)))
              (setup-lunch-1
                (:CONSTS (veggy-orders 15) (nonveggy-orders 85)))
              (setup-robodemo-1
                (:CONSTS (demo-population 100))))))
```

Andrew, knowing what to do next starts filling up a food-order form. (We show the relevant part of the procedure to see if things are going fine.)

```
>> (fill-order-form (veggy-entry 15) (nonveggy-entry 85))
:) ok [13:22]
>> (show-process setup-breakfast-1)
(setup-breakfast-1 (:SEQ (fill-forms-1
                          (:AND (fill-order-form-1
                                (:CONSTS (start-time 13:22)
                                           (end-time 13:22)
                                           (veggy-entry 15)
                                           (nonveggy-entry 85)
                                           (order-form-filled form-1)
                                           (done)))
                                fill-delivery-form-1
                                fill-payment-form-1))
                      (approve-forms-1
                        (:CONSTS (order-form-to-check form-1)))
                      (send-forms-1
                        (:CONSTS (order-form-to-send form-1))))
  (:CONSTS (veggy-orders 15)
            (nonveggy-orders 85)
            (start-time 13.22)))
```


Andrew then fills a delivery form and a payment form. After doing so he forgets to get the forms approved and tries to send them to the vendor.

```
>> (send-forms (order-form-to-send form-1)
              (delivery-form-to-send form-2)
              (payment-form-to-send form-3)
              (vendor-to-send Cycle-Factory))

!) Inconsistency in action type. Expected action [approve-forms-1].
```

Andrew is alerted to the problem, so he corrects his mistake. He sends the forms to his boss to get approval. Now, he is ready to send them to the vendor.

```
>> (send-forms (order-form-to-send form-1)
              (delivery-form-to-send form-2)
              (payment-form-to-send form-3)
              (vendor-to-send Cycle-Factory))

!) Inconsistency in expected role: The filler [Cycle-Factory] for role
   [vendor-to-send] is not ok.
```

Andrew realizes that he accidentally chose an incorrect vendor. He corrects the problem by sending the forms to the vendor Scone-Factory. The next task for Andrew is to setup the keynote speech. First, he must allocate a room for the keynote speech. The system for allocating rooms is unreliable, so Andrew needs to wait for 1 minute and then confirm the reservation.

```
>> (allocate-room (room-allocated Wean-Hall))

:) ok [13:44]

>> (confirm-room (room-to-confirm Wean-Hall))
```

```
!) Execution expected later: [13:44 + 1-minute]
>> (confirm-room (room-to-confirm Wean-Hall))
:) ok [13:45]
```

Andrew now goes on with the rest of the keynote speech. He reaches the `order-nice-thing` step, which can be done by either the `order-flowers` action or the `order-balloons` action. Andrew decides to order balloons.

```
>> (fill-balloon-order-form ... )
:) ok [13:52]
>> (fill-delivery-form ... )
:) ok [13:53]
>> (fill-payment-form ... )
:) ok [13:54]
>> (approve-forms ... )
:) ok [13:55]
```

His boss doesn't approve the forms. He thinks the balloons are too silly for a keynote speech. So Andrew starts ordering flowers instead, leaving the `order-balloons` procedure incomplete.

```
>> (fill-flower-order-form ... )
:) ok [14:05]
>> (fill-delivery-form ... )
:) ok [14:06]
>> (fill-payment-form ... )
:) ok [14:07]
>> (approve-forms ... )
:) ok [14:10]
```

```
>> (send-forms ... )
:) ok [14:12]
```

Here is what happened to the branching procedure `order-nice-thing`. Note that one branch is partially complete, and the overall procedure is complete, because the second branch is complete.

```
>> (show-process order-nice-thing-1)
(order-nice-thing-1
  (:OR (order-balloons-1 (:SEQ (fill-balloon-forms-2 (:AND ...)
                                (:CONSTS (done) ...))
                                (approve-forms-2
                                  (:CONSTS (done) ...))
                                send-forms-2)
        (:CONSTS ...))
    (order-flowers-1 (:SEQ (fill-flower-forms-2 (:AND ...)
                                                  (:CONSTS (done) ...))
                            (approve-forms-2
                              (:CONSTS (done) ...))
                            (send-forms-2
                              (:CONSTS (done) ...)))
        (:CONSTS (done) ...)))
  (:CONSTS (done) ...))
```

This completes the event monitoring demonstration. We've checked and confirmed that the system correctly recognizes incorrect action types, incorrect role information and temporal mistakes. The system can also handle switching to alternative actions in a branching process.

4.2 Detection Experiment: National Security

To demonstrate the event detection module, we will use the national security domain. We assume that the system is being used by a government agency to detect dangerous procedures happening in the world. We assume the information flows from the world through field agents. The information sent involves actions done by a single agent (may be a group of people who are collaborating.)

The knowledge bases encode information about building various kinds of bombs (sound bombs, improvised explosive devices, biological bombs, nuclear bombs etc.) Each type of bomb has multiple procedures associated with it (10 high level bomb-building procedures are included), and many procedures share similar basic actions. We demonstrate the use of the three different algorithms and how the retrieved set of procedures is affected. We manually choose the algorithm to use in order to have more control on the demonstration.

Experiment Transcript

Event detection is started. Someone obtains plastic containers and then obtains duct tapes. Both of these actions are pretty generic, and most bomb-building procedures involve these actions.

Detecting events ... (Algorithm 1)

```
>> (obtain-plastic-container (containers-obtained p-containers-1))
```

```
27 events possible
```

```
>> (show-possible)
```

```
build-bomb
```

```
build-sound-bomb build-IED build-bio-bomb build-chem-bomb
```

```
build-sound-bomb-tube build-sound-bomb-trash build-sound-bomb-suitcase
```

```
build-IED-weight-trip build-IED-line-trip build-IED-movement-sensor
```

```
build-IED-heat-sensor
```

```
build-bio-bomb-gas build-bio-bomb-liquid build-bio-bomb-suicide
```

```
build-chem-bomb-gas build-chem-bomb-liquid build-chem-bomb-suicide ...
```

With the first piece of knowledge, the first algorithm accurately eliminates building nuclear bombs. Nuclear bombs, according to the domain knowledge, do not involve plastic containers but aluminum ones. Later, the system learns that glass containers were bought.

```
>> (obtain-glass-containers (containers-obtained g-containers-1))
15 events possible
>> (show-possible)
build-bomb
build-bio-bomb build-chem-bomb
build-bio-bomb-gas build-bio-bomb-liquid build-bio-bomb-suicide
build-chem-bomb-gas build-chem-bomb-liquid build-chem-bomb-suicide
obtain-parts-bio-bomb-gas      obtain-parts-bio-bomb-liquid
obtain-parts-bio-bomb-suicide obtain-parts-chem-bomb-gas
obtain-parts-chem-bomb-liquid obtain-parts-chem-bomb-suicide
```

According to domain knowledge, buying glass containers is only involved in building chemical and biological bombs. At this point we believe Algorithm 1 has done enough, so we switch to Algorithm 2 where simple process structure is taken into account. When we switch to Algorithm 2, the biological and chemical suicide bomb procedures are eliminated (domain knowledge tells us that terrorist organizations first convince a recruit to pledge on video for suicide bombings, which hasn't been observed in the world.)

```
>> (set-algorithm 2)
Using Algorithm 2. Rescanning history.
11 events possible
>> (show-possible)
build-bomb
build-bio-bomb build-chem-bomb
```

```
build-bio-bomb-gas  build-bio-bomb-liquid
build-chem-bomb-gas build-chem-bomb-liquid
obtain-parts-bio-bomb-gas      obtain-parts-bio-bomb-liquid
obtain-parts-chem-bomb-gas     obtain-parts-chem-bomb-liquid
```

The next observed action is buying empty aerosol cans, which leads the system to eliminate bombs with liquid hazards.

```
>> (obtain-aerosol-cans (obtained-cans aerosol-cans-1))
7 events possible
>> (show-possible)
build-bomb
build-bio-bomb build-chem-bomb
build-bio-bomb-gas build-chem-bomb-gas
obtain-parts-bio-bomb-gas obtain-parts-chem-bomb-gas
```

At this point we switch to the third algorithm and use all available domain knowledge. Switching to the third algorithm, confirms that the `obtain-parts-...` subprocedures have been completed.

```
>> (set-algorithm 3)
Using Algorithm 3. Rescanning history.
Completed: obtain-parts-bio-bomb-gas
Completed: obtain-parts-chem-bomb-gas
5 events possible
>> (show-possible)
build-bomb build-bio-bomb build-chem-bomb
build-bio-bomb-gas build-chem-bomb-gas
```

The system can be used to query the next expected action for a possible procedure. For instance, the expected action for the `build-bio-bomb-gas` procedure is stealing biohazard material and using the obtained glass containers for transfer.

```
>> (show-expected build-bio-bomb-gas)
(biohazard-theft (:CONSTS (theft-containers g-containers-1)))
```

4.3 Analysis

With the limitations and assumptions described in Chapter 1, event monitoring and event detection are very similar problems. The fact that the third detection algorithm (function 8) is almost identical to the event monitoring algorithm (function 1) is evidence for the similarities. At a high level both tasks are responsible for checking and propagating the constraints in the received information packet through the procedures being tracked. The only difference between monitoring and detection is what happens when inconsistencies occur. Monitoring makes notes of them while detection eliminates inconsistent procedures. The reason for having multiple detection algorithms is to reduce the number of tracked procedures as quickly as possible to keep the computational demand tractable.

The limitations and assumptions we've set for event monitoring necessitate a tightly controlled environment such as a desktop computer. We've assumed that every action the user does is observed. However; realistically the user may move out of the controlled environment (like picking up the phone and calling someone, instead of e-mailing them), essentially breaking the monitoring process. It is important to study user behavior in the environment the system will be deployed to set realistic constraints to the task.

The event detection task has even stronger assumptions. Not only did we assume that every action of the user were observed, we assumed that every observed action was relevant to the procedure being executed. In the national security case, for instance, we may assign the field agents the responsibility for judging relevancy. However, it is unrealistic to assume that the field agents

will observe every action or know if each action is relevant or not. It is fundamentally important to further study how these assumptions can be relaxed or eliminated completely. As suggested earlier, different modules in a knowledge-based architecture may be responsible for these problems, or the detection algorithms can be implemented as ranking algorithms that never eliminate procedures.

Writing general and open-ended knowledge bases, such as those we presented for time and events, require a significant amount of research and design. The process of developing these knowledge bases showed us that expressiveness in a knowledge-base system is extremely important. Representational constraints will undoubtedly limit the kinds of tasks the system can support. However, the expressiveness of the system brings a greater responsibility to the developer in providing the right kind of reasoning mechanisms that are both useful and efficient.

Finally, it is very important to note that in event monitoring and event detection, or any other knowledge-based problem, the role of the domain knowledge bases are extremely important. Scone knowledge bases are somewhat human-readable and for this reason may be easy to learn and use. However, for wide scale use it is important for a knowledge-base system to have a natural language interface. Natural language interfaces would not only be useful for writing domain knowledge bases, but would also be invaluable for the information acquisition step.

Chapter 5

Conclusion and Future Work

The primary goal of the thesis was to demonstrate the use of expressive knowledge-base systems that allow simple and efficient reasoning mechanisms help solve complex AI problems. We provided solutions to two such problems, event monitoring and event detection that use common knowledge bases developed for the Scone knowledge representation system. These knowledge bases were developed to be as expressive and general as possible. Any system that requires representing and reasoning about time, temporal world-states and events should be able to make use of these knowledge bases. Our solutions to the event monitoring and event detection tasks consisted of about 500 lines of task-relevant code, combined. Most of the code written was for getting the information from the environment and managing and propagating process-related constraints. In contrast, about 3500 lines of knowledge base code was written.

The presented work may be extended further in at least two ways. One can add new features to the knowledge bases which will allow for more flexible monitoring and detection. One can also implement knowledge-based solutions to problems surrounding event monitoring and detection (such as information acquisition, learning and user interaction).

5.1 Extending the Time Knowledge Base

The presented time KB provides a way to represent and reason about time points and closed time intervals (the start and end points of the interval are contained in the interval.) A healthy way to extend the time KB would be to implement a flexible and expressive interval KB. The interval algebra should support open-ended intervals so that flexible durations are possible to represent. For instance, in the event monitoring and detection tasks having only closed intervals forces us to define precise temporal gaps between events (i.e. confirm the reservation exactly 3 minutes after you submit it.) With open-ended intervals it would be possible to assert flexible temporal constraints such as minimum and maximum durations. It is also important for the interval algebra to allow probabilistic durations. It should be possible, in general, to assert that the life-time of a human being follows a probability distribution, and then to query how likely a particular age is for a human being.

Intervals having two reference points (the start and the end) limits the time KB to two temporal contexts per interval. More reference points in an interval would provide a richer representation for temporal changes. Cyclical actions, such as patrolling, have important world-state information during the action rather than strictly before and after the action.

Finally, recurrent times and events are important concepts that are not representable in the provided framework. For example, we cannot represent the simple reference “every Monday” or the more complex one “between 6pm and 3am of every Wednesday of every other week.” Using such recurrent time references one can then have recurrent events, such as taking the recyclable trash out which has to happen “between 6pm and 3am of every Wednesday of every other week” in Squirrel Hill, Pittsburgh. Recurrences are extremely important concepts that are all around us and we must develop tools that make it easy to reason about them. It is important to note that for recurrences, representation is a simple problem to solve. It is more important to correctly frame what kinds of reasoning the KB will provide. The most important feature that a time KB should provide is the ability to instantiate a recurrent time interval given another time frame. For example, for a recurrent

reference “every Monday”, if one asks what actual references it corresponds to in a certain month s/he should be able to get the Mondays in that month. Another important feature is querying whether a time reference matches a recurrent references profile. So one might ask whether or not the 23rd of January 2008 matches the recurrent reference “every Monday.” More application specific queries can be implemented, but these two features seem to be the most fundamental and important.

5.2 Causes, Intentions and Goals

The time KB provides a way to represent the before and after world-states of a time interval, and in the context of events these world-states may be interpreted as pre-conditions and post-conditions of an event; however they are not strictly representations of causes or intended effects of the event. For various applications like planning or mental modeling, goals, causes and intentions might be important aspects that need to be represented separately.

Consider the two actions of pouring milk on the table, and spilling milk on the table. These two actions differ from each other only in their intentional aspects. Applications such as natural language comprehension or mental modeling would require a framework for representing intentions.

Closely tied to intentionality, is the concept of causality. Like intentions, being able to represent and reason about causes are necessary for natural language comprehension. What causality is, is a millennia-old question studied by philosophers and physicists. We will stay a safe distance away from these philosophical questions, but look at how different forms of causality may be represented. We wish to make a point that there must be a formal way to represent and reason about causes of events. For example, it should be possible to assert that Clyde caused an apple to fall. It is also important to allow for multiple kinds of causality. Consider the following simple story.

1. Clyde went running and became thirsty. (event causes effect)
2. So, he went to the kitchen to drink water. (state causes action)
3. In the kitchen he saw a mouse and fainted. (state causes event)

Causation seems to be transitive, e.g. if x causes y and y causes z , then we may reason that x caused z . When asked about the above story, few will claim that Clyde fainted because he went running or because he went to the kitchen. So although, the statements in the parentheses all contain the word ‘cause’, the transitivity seems to not apply. Why this is the case is one of the questions we want to stay away from, but it is clear that these cases should be representable.

Another closely related concept is goals. For the problem-solving domain, preconditions, goals and effects are the bricks with which one builds solutions. There is nothing fundamentally difficult about representing goals in Scone; however, we would like to explicitly mention that goals are semantically different (since they are usually attributed to the agents) and therefore should be represented separately.

5.3 Executable Actions and Planning

With the presented events KB, Scone is a passive representation and reasoning system; it does not change its own state or affect the external world. For this reason we have referred to the system that does event detection and/or event monitoring an intelligent *assistant* and not an intelligent *agent*. One could extend the events KB to include actions that Scone could execute and therefore change its own state or the world around it.

For instance, let’s assume that for a time interval, the begin and end time points are defined, but the duration of the interval is not explicitly defined. If we ask Scone for the duration using generic query functions (those that are not supplied by the time KB) we would not get an answer. Only if we go through the time KB will the duration be calculated and an answer returned. The reason is the lack of IF-NEEDED and IF-ADDED procedural hooks in Scone. These hooks are procedures that are executed at query and insertion times, respectively. So for the duration example, an IF-NEEDED hook may be attached to the duration property of time intervals that calculate the duration if the begin and end points are defined. To extend this example, let’s say there are two end-to-end time intervals `time-interval-1` and `time-interval-2` and we know the begin time and

```
(NEW-EVENT-TYPE add-two-numbers (Scone-executable)
  (n1 add-number-1 number)
  (n2 add-number-2 number)
  (sum sum-of-numbers number)
  (f lisp-function)

  (:ALWAYS
    (f EQ (NEW-LISP-FN ((x n1) (y n2) (:result sum))
      (+ x y))))
  (:BEFORE
    (sum EQ unknown-value))
  (:AFTER
    (NOT (sum EQ unknown-value))))
```

Figure 5.1: How a Scone executable function could be defined.

the duration of the first interval, and only the duration of the second interval. When asked what the end time of `time-interval-2` is, Scone should be able to answer the question by first calculating the end time of the first interval, which is the start time of the second interval, and then the end time of the second interval. If we define IF-NEEDED hooks for each property, our initial query will fire an IF-NEEDED procedure, which will make more queries and therefore may fire more IF-NEEDED hooks. Without procedural hooks, such reasoning mechanisms would have to be implemented in a specific way, just as we did for the time KB.

Procedural hooks could be easily implemented by attaching Lisp functions to Scone elements; however, this would leave Scone unable to reason about its own actions. Alternatively, we could attach Scone-executable action roles to elements that represent procedural hooks. These Scone-executable actions would have embedded Lisp functions with arguments and result values that are bound to other Scone elements. Figure 5.1 shows a mock definition of the `add-two-numbers` Scone-executable action that has four roles: the embedded Lisp function, the two added numbers and the sum. The Lisp function (`f`) is a lambda expression whose arguments and result (`x`, `y` and `:result`) are bound to Scone elements (`n1`, `n2` and `sum`.) The action also has descriptions of the before and after states: at the beginning of the action the sum is not defined; at the end of the action

the sum is known.

There are multiple advantages of Scone being able to reason about its own actions. With semantic representations of Scone's actions and a simple planner, Scone can plan out what it needs to do to answer the query. The same mechanism would give Scone the ability to change the world around it (if, for instance Scone is used as the knowledge component in a robot.) Scone could learn typical properties of its own actions over time, such as the durations and typical problems it encounters, to make its plans more robust and correct. The final knowledge based reasoner/planner/learner would resemble a complex AI system, much like the PRODIGY architecture [4]. Without a semantic representation of the actions, the best Scone can do is to execute chains of procedural hooks as described above.

It is important to remind ourselves that Scone is too expressive to have logically sound and complete reasoning mechanisms. So, the planner that uses Scone cannot be guaranteed to be optimal either. For this reason the use cases for the planning component must be carefully selected. Scone, in general, is intended to be used as a scalable, real-world oriented system. By 'real-world' we mean situations where unanticipated problems may arise for complex reasons. For instance, in a purchase order scenario, the boss may reject a purchase order, because s/he believes a lower price could be found. This is a 'higher-order' problem that can be alleviated by either convincing the boss, or finding a lower price. Scone's goal is to be able to address such complex situations in a reasonable way and a planner that uses Scone should be designed with the same goal in mind.

5.4 Information Acquisition

The information that flows from the world to event detection and monitoring systems may be in many forms. The raw information may in the form of user interaction logs within a desktop environment, sensor data from hardware or possibly ungrammatical text. These raw forms of information need to be interpreted into what we have referred to as information packets. More precisely, an information packet is a self-consistent chunk of knowledge compatible with the used knowledge

bases. In event monitoring and detection, an information packet corresponded to an event happening with relevant time and role information and we assumed that this information packet was provided by another module which has access to the same KBs.

When the raw information is precisely structured data (like data gathered by sensors or collected by event loggers), interpreting the data is a simple matter of structuring the data into a familiar form. When the raw information is unstructured, like natural language, the information acquisition task becomes a very difficult knowledge-dependent problem. It is crucial for any system that attempts to acquire information to have access to the same flexible and expressive knowledge bases used by other modules in the same architecture. Apart from the obvious need for architectural compatibility between modules, there are several functional reasons for this.

Using an expressive knowledge base allows information acquisition modules to convert raw information to knowledge at a higher level of granularity. The knowledge base itself can then be responsible for forming larger knowledge structures. For example, for natural language input, one may use a construction based grammar formalism to build large knowledge structures from smaller ones. The information acquisition module would parse the text and the knowledge bases would provide possible matches for the constructions. Also, using knowledge bases would provide a natural resource for handling ambiguities, especially lexical ones (such as anaphora and homonyms.)

5.5 User Interaction

When an inconsistency is detected in event monitoring, the system only makes notes of them. The user is not directly alerted or warned. When and how to provide these notes to the user is an important aspect that should be studied carefully. For example, the infamous Microsoft Office Assistant Clippy has a very bad reputation for being annoying. The primary reason for the dissatisfaction was its intrusive behavior, which resulted in turning it off by default in newer versions of Microsoft Office (see Microsoft's announcement cited in the bibliography: [8]). For any intelligent assistant to be realistically useful, its interactions with the user must be well designed. We believe the interac-

tion module of any assistant can benefit tremendously from an expressive knowledge representation system.

Most basicly, the KRS can be used to store personal profiles of users, so interactions can be customized for each user. More interestingly, interactions can benefit from the common-sense knowledge in the knowledge bases to appear thoughtful and intelligent. For example, events that involve high-rank colleagues make take precedence and events that involve family may take even higher precedence, depending on user preferences. Any common sense knowledge, like seasons, times of day, national and religious holidays, sports events, or even personal hobbies may be used to define helpful interactions.

5.6 Learning

We've assumed that the used knowledge bases contain the processes that are to be monitored or detected. The events KB provides many convenience functions for defining events; however, complex processes are still tedious to define. It should be easy to add new processes and procedures to the knowledge bases. There are multiple approaches to this learning task. When monitoring or detecting a process, the system can store the executed actions even if the monitoring or detection fails. Later, the system can ask the user whether the stored process is meaningful and correct. Also, the user should be able to initiate the learning by telling the system to watch what s/he is doing. This *learning by watching* procedure may not viable for all domains, especially for event detection processes where the events to be detected are deemed dangerous. It is desirable to provide a natural-language based interface for defining complex knowledge structures such as processes. The interface can be a simplified language, an artificial language or even an attempt at complete natural-language understanding. Whatever the interface is, we believe having use of an expressive knowledge base will prove invaluable.

Bibliography

- [1] James F. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.
- [2] K. Bach. Actions are not events. *Mind*, 89, 1980.
- [3] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The Berkeley FrameNet project. In *COLING/ACL-98*, pages 86–90, 1998.
- [4] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy: an integrated architecture for planning and learning. *SIGART Bull.*, 2(4):51–55, 1991.
- [5] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [6] Wei Chen and Scott E. Fahlman. Modelling mental states and their interactions. In *AAAI 2008 Fall Symposium on Biologically Inspired Cognitive Architectures*, 2008.
- [7] R. M. Chisholm. The descriptive element in the concept of action. 61:613–24, 1964.
- [8] Farewell Clippy. What's happening to the infamous Office assistant in Office XP, April.
- [9] Gilles Dowek. Higher-order unification and matching. pages 1009–1062, 2001.
- [10] C. J. Ducasse. On the nature and the observability of the causal relation. 23:57–68, 1926.
- [11] Scott E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, Cambridge, MA, 1979.
- [12] Scott E. Fahlman. Marker-passing inference in the scone knowledge-base system. In *KSEM*, pages 114–126, 2006.

- [13] Scott E. Fahlman. The Scone Project, 2008.
- [14] Christiane FellBaum, editor. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.
- [15] R. Fikes and S. Makarios. Kani time ontology. Technical report, Knowledge Systems Laboratory, 2004.
- [16] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI*, pages 608–620, 1971.
- [17] Charles J. Fillmore. The case for case. In Emmon W. Bach and Robert T. Harms, editors, *Universals in Linguistic Theory*, pages 1–88. Holt, Rinehart & Winston, New York, 1968.
- [18] Charles J. Fillmore. Frame semantics and the nature of language. In *In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech*, volume 280, pages 20–32, 1976.
- [19] G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 1992.
- [20] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 33–42, New York, NY, USA, 1999. ACM.
- [21] Benjamin Han and Alon Lavie. A framework for resolution of time in natural language. In *ACM Transactions on Asian Language Information Processing (TALIP)*, pages 11–32, 2004.
- [22] Carl Hewitt. Planner: A language for proving theorems in robots. *Proceedings of First IJCAI*, pages 295–301, 1969.
- [23] Jerry R. Hobbs and Feng Pan. An ontology of time for the semantic web. *ACM Transactions on Asian Language Information Processing (TALIP)*, 3(1):66–85, 2004.

- [24] M. James and L. Dubon. An autonomous diagnostic and prognostic monitoring system for NASA's deep space network. In *Aerospace 2000 Conference*, 2000.
- [25] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
- [26] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
- [27] Heikki Mannila and Dmitry Rusakov. Decomposing event sequences into independent components. In *Proceedings of the First SIAM Conference on Data Mining*, pages 1–17. SIAM, 2001.
- [28] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.
- [29] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.
- [30] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.
- [31] Marvin Minsky. A framework for representing knowledge. Technical report, Cambridge, MA, USA, 1974.
- [32] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31:71–106, 2005.
- [33] Edwin P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge repre-*

sentation and reasoning, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

- [34] Vasco Calais Pedro. From text to actions, semantic reasoning within closed domains. Master's thesis, Language Technologies Institute, Carnegie Mellon University, 2004.
- [35] Steven Pinker. *The Stuff of Thought: Language as a Window into Human Nature*. Viking Adult, September 2007.
- [36] Gilbert Ryle. *The Concept of Mind*. The University of Chicago Press, 1949.
- [37] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In *IJCAI*, pages 412–422, 1973.
- [38] Karin Kipper Schuler. *VerbNet: A Broad-Coverage, Comprehensive Verb Lexicon*. PhD thesis, Computer and Information Science, University of Pennsylvania, 2005.
- [39] G. H. von Wright. *Norm and Action. A logical Inquiry*. Routledge and Kegan Paul, London, 1963.

Appendix A

Event Monitoring and Event Detection Helper Functions

Function 10: propagate-from

Input: *from*: the subevent to propagate the constraints from

Input: *process*: the process to propagate the constraints to

Output: *process2*: the updated *process*

process2 \leftarrow (without-subevents *process*)

process2 \leftarrow (apply-constraints *from process2*)

foreach *subevent_i* \in *process* **do**

if *subevent_i* = *from* **then** *process2* \leftarrow (add-subevent *process2 from*)

process2 \leftarrow (add-subevent *process2* (apply-constraints *from subevent_i*))

return *process2*

Function 11: apply-constraints

Input: *from*: the event to get the constraints from

Input: *process*: the process to propagate the constraints to

Output: *process2*: the updated *process*

process2 \leftarrow *process*

foreach *unification_i* \in (get-unifications *from process*) **do**

foreach *constraint_j* \in (get-constraints *from*) **do**

if (constraint-type *constraint_j*) = (first *unification_j*) **then**

c2 \leftarrow (list (second *unification_i*) (constraint-value *constraint_j*))

process2 \leftarrow (add-constraint *process2 c2*)

return *process2*

Function 12: check-structure

Input: *candidate*: event to check

Input: *event*: the event that happened last

Output: if the structure matches the updated *candidate*, NIL otherwise

if *candidate* has a subprocedure **then**

switch (expansion-type *candidate*) **do**

case :SEQ

$response \leftarrow (\text{check-structure}(\text{next-subevent } candidate) \text{ event})$

if $response == NIL$ **then return** NIL

else

 mark (next-subevent *candidate*) as done

if (next-subevent *candidate*) == NIL **then** mark *candidate* as done

return *candidate*

case :OR

foreach $subevent_i \in candidate$ **do**

$response \leftarrow (\text{check-structure}(\text{next-subevent } candidate) \text{ event})$

if $response \neq NIL$ **then**

 mark $subevent_i$ as done

 mark *candidate* as done

return *candidate*

return NIL

case :AND

foreach incomplete $subevent_i \in candidate$ **do**

$response \leftarrow (\text{check-structure}(\text{next-subevent } candidate) \text{ event})$

if $response \neq NIL$ **then**

 mark $subevent_i$ as done

if (n-incomplete-subevents *candidate*) == 0 **then**

 mark *candidate* as done

return *candidate*

return NIL

else if $event == candidate$ **then**

 mark *candidate* as done

return *candidate*

else return NIL
