Abstract

This thesis presents a framework for representing and reasoning with temporal data in the TrollCreek CBR-system. Supplementing TrollCreek with this ability will aid in prediction tasks, that is: Foreseeing what will occur in a new problem situation based on comparisons with stored situations.

The representation is based on embedding one or several *time lines* within a case. The reasoning mechanism abstracts these time lines into one, and compares this abstracted time line with other abstracted time lines in the case-base. In this thesis we have implemented a simple non-knowledge-intensive abstraction scheme and used sequence comparison methods for the comparison of time lines.

The thesis also contains an example of the framework in use. The example is of a proof-of-concept type, and does not involve a real-world scenario.

Acknowledgements

I would like to thank my supervisor Agnar Aamodt for his help during during the writing of this thesis. Thanks also go out to Frode Sørmo for helping in figuring out how the TrollCreek-code is structured.

Contents

Chapter 1

Introduction

1.1 Goals

The primary goal of this thesis is to develop a method for reasoning with temporal data within the Case-Based Reasoning (CBR) system TrollCreek. In more detail this means developing a framework for representing temporal data and a method of analyzing and comparing sets (cases) of temporal data.

The primary goal is composed of the following subgoals:

- Develop a method for reasoning with temporal data using CBR by drawing on related work done at NTNU and internationally.
- Implement the core-components of the method in the current version TrollCreek.
- Test and evaluate the method.

1.2 Background and motivation

Case-based reasoning is one of the main areas of research at the Artificial Intelligence and Learning Group (AIL) here at NTNU. Much work is done with the CBR-system TrollCreek, developed at NTNU and built on the Creek architecture developed by Agnar Aamodt [Aam91].

Traditionally CBR-systems have not utilized temporal data, and TrollCreek has not been an exception. What is contained in a case is viewed as a snapshot of the current situation, and this is sufficient for many purposes.

One area of research where the TrollCreek system is used is within the prediction of unwanted events in oil drilling. Such a prediction task will benefit if the underlying system has the ability to represent and reason with temporal data.

This thesis is not the first endeavour done at AIL to use temporal data in CBR. Martha Dørum Jære developed a framework for this in her master thesis [Jær01]. This framework was implemented in JavaCreek, which was a predecessor to TrollCreek. The framework was carried over to the current version of TrollCreek in a project done during the autumn of 2004 [Bre04].

The work done in this thesis will build on the experiences made from implementing the temporal framework developed by Jære in TrollCreek, but it will not extend this particular framework. The work will also draw on ideas from related research internationally.

1.3 Methods of investigation

The main focus of this thesis is on developing a framework for temporal reasoning within TrollCreek. The framework should not just exist in theory, and we will implement crucial parts of it in the current version of TrollCreek to show that the ideas presented can be implemented in practice. To illustrate our ideas in practice we need a test domain. We will not do a performance analysis of the framework within the test domain, but will instead concentrate on describing how the framework functions.

1.4 Structure of the thesis

The thesis will start off with a chapter that gives a short description of CBR in general, and a more thorough description of the TrollCreek system. Chapter 3 contains descriptions of other work done within temporal CBR, and which has served as sources of ideas for our framework. Chapter 4 documents the work done in selecting a domain and dataset to test our framework. In Chapter 5 we begin describing the temporal framework. We start with the representation, and in the next chapter we move on to the reasoning mechanism. In Chapter 7 we give an example of the framework in use. The thesis is wrapped up with a chapter that discusses issues that have arisen in the development of the framework and some pointers to further work.

Appendix A contains information on what has been changed and added to the existing jCreek-code. This is meant as a help to anyone that needs to figure out what has been done to code. In addition to the appendix the JavaDoc may also be helpful for this purpose.

Chapter 2

CBR and TrollCreek

In this chapter we will give a short overview of CBR in general and the TrollCreek system in particular.

2.1 Case-Based Reasoning

The main idea behind CBR is to compare a new problem with previous problems where we have found the solution, and that we in addition have stored in what is called a case-base. An unsolved problem is called an unsolved case, and solved problems are called solved cases. The case-base thereby becomes the "intelligence " of the system. The reasoning is based on the assumption that cases that in some way are judged to be similar to the new problem have solutions that may be applied to the new problem too. The cases that are most similar to the new problem are retrieved from the case-base and used to construct a solution. This kind of reasoning is apparent in much of human problem solving:

- A physician may after having examined a new patient recall a previous patient with similar symptoms, and he may then apply that patient's diagnosis to the new one.
- Just replace *physician* with *car mechanic* and *patient* with *car* in the above example and you have a new scenario where humans use case-based reasoning.

By reusing solutions to earlier problems we might expect that CBR can be based on shallow knowledge, and as a consequence of this require less knowledge engineering than other AI approaches [Cun98]. Even though CBR-systems often are applied in domains where the domain knowledge is incomplete, there have been done many efforts to incorporate what domain knowledge there is into CBR-systems. TrollCreek, as we shall see, is an example of a system that will use domain knowledge if it is present.

2.1.1 The CBR-cycle

There are variations from system to system regarding how cases are retrieved, how the solutions are constructed and so on. A general framework for describing

CBR-systems have been developed by Aamodt and Plaza [AP94]. The framework can be viewed as a cycle, and a graphic overview is given in Figure 2.1.

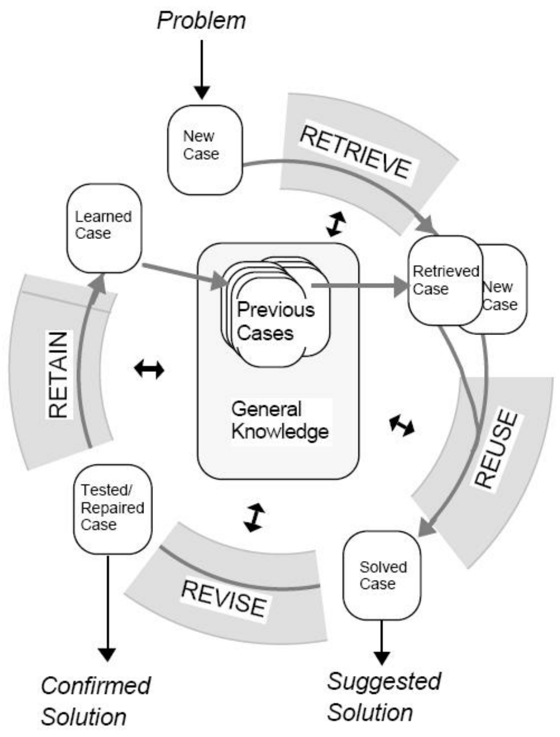


Figure 2.1: The CBR-cycle (the figure is taken from [AP94])

The CBR-process is decomposed into four subtasks:

Retrieve. Cases that are similar to the input case are retrieved from the case-base. A method that computes the degree of similarity between two cases is needed under this step. It is this step that has received the most attention in the history of CBR, and it shows in the number of different methods used to retrieve similar cases.

Reuse. The solutions of the retrieved cases may need some modifications to apply to the input case. The task of the Reuse-step is to modify the solutions of the retrieved cases so that they apply to the new problem.

Revise. Here the solution is evaluated. Did the solution work? If not; what went wrong? If the solution did not work steps may be taken to repair it.

Retain. Here the decision is made whether the new case, or something from it, should be added to the case-base. It is under this step the system has the ability to learn from experience. It does this by retaining information gathered from solving a new problem.

These are the four general steps in the CBR-process. As mentioned the actual implementation is different from system to system, and all steps may not be included. For instance, a CBR-system may just consist of the Retrieve-step and none of the rest. This system would have a static case-base and return cases that are similar to the input case, but it would not try to adapt the solutions of the retrieved cases.

2.2 TrollCreek

We now turn from CBR in general to TrollCreek. Since TrollCreek is the product of a long process that has produced several CBR-systems, we will to avoid possible confusion start off by presenting some definitions:

- Creek: An architecture for knowledge intensive problem solving and sustained learning. This architecture was developed in Agnar Aamodt's PhD-thesis [Aam91].
- CreekL: The Creek representation language.
- LispCreek: The first system built on the Creek architecture. This system implemented the whole CBR-cycle, and was written in LISP.
- JavaCreek: An early Java-implementation of the Creek architecture.
- TrollCreek: A CBR-system implemented in Java that builds on the Creek architecture. This is the system we will be working with in this thesis.
- jCreek: Term used to refer to the code that TrollCreek is built on.

The Creek architecture which TrollCreek is built on is characterized by a strong coupling between cases and domain knowledge, and is targeted at open and weak theory domains [Aam04]. The Creek architecture is knowledge-intensive, which means that the case matching, and reasoning in general, uses domain knowledge. As a consequence the similarity assessment between cases is based on more than just looking for directly matching features. How Creek relates to less knowledge-intensive systems can be seen in Figure 2.2.

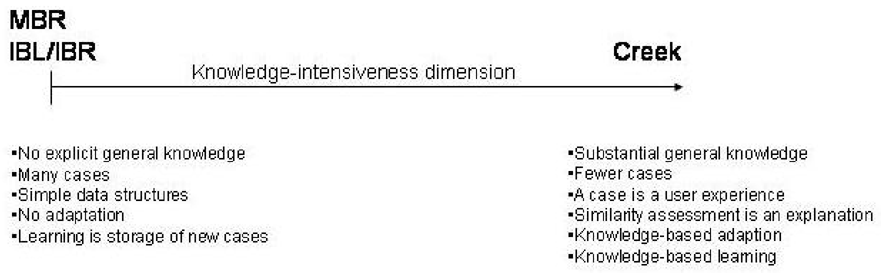
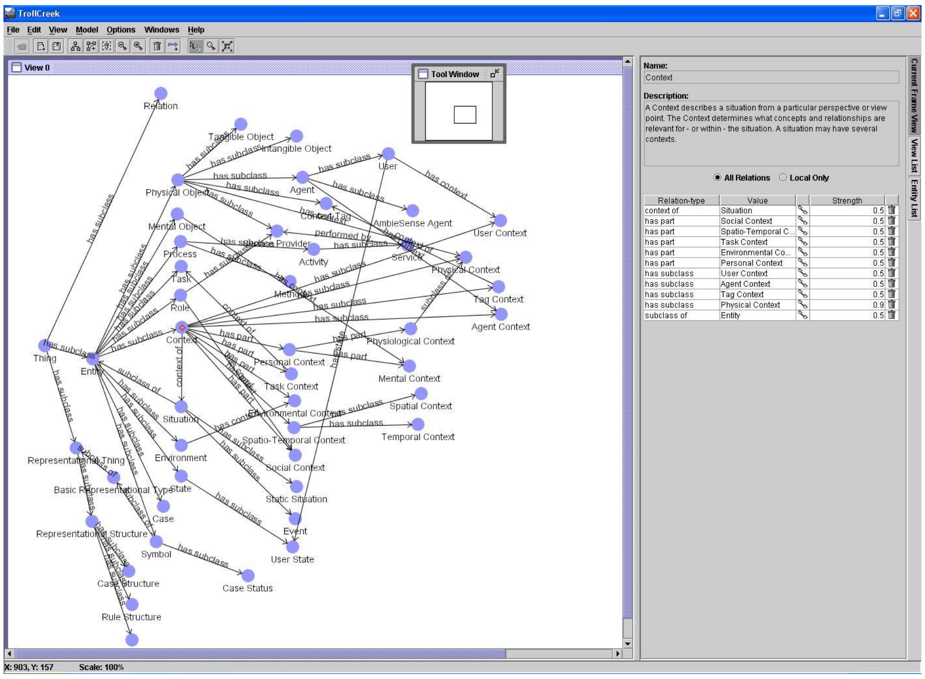

MBR
IBL/IBR
Creek
Knowledge-intensiveness dimension
•No explicit general knowledge
•Many cases
•Simple data structures
•No adaptation
•Learning is storage of new cases
•Substantial general knowledge
•Fewer cases
•A case is a user experience
•Similarity assessment is an explanation
•Knowledge-based adaption
•Knowledge-based learning

Figure 2.2: Knowledge-intensiveness dimension (adapted from [Aam04])

The strong coupling between cases and domain knowledge can be seen in the way they are modelled. Both are modelled in a single semantic network, called a knowledge model. This increases the knowledge-intensiveness of the cases since their features are nodes in the semantic network. The nodes in the network are concepts in the knowledge model, and the links between the nodes correspond to relations between concepts.

Concepts may be general or prototypical concepts, cases, heuristic rules and so on. An excerpt from a knowledge model is seen in Figure 2.3. In addition to viewing a concept as a node in a semantic network, it can also be seen as a frame.



Name:
Context

Description:
A Context describes a situation from a particular perspective or view point. The Context determines what concepts and relationships are relevant for - or within - the situation. A situation may have several contexts.

[✓ All Relations] Local Only

Relation-type	Value		Strength	
context of	Situation		0.5	
has part	Social Context		0.5	
has part	Spatio-Temporal C...		0.5	
has part	Task Context		0.5	
has part	Environmental Co...		0.5	
has part	Personal Context		0.5	
has subclass	User Context		0.5	
has subclass	Agent Context		0.5	
has subclass	Tag Context		0.5	
has subclass	Physical Context		0.9	
subclass of	Entity		0.5	

Figure 2.3: Excerpt from a knowledge model. The figure is taken from the TrollCreek knowledge model editor

In addition to a framework for representing knowledge the Creek architecture also specifies inference methods that operate on the representation. This includes an inheritance method for properties of concepts which uses *plausible inheritance* [Sør00]. For instance are all properties belonging a concept inherited by another concept that has a *has sublass*-relationship with the former. Other inheritance rules may be specied by the user.

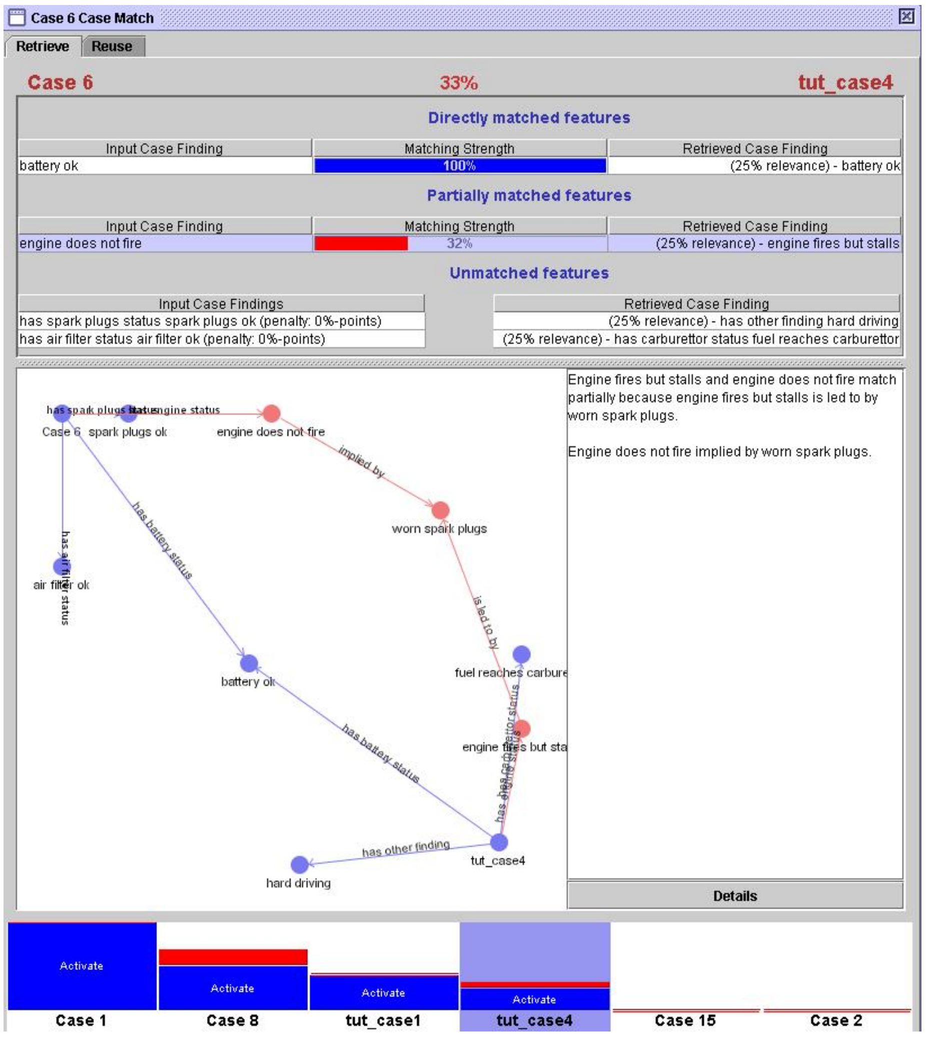
Creek also comes with a top-level ontology that further knowledge modelling needs to adhere to. For instance are cases intended to be subclasses/instances of the predefined concept *Case*, new relation types to be subclasses/instances of the *Relation* concept, and so on.

The relations in the semantic network are weighted. This means that they have a value between 0 and 1 (inclusive at both ends) associated with them. This value is called the *explanation strength* of the relation. How it is interpreted depends on the type of relation. Some relation types actually do not have a clear definition of what the value means. The weights are used in the case matching as we shall see.

TrollCreek also comes with a knowledge model editor. This editor provides GUI-elements for manipulating and viewing the contents of a knowledge model. The GUI-elements are disassociated from the reasoning elements. What this means is that the actual code that does the reasoning, and knowledge representation for that matter, is not dependent on the GUI-elements to function. Instead the GUI-elements provide an user-friendly interface to the underlying reasoning mechanisms.

2.2.1 Case matching

A case is as everything else represented as a node in the semantic network. It is described by its findings, which are the concepts that the case has *has finding*-relations to. Figure 2.4 shows how the results from a case matching is displayed in the knowledge model editor.



Case 6 33% tut_case4

Directly matched features

Input Case Finding	Matching Strength	Retrieved Case Finding
battery ok	100%	(25% relevance) - battery ok

Partially matched features

Input Case Finding	Matching Strength	Retrieved Case Finding
engine does not fire	32%	(25% relevance) - engine fires but stalls

Unmatched features

Input Case Findings
has spark plugs status spark plugs ok (penalty: 0%-points)
has air filter status air filter ok (penalty: 0%-points)

Retrieved Case Finding
(25% relevance) - has other finding hard driving
(25% relevance) - has carburettor status fuel reaches carburettor

Engine fires but stalls and engine does not fire match partially because engine fires but stalls is led to by worn spark plugs.

Engine does not fire implied by worn spark plugs.

Figure 2.4: Case matching

When a new unsolved case is added to the knowledge model and a case matching is run, TrollCreek tries to find similar cases by matching the findings of the new case with those of the solved cases in the case-base. This process consists of three steps and can be described using the *activate-explain-focus* cycle [Aam93]. This cycle is actually a generic "explanation engine" that has been specialized for each of the tasks in the CBR-cycle (see Figure 2.5).

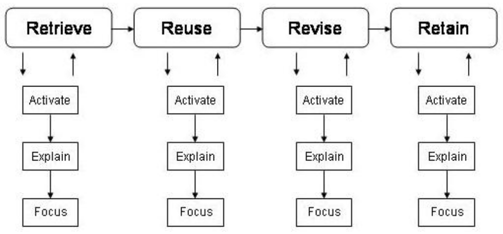


Figure 2.5: Relationship between the CBR-cycle and explanation engine

The steps operate as follows:

- *Activate.* Activates relevant parts of the semantic network.
- *Explain.* Uses the activated part from the previous step to generate explanations between concepts.
- *Focus.* Focuses in on and selects a conclusion that conforms with the goal.

We will now elaborate on how each of the steps are implemented in TrollCreek (we will concentrate on the Retrieve-step of the CBR-cycle).

The Activate step uses spreading activation along taxonomic, causal, associational and application-specific relations to establish a context for the problem. In practice this means that TrollCreek searches for directly matching findings between the input case and a case in the case-base: If the input case has the finding *green* and one of the cases in the case-base also has this finding, it counts as a direct match. How the strength is computed is shown in Equation 2.1.

$$sim(C_{IN}, C_{RE}) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} sim(f_i, f_j) * \text{relevance factor}_{f_j}}{\sum_{i=1}^{m} \text{relevance factor}_{f_j}} \tag{2.1}$$

C_{IN} and C_{RE} are the input and a retrieved case respectively. For direct matches, which are the only ones found in the Activate step, the function $sim(f_i, f_j)$ returns 1. The relevance factor is value that is a combination of the predictive strength and importance of a feature for a stored case. We will not go into what predictive strength and importance are, just suffice to say that these values are represented as explanation strengths of the *has finding* relations that attach findings to a case.

Cases with activation strengths above a given threshold are carried over to the Explain step. Here TrollCreek tries to match findings that were left unmatched in the Activation step. Again this is done by using spreading activation. Here the $sim(f_i, f_j)$ function is more elaborate. The value returned by it is determined by the length and strength of the explanation paths between the two features that are compared. Finally the Focus step selects those cases that are most similar to the input case.

In addition to the aiding the matching process through explanation paths, the domain knowledge may also provide a "backup" problem solving capability if no cases similar to the new problem are found.

The research related to the development of TrollCreek has so far concentrated on the Retrieve-step of the CBR-cycle. The Reuse-step is still in an early stage, and Retain and Revise have not been implemented yet.

2.2.2 Comparators

When two concepts (nodes in the semantic network) are to be compared, TrollCreek needs to know how to compare them. A comparison between two numbers is done in a different manner than a comparison between two cases. Some concepts cannot be compared either, for instance a number and a case.

To know how to compare concepts, the nodes that represent them in the network must have *comparators* attached to them. A comparator is just as everything else represented as a node in the knowledge model. Comparators are subclasses of the *Comparator* concept that is defined in the top level ontology in Creek. Comparators encapsulate a Java class that does the actual comparison. Only nodes (concepts) that have the same comparator can be compared. It is necessary to know this when we later describe how the reasoning mechanism in this thesis is implemented.

A comparator for a concept is specified using the *has comparator* relation. In Figure 2.6 we see the comparator specification for the concepts *Symbol* and *Case*.

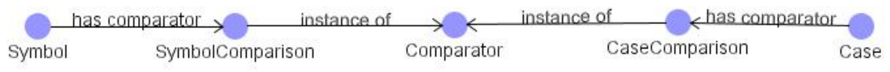


Figure 2.6: Comparator-structure in TrollCreek

Chapter 3

Related work

In this chapter we will present work others have done within temporal CBR. This includes the earlier work done in JavaCreek/TrollCreek. We will go into more detail regarding the systems that have provided us with ideas for our framework, than with those that have not influenced us. The former will each receive a section, while we will do away the latter right now:

Branting and Hastings' CARMA system [KLBL97]. CARMA is a rangeland grasshopper management advising system. This means that it comes with recommendations as to the best way to battle grasshoppers that consume the forage. CARMA uses both case-based and model-based reasoning. A solved case is a "grasshopper situation" that an expert has analysed and provided with a solution (i.e. apply pesticides at a certain period). A new case lacks the expert advice part. The task of the case-based reasoning is, of course, to select the best matching cases in the case-base. The date is one of the attributes that is stored within a case and is used in the retrieval, along with other features. Any differences between the best matching input case and the best matching case are "explained away" using model-based adaptation (this would be placed under the Reuse-step in the CBR-cycle). This adaptation of the best matching case involves explaining how the date differences of two cases affects the predicted forage consumption. This is what can be called the temporal reasoning part of the system.

Bjarne Hansen's WIND-1 system [HR01]. WIND-1 has its origin in Hansen's master thesis [Han00]. The task is to predict the weather. An input case is a series of weather observations, and this sequence is matched against those stored in the case-base. The cases have fixed length, 24 hours to be more specific. The temporal span is divided into three parts: the 12 recent hours, the current hour and the 12 future hours. A new problem (case) is missing the future part, and it is this the system attempts to construct.

Ram and Santamaría's Self-Improving Navigation System (SINS) [RS97]. The system deals with robot navigation, more specifically: Controlling the engines of a small mobile robot by examining the readings from its sensory inputs. The SINS system addresses some specific issues. The representation is continuous, which means that a case contains an array of variables that record the sensory inputs of the robot over a period of time. The variables are all numeric. The SINS system is also meant to run in real-time. As the robot moves around the environment sensory inputs are matched continuously against the cases in the case-base. This calls for an efficient way of retrieving and matching cases.

3.1 Ceaseless Case-Based Reasoning

A short description of the Ceaseless CBR-architecture is given in [MP04]. For a full description consult Francisco J. Martín's PhD-thesis [Mar04], which is the basis for the article.

The Ceaseless paradigm has some features that sets it apart from most traditional CBR-systems, in particular:

- Its input is an unsegmented sequence of events that is constantly evolving.
- The events in the sequence can stem from multiple sources.
- Several cases can occur at the same time.

To test the framework Martín applies Ceaseless CBR to the task of intrusion detection in a computer network. This domain offers the possibility of testing the special features of Ceaseless CBR. The input to the system is an ever-evolving sequence of alerts coming from multiple computers in a network.

A case in the case-base describes a specific kind of intrusion attempt. More specifically a sequence of events, which if present in the log may point to an attack in progress. Cases are described in the form of what is called *actionable trees*.

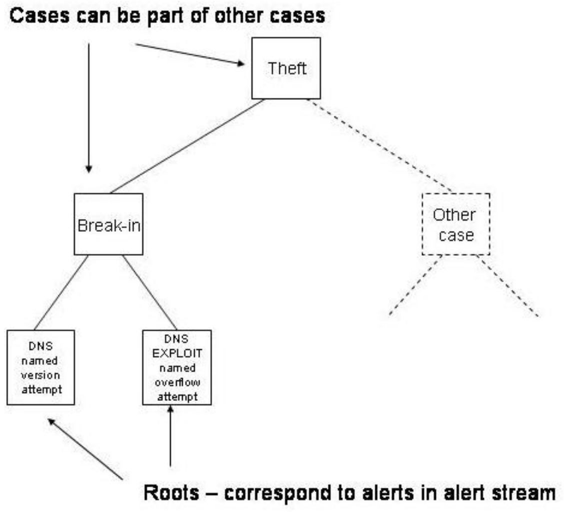

Cases can be part of other cases
Theft
Break-in
Other case
DNS named version attempt
DNS EXPLOIT named overflow attempt
Roots – correspond to alerts in alert stream

Figure 3.1: Actionable trees

A key point is to detect an attack in an early stage so that the network administrator has time to do something about it. On the other hand if the system produces too many false alerts it loses its value.

Even though Ceaseless CBR differs from traditional CBR-systems on several points, it is still implemented in accordance with the CBR-cycle. Indeed, all the steps in the CBR-cycle have been implemented.

Retrieve. As mentioned the input is an evolving sequence of alerts. The systems segments this stream as it arrives. The segmentation is done according to a window model. It can for instance select all alerts the occurred between 12.00 and 12.10. The sequence of alerts is continually compared with the cases in the case-base. The Retrieve-step also keeps a collection of case activations that represent the current situation. A case activation is a hypothesis that says how likely it is that a case has occurred.

To obtain the case activations the window of alerts is analyzed. The cases in the case-base are aligned to the sequence of alerts using edit distance measures. The first sequence is a sequence of alerts that is a case, and the other sequence is the alerts in the selected window. A good match leads to a high activation value for the case. The activation value is also dependent on the rarity of the alerts that make up a case. Rare alerts lead to higher activation values.

One has to remember that many cases share alerts, and some cases may be subcases of others. This means that there may be several different case activations for a single case, and some alerts may be involved in several case activations. The output of the Retrieve-step is a set of case activations that serve as input to the Reuse-step.

Reuse. The Reuse-step searches for the combination of case activations that best explains the most recently received alerts. A combination of case activations that explain all alerts is called, logically enough, an explanation. This search for the best explanation is quite complicated, and involves the use of a belief function to determine which case activations are susceptible of being used to prioritize the corresponding alerts.

Due to the fact the input is always evolving, a case may be spread over several windows. To account for this the case activations must a stored as the window slides over the alert stream. Case activations must also be deleted as the window moves forward. If they were not the system would start generating many false alarms after a while. The strength of the case activations decrease as the window moves forward, if not any new evidence backing up the case activation is found.

Revise. Here the best explanation of the alerts is presented to the user. The user may at this stage revise the combination of case activations that is being proposed as the best explanation.

Retain. After the solution has been revised by the user the Retain-step updates the case-base.

3.1.1 Discussion

The Ceaseless CBR-model differs from other systems in that the input is a constantly evolving sequence of events, not a clearly defined unsolved case. This is a central concept in Ceaseless CBR, and affects much of the design of the framework. We will in this thesis work with the traditional idea where the input is in the form of a case with clearly defined borders. Should this situation be represented in Ceaseless CBR, it would mean we had a static input sequence. The framework would not be utilized fully, but it would still function.

We can also point out the fact the actionable trees (that represent the cases) are very concrete. It is not clear how the system would hold up if the cases in the case-base contained noise.

3.2 TempoExpress

TempoExpress is a CBR system that performs tempo transformations on melodies. This may at first not sound like a task where CBR could be of use, but the process of altering the tempo involves more than just speeding up or slowing down the tune. When a musician performs the same musical piece at different tempos there will be changes in other aspects of musical expression too, for instance addition or deletion of notes, changes in timing and so on. TempoExpress aims to include such changes when altering the tempo. To stop any potential objections against the rationale that a musician adds or deletes notes when performing a piece, we will point out that TempoExpress has been used on jazz-melodies, not classical works. In jazz such alterations are an essential part of the performance.

The most up to date description of TempoExpress can be found in [GAdM04]. An earlier version is described in [AGdM03], and it is interesting to see what has been changed between that one and the current, which is the one we will be focusing on.

A new case is a performance of a piece of music that is to undergo a tempo transformation. In addition to the actual recording of the performance, a MIDI-encoding of the performance and a desired tempo is needed. These three elements form the basis of a case. The case-base contains stored performances, and these are used as "guidelines" for how to transform the input case.

Music obviously includes a temporal dimension, and in retrieving cases from the case-base and adapting them to the problem at hand TempoExpress performs a form of temporal reasoning.

In the Retrieve step the cases are first filtered on the tempo value. Only cases that have a tempo that lie sufficiently close to the input case are retrieved, this set of cases then go through a second filtering process. Here an edit distance approach is used to give a measure of a similarity between the input case and each of cases retrieved in the first step. In the current version of TempoExpress the edit distance calculations are not done directly on notes of the score in the Retrieve-step, as opposed to the approach in [AGdM03], but instead on an abstraction of the score. The abstraction is based on what is called the "Implication/Realization" model, which is a theory of perception and cognition of melodies.

In the Reuse step a performance of the input case at the desired tempo is constructed. This is achieved by using constructive adaptation [PA02].

3.2.1 Discussion

The temporal aspect is not as explicitly treated in TempoExpress as in the other systems. The focus is on melodies, not on how to represent or reason with time in a more general fashion. Since melodies have length some temporal framework is needed, and the ideas incorporated in this framework can be generalized and applied to other domains. We have for instance seen that Ceaseless CBR uses sequence comparison methods as TempoExpress does, and we will see below that the abstraction according to the I/R-model has some similarities with what is done in ICONS, which is described later.

3.3 Temporal framework for CBR*Tools

Michel Jaczynski has developed a framework for including temporal information in CBR. This framework is described in [Jac97], and is implemented in CBR*Tools, which is a software library for CBR-tasks developed by Jaczynski and Brigitte Trousse.

The framework uses time series to represent temporal information. One or several related time series can be stored in what is called a record. The record together with its context constitutes a case. The record's context contains additional data not expressed in the time-series, i.e. a patient's name if we imagine the framework being applied in a medical domain.

Time series can be of two types: sampled and event-based. In sampled time series there is equal spacing between two neighbouring points, while event-based have an arbitrary spaces between points. The values represented at the points can be numerical or symbolic. How the comparison actually is done is left to the user.

A domain expert may analyze the time series in a case and specify *elementary behaviours*. These basically point out something important in the record, and are grounded in reference dates. The elementary behaviours are used in the case comparison. An example of a case record is shown in Figure 3.2.

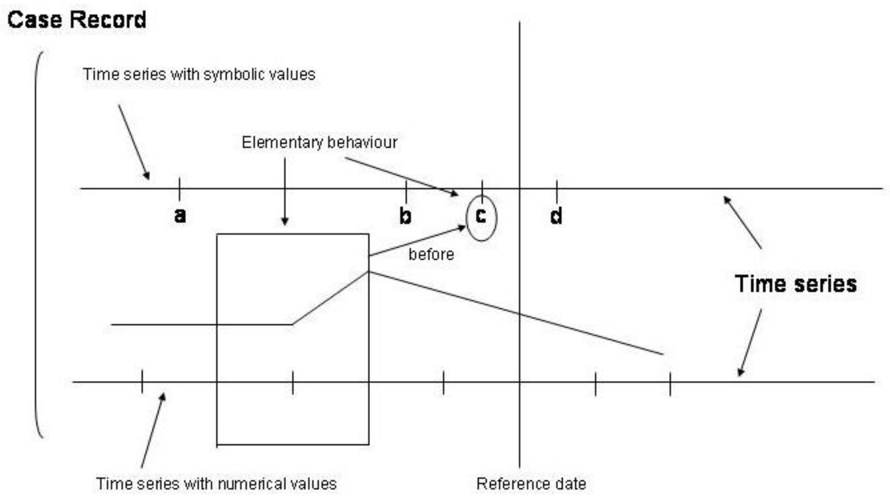


Figure 3.2: A case record

Cases can be of three types:

- Abstract cases. The cases were we are most certain are "correct." They come from domain knowledge. Jaczynski also uses the term domain scripts when describing abstract cases.
- Concrete cases. These are what we normally think of when we talk about cases. They are concrete situations that have been recorded.
- Potential cases. These cases do not have a concrete representation. They are represented by templates, and the system searches through concrete

cases trying to find sections that match the templates. If such a section is found a concrete case is created.

When retrieving similar cases the system first looks for matching abstract cases, then concrete cases and finally if no matching case is found it uses potential case templates to search for behaviour in the input case that fulfils the template. If such behaviour is found a new concrete case based on the input case is created.

The article includes an example of the framework in use dealing with plant nutrition: The point is to adjust the amount of nutrition mixture given to a plant based on a history of several parameters (i.e. maximum day temperature). We are not told if the plant in question was satisfied with the rationing of the nutrition mixture.

3.3.1 Broadway

A larger example that uses the framework is given by Jaczynski and Trousse in [JT99]. The paper describes the Broadway system, which is a browsing advisor for the Internet. A browsing advisor gives the user advise on what pages he might want to visit. In Broadway this advice is mainly based on past navigations of users.

The temporal framework is used to record the order of pages visited. Actually four different parameters that evolve over time are recorded as part of the navigation:

1. The URL of the document.
2. Content description in the form of keywords in the title.
3. The user's evaluation of the document.
4. The time the user spent viewing the document. This is measured relatively to the document's size.

Here we see that the framework's ability to include multiple time series is utilized. To make cases a situational template (potential case), searches through the recorded navigations and creates concrete cases as it finds behaviours that satisfy the template. A concrete case references a navigation at a specific moment. This moment represents the division between past and future, and the future pages of a concrete case that the user has evaluated as relevant or highly relevant, constitute the set of advised pages for that case.

The retrieval process generally follows those strategies as laid out in [Jac97]. Similarity measures between the different parameters in a navigation need to be implemented for this domain specifically, as would be expected since the framework left such similarity measurements up to the user.

The original framework paper only focused on the Retrieve-step, but Broadway also has a Reuse-step implemented. In the Reuse-step the k most reusable pages gathered from the Retrieve-step are selected and ordered according to a metric called *reusability*. The reusability value for a page is based on several features of a page. To name some: The number of cases that advise the page, the best similarity of the cases that advise the page and the number of successful accesses to the page.

3.3.2 Disucussion

The paper presents both a general framework and its implementation into an existing CBR-system. It is not as closely connected to a specific domain as most of the other surveyed systems are.

Domain knowledge is used to create abstract cases, potential cases and elementary behaviours within concrete cases. The domain knowledge is used in quite another way than in the other systems surveyed. It does seem like a bit of manual effort is needed to "groom" the cases so that the system will produce good results.

The process of identifying elementary behaviours bears some resemblance to a temporal abstraction task. Each elementary behaviour can be viewed as an abstraction task whose goal it is to identify the situation stored in it. This may not sound to clear at this stage, but we will come back to the subject of temporal abstraction later, and then the similarities between abstractions and elementary behaviours may become clearer.

3.4 The ICONS-project

The ICONS system is developed by Rainer Schmidt and colleagues and is described in [SHPG96]. The goal is to predict kidney failure for patients in an intensive care unit. It attempts to do this by analyzing data collected from the patient, comparing this data with that of other patients in the case-base, and finally retrieving those cases that are most similar.

The data collected from the patient comes from a monitoring system (NIMON), which supplies 13 measured and 33 calculated parameters relating to the renal function. Based on these parameters a state describing the kidney function is calculated (i.e. reduced kidney function). In other words the parameters are being abstracted into a single state that describes the kidney function for that. A chain of such states for a patient constitutes a case.

By abstracting all of the parameters into a single state, the reasoning becomes much easier. The system now only has to look at how one parameter evolves from one day to the next.

To compare cases they use what they call *trend descriptors*. They use four descriptors, and each one is made by domain experts. So what is exactly is a trend descriptor? It says something about how the trend evolves. Two of the descriptors used in the system look like this:

- T1: This is just the state at the current moment.
- T2: This looks recursively back in the case from the current state trying to find a continuous trend.

Cases that have similar trend descriptors are judged to be similar. The trend descriptors are made on the basis of what domain experts look for when they analyse the data. This means that the trend descriptors are domain dependent. An example of a case and trend descriptors is shown in Figure 3.3.

To speed up the retrieval the cases are stored in a tree structure (reminiscent of a suffix tree), where each node is the value of some trend descriptor. We will not go into any more details about this here.

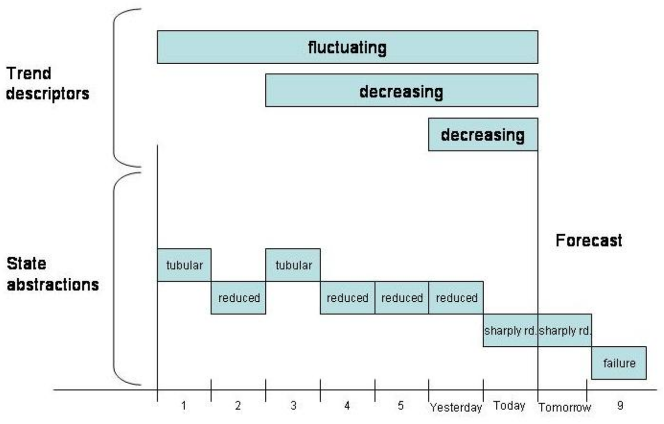


Figure 3.3: A case and its trend descriptors

Before we round off our description of this system we will answer an important question: What is the current moment in an old case? A case has a maximum length of ten days. Seven days are treated as a history and the remaining days as a forecast. This means that the current moment can be one of three days in an old case.

3.4.1 TeCoMed

The team behind ICONS has developed another CBR-system that is worth mentioning here; the TeCoMed project which is described in [SG02]. Here the goal is to compute early warnings against influenza waves. The case-based reasoning is done in the same manner as in ICONS, so we will not go into the details of this system.

3.4.2 Discussion

Temporal abstraction is central in ICONS and TeCoMed. The systems depend on the fact that the contemporaneous parameter values under consideration are abstracted into one state. The abstraction and case matching are actually quite separate. The matching mechanism could be used without the abstraction if the original data only consisted of a single time line.

The ability of the framework to function in more than one domain is proved by the fact that it has been implemented in two projects.

3.5 Historical Case-Based Reasoning

Jixin Ma and Brian Knight have developed a framework temporal CBR which they call “Historical Case-Based Reasoning” [MK03]. The framework is grounded in a many-sorted reified logic, and builds on a time theory previously developed by Ma and Knight [MK94]. The framework supports both time points and intervals. Time points have zero duration, while intervals have a duration that is represented by a non-negative real number. Points and intervals are related to other points and intervals by relations, of which there are 30 different types. Inspection of the set of relations shows that the framework is related to that developed by James F. Allen [All83], which we will return to later. The set of relations used in both frameworks are similar, but while Allen only deals with intervals, and hence the relations only connect intervals with intervals, Ma and Knight deal with intervals and points. This means that their set of relations can be classified into four groups:

- Equal, Before, After. Relate points to points.
- Before, After, Meets, Met by, Started by, Contains, Finished by. Relate points to intervals.
- Before, After, Meets, Met by, Started by, Contains, Finished by. Relate intervals to points.
- Equal, Before, After, Meets, Met by, Overlaps, Overlapped by, Starts, Started by, During, Contains, Finishes, Finished by. Relate intervals to intervals. This is by the way the set of relations that Allen uses.

To describe what happens and endures at different times, the term *fluent* is used. A fluent denotes something that holds true over a period of time. Fluents are not directly connected to points and intervals. Zero or more fluents are attached to what is called an *elemental case*, and this elemental case is in turn connected with time elements.

What is usually thought of as a case is called a *case history* in this framework. A case history consists of:

- A set of predicates (in the logic sense) that defines which elemental cases that hold at different time elements.
- A number of time elements; points and intervals in other words.
- A set of "meets" relations that tells which time elements that meet each other. All of the 30 relations that can hold between points and intervals can be expressed in some form using only the meets relation, so to make things simpler this is the only relation used in the paper.
- A set of durations that tells how long of some the time elements last.

A graphical example of a case history is shown in Figure 3.4.

Those are the elements of the representational part, now we turn to the reasoning. The aim is to give a similarity measure between two case histories. This similarity measure consists of two parts: A non-temporal similarity degree and a temporal similarity degree.

The non-temporal similarity degree is based on elemental case matching. The matching is done by finding the pairing of elemental cases from the two histories that gives a maximum score. This basically means that if two elemental cases share many fluents they get a high score, otherwise a low score.

The temporal part is to be done by using conventional graph similarity measurement methods. Such methods can be used since a case history can be viewed as a graph.

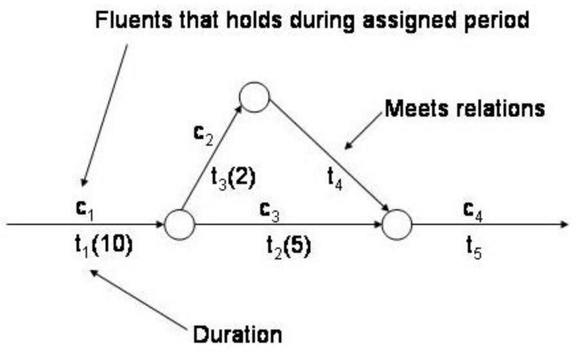


Figure 3.4: Graphical case history

3.5.1 Discussion

The framework does not seem to have been implemented in any real world system yet, and it seems that a number of questions must be resolved before it will be. For instance it is not clear what graph similarity metric should be used.

It seems that the framework does not aim to be knowledge intensive. The non-temporal matching scheme just looks at the presence or absence of fluents without going into more depth, and the temporal matching scheme will probably follow a similar strategy.

The framework is similar to the one previously developed in JavaCreek in that the representation can be viewed as a graph.

3.6 Temporal reasoning in TrollCreek/JavaCreek

As mentioned earlier Martha Dørum Jære implemented a temporal framework in JavaCreek in her master thesis [Jær01]. A shorter description of the framework can be found in [JAS02].

The representation of temporal data builds directly on the framework presented by James F. Allen in [All83], and you will notice that the framework developed by Jære has some similarities with Ma and Knight's.

3.6.1 Allen's framework

The basic temporal element in Allen's framework is the interval. An interval has some length, but the length is not specified. In other words: We only know that an interval has some length, but not if it lasted one minute, one day or one year.

Intervals are connected to each other through relationships. There are 13 relations in all, and this is because of the simple reason that there are 13 possible ways an interval may be related to another interval. The resulting structure of intervals and relationships is a graph/network. A simple network is shown in Figure 3.5.

A key point in Allen's theory is that several relationships can be defined between two intervals. This means that we are not certain which of the inter-

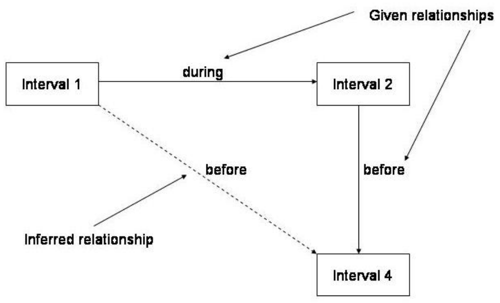


Figure 3.5: A simple example showing Allen's framework

vals that hold. This is referred to as a disjunction of relationships. If we had complete temporal knowledge about the situation an interval would just have one relationship with each of the other intervals. This does not mean that we have to specify a relationship from one interval to all the others manually when we add a new interval to the network. The theory also comes with an inference mechanism that infers possible relationships between intervals by traversing the network.

It will be clear that if all intervals have one or more relationships with the other intervals in the network it will become cluttered after a while. This will also bog down the inference mechanism. To make the network less cluttered Allen proposes the use of what he calls reference intervals. A reference interval groups together clusters of intervals. The cluster of intervals grouped together under the reference interval is related to the other intervals in the network only via the reference interval.

3.6.2 Implementation of framework in TrollCreek

Not everything in Allen's framework was carried over to JavaCreek/TrollCreek. For instance was the possibility of having reference intervals left out, but much of the rest was implemented. This includes the concept of disjunctions of relations between intervals, which poses a problem when it comes to using the framework for reasoning. The representation is actually some steps ahead of the reasoning in that the reasoning mechanism does not handle disjunctions, but the representation still allows them, more on this below.

A temporal case can have findings attached to it, as normal cases, and intervals. It is the presence of intervals that separate the temporal cases from the "normal" cases.

The intervals can be related to each other by using the 13 relations from Allen's framework. In addition to these relationships an interval has findings attached to it. An interval can in some respects be viewed as a sub-case to the overarching temporal case.

3.6.3 Temporal reasoning

The reasoning focuses on predicting unwanted events. To aid in this two special states are defined: the *alert* state and the *alarm* state. A retrieved case triggers an alert if a matching past experience indicates an upcoming unwanted event. An alarm is triggered when a seemingly unavoidable event is about to happen. In practice the states are represented as *markers* at specific intervals in cases in the case-base, and an alarm is sounded if they are found during the case matching.

The actual case matching is done by what is called the dynamic ordering algorithm. This algorithm computes a temporal similarity degree between two cases by "moving" the input case along cases in the case-base, more specifically:

1. Find the first interval in the input case (IC) and the case it is to be compared with (CC). These intervals are called intervalIC and intervalCC respectively.
2. Check intervalIC and intervalCC for matching or explainable findings.
3. If a match is found under the previous step, the temporal path strength is updated.
4. Check getSameTimeIntervals for new information and special situations. If there are special situations then perform some action.
5. Get the next interval from CC and IC using getNextInterval.
6. Unless getNextInterval is empty go to step 2.
7. Return temporal path strength.

Two methods are used by the algorithm. The method *getNextInterval* retrieves the interval that is closest to the current one in the future. The method *getSameTimeIntervals* retrieves the intervals that share time points with the current one.

Apart from returning a temporal similarity degree we see that the algorithm takes care of the alert and alarm states under point 4. Closer inspection of the algorithm reveals that it makes an implicit assumption that we have only one path through the network of intervals. This is not necessarily so. Allen's framework lets us specify that "A *before* B" can be as possible as "B *before* A," within the same temporal network. How such situations should be handled is left unanswered.

A real-world domain was implemented using the framework. The domain was that of predicting unwanted events in oil drilling. More specifically: Predicting a condition known as *stuck pipe*. Consult [JAS02] for an overview of how this implementation was done.

3.6.4 Discussion

Allen's framework allows the expression of very complex situations, but this comes at a price. The price is that the framework becomes harder to use in the reasoning process. There is also the question of whether the type of uncertainty Allen's framework offers is the most useful for most CBR purposes. If we are working with time stamped data, which seems to be the norm for the systems we have surveyed, we are not really interested in being able to tell whether A came before B, or if it was the other way around.

The temporal reasoning work done in JavaCreek stands out from the other systems in that it exclusively uses qualitative data, not quantitative. This allows for expression of uncertainty since we do not need to know when an interval started or how long it lasted. This is something that is a direct consequence of using Allen's framework, since it does not incorporate such data.

3.7 Relations to our framework

What lessons can we draw from these systems when we are now going to develop our own temporal framework?

Most of the systems have a very tight connection to their domain, and it seems that transferring the systems to another domain range from easy to impossible. Our goal is to develop a general framework that will be applicable to a number of domains. The two frameworks that articulate their domain independence most clearly of the ones we surveyed, is the one developed by Ma and Knight and the one by Jaczynski.

Ma and Knight's framework has a feature that it will share with ours: The representation can be visualized as a semantic network. Data is represented in a semantic network in TrollCreek, so we do not have much choice. There is of course the question of how much of the temporal data that should be represented explicitly in the semantic network. Do we want the display all the elements in the model like Ma and Knight does, and Jære for the matter? Or do we want to make some sort of abstraction? I.e. letting some nodes embed complex data objects? We could define time line objects in the code, and then display the time lines as concepts in the knowledge model, while not displaying explicitly all the time points a time line is made up of.

It is not clear how this kind of embedding will fit into the Creek-architecture, and we will not explore this topic here. We will go with the idea of including all the elements in our temporal representation explicitly in the knowledge model.

This means that the representation will have, superficially at least, similarities to the frameworks developed by Ma and Knight and Jære. These two representations are both influenced by Allen's framework for temporal reasoning. Since usage of Allen's framework in TrollCreek thereby already has been investigated, we will try to find a different way of representing temporal data. As a digression we can mention that Schmidt and colleagues have voiced objections against using Allen's framework in multiparametric course analysis [SG01]. They base this sentiment on the discussion by Elpida T. Keravnou in [Ker95]. The further description of how we want our representation to be is in Chapter 5.

Since TrollCreek is a knowledge-intensive system we want the temporal reasoning to make use of relevant domain knowledge is such is present. Do any of the systems in this chapter use domain knowledge to enhance their reasoning? CARMA, WIND-1, Ceaseless-CBR, TempoExpress, ICONS, TecoMed, Jaczynski's framework and Jære's framework all use domain knowledge in some sort or another. However, this domain knowledge is often so firmly connected with the reasoning mechanisms that it will be difficult to replace it with that of another domain. I.e. the model used to adapt cases in CARMA. Of course the general idea of combining model-based and case-based reasoning can be applied across a number of domains. However, we would like the domain knowledge used to be easily replaced.

The way to achieve this is to define a general framework for representing temporal domain knowledge in the knowledge-model, and making the reasoning mechanism use this knowledge. Now some questions pop up: What knowledge can we expect to have, and how can we utilize it? It is from here the discussion in Chapter 6 starts off.

Chapter 4

Dataset selection

In order to be able to test the temporal framework we wanted to find a dataset that could be used as an example domain.

Before we discuss the pros and cons of the datasets we considered, we will answer the question: Why not just use the one employed by Jære in her master thesis? The reason is that a dataset was not really used. The thesis included two cases that were used in an illustration of how the framework and reasoning worked. It seems that the oil companies are not too willing to give out the kind of information that the cases are made up of. We therefore chose at the start not to use that domain.

Not just any dataset will do. We would like the dataset, and the adjoining domain, to meet the following criteria:

1. The dataset needs to include a temporal dimension.
2. The dataset should not just be a time series that follows the evolution of a single parameter.
3. The dataset must be able to be divided into several cases.
4. There should be some domain knowledge available, but a domain where there is complete domain knowledge is not interesting since a case-base would then be superfluous.
5. Due to time constraints the domain should not be too complex.

4.1 Datasets from the Internet

The Internet is a natural place to start looking for datasets. Finding examples of time series consisting of one parameter value is easy. However, we need to build a case-base, so we need several related time series. This reduces the selection, and it is further reduced when we add the requirement that the time series should follow several parameters.

Amongst other things we explored the archives available from the UCI Machine Learning Repository [BM98]. Most of the datasets there did not include temporal data and were intended for use in classification tasks. The archive contained two datasets we deemed worthy of further exploration:

4.1.1 Bach chorales

This dataset is registered under the name *Bach Chorales (time-series) Database.* The dataset contained the soprano line of 100 chorales, in other words a single line of notes. The dataset had previously been used in computer aided generation of music. CBR has been used in making music [Per98], and also in performance of music [GAdM04][AGdM03] [TW03], so it should be possible to come up with a scenario.

One possibility is to create continuations on small sequences of notes based on an existing case-base of musical works and some domain knowledge, for instance which notes that are most suited to be played over a certain chord. This can be viewed as a kind of prediction task.

Music can in general fulfil the first four points on our list of criteria for selection. The problems come when we reach the final point. How to compare two scores/cases? How to predict a continuation? What domain knowledge should be included? These are serious obstacles, especially if you have little prior musical knowledge. So the decision was made that this domain was too complex for our purpose.

As a digression we can mention that Bach turns up in prediction tasks elsewhere too. In a "prediction contest" participants were given different datasets and asked to predict the continuation of them [WG93]. One dataset contained an encoding of an unfinished fugue by Bach. The participants were not told anything about the datasets, so to them it was just collections of numbers. Not to all though, some musically inclined persons recognized the Bach-dataset as a score.

None of the participants used anything resembling case-based reasoning. How could they when they were only given a single dataset from each category and not even told where it came from? The methods employed included neural nets, Markov models, curve fitting and such.

4.1.2 Data from diabetes patients

This dataset bears the name *Diabetes Data.* It was originally prepared for the 1994 AAAI Spring Symposium on Artificial Intelligence in Medicine. It contains data recorded from 70 diabetes patients over a period ranging from weeks to months. The data contains information about the patients' diet, glucose levels, insulin administration, amount of exercise done and other factors that are relevant to the treatment of the illness.

The original idea behind the dataset was to see how AI-methods could aid in the management of patients with diabetes. This included analyzing the data to find critical episodes and generating guidelines for each patient on how to best administer the treatment, which mainly consists of diet considerations, exercise and insulin injections.

Developing a system that warns the patient that it is about time to make some kind of intervention (i.e. insulin injection) based on an analysis of his recent doings and/or comparison with other patient records, could be something that would allow us to use temporal CBR.

If such a system would have any practical value is left unanswered, since we do not have much medical knowledge. This lack of knowledge is a hindrance, and especially in such a complex domain as management of diabetes patients. In the end the decision was made to abandon the domain because of this complexity.

4.2 Local data sources

We also made an effort to get hold of datasets from local sources. Three possibilities were considered.

4.2.1 Progam for helseinformatikk

Medical oriented data from "Program for helseinformatikk" ("Program for health informatics") here at NTNU, which amongst other things is involved in the development of Electronic Patient Journals (EPJ). Part of this work involves the transformation of existing patient data, in whatever form it may be, into a form that can be used by the EPJ. A consequence of this is that they have access to large quantities of medical data.

Clearly the amount of data was not a problem, but the question still remained whether the data could be used to construct a test domain that suited our purposes. A possible scenario that was discussed regarded the use of CBR to predict conflicting medication.

In this scenario a set of prescriptions given a single patient over a period of time would constitute a case. This needs to be supplemented with a doctor's evaluation of some the cases, i.e.: This patient did not experience any adverse effects, but this patient did, and so on. A large case-base where we do not know the outcome of any of the cases is of little use. In addition to this, domain knowledge in the form of already known adverse effects between certain types of drugs could be included.

However, there were also some obstacles involved in using medical data. Much of the data contained sensitive information, and measures to make the data anonymous are needed prior to them being used in research. This process also involves clearing the use of the data with the proper authorities. All this takes time, which is a scarce resource.

In addition to this the scenario might quickly become more complex than is desired, thereby violating our fifth requirement. This led to us not pursuing the use of medical data any further.

4.2.2 Web-logging data

An opportunity to use web-logging data from a search engine that specializes in finding calls for papers for conferences arose while we were searching for a suitable dataset.

The web-logging data contained the movements of the users that visited the site. Included here are the queries the users made. A possible task could be to suggest pages the user might find interesting by comparing the movements of the user so far with movements of other users. A case would then be a sequence of pages visited, with eventual other information that was collected about the user.

This resembles how the Broadway system works [JT99]. However, looking at that work amongst other things made it clear that the task was rather complex. Again the decision was made not to pursue the idea any further.

4.2.3 DNA microarray experiments

Using data from DNA microarray experiments was also considered. The task could be to predict the outcome of an experiment by comparing it with previous experiments. The experiments are carried out in several stages, thereby providing what can be viewed as a series of events. So the input experiment would be an unfinished experiment, and the output a prediction of the remaining stages.

This task of prediction the remaining stages did not seem to have any practical value, and it was unsure if it would be possible to make a good prediction based on earlier experiments. This coupled with the fact the DNA analysis is a complex domain, led us to abandon the thought of using data from DNA microarray experiments.

4.3 Generation of datasets

In the end it was decided that we should make our own datasets. The reason for this is that modelling one of the domains we investigated would require us to spend much time on understanding the intricacies of that domain, and that would take time away from the main goal of this thesis: Developing a general framework for representing and reasoning with temporal data in TrollCreek.

Our test domain should of course resemble a real-world situation, and it should fully utilize the framework we will develop. We will not do a performance analysis on our test domain. The example will be of a proof-of-concept type where we show how temporal data is represented and how the reasoning mechanism functions.

The test domain and example are described in Chapter 7.

Chapter 5

Representation

In this chapter we will describe how the temporal data is represented. The representation does not build, as previously mentioned, on that developed in JavaCreek by Jære. In fact the two representations differ from each other to such a degree that something expressible in one may not be so in the other. We will start off by going into some of the design decisions.

Except for Ma and Knight's framework and the work done earlier in JavaCreek, the systems we surveyed use some sort of sequential time line as the basis for their representation. Time lines are certainly easier to reason with than a net of intervals related to each other. It was a goal to make the representation simpler than that previously implemented in JavaCreek, and this made it natural to adopt the idea of using time lines.

Jaczynski's framework is the only one of the time lines based systems we surveyed that allowed multiple time lines to be stored within a case; one time line for each parameter considered. This ability had some appeal to us, since temporal data from the oil drilling domain is in the form of several concurrent time lines. We are not using this domain here, but nonetheless because of the research done with TrollCreek within oil drilling (see Chapter 3.6.3) it would be nice to have a representation that allowed multiple time lines within a case. This would make it easier to encode data from the oil drilling domain in eventual future use.

Another major difference from the previous representation is the lack of the ability to represent uncertainty. Representing and reasoning with uncertainty is a complex task, so we decided to drop it altogether and instead focus on other aspects of the representation. Hopefully this will not be a big loss. As mentioned in Chapter 3.6.4 it is also uncertain whether the kind of uncertainty offered by Allen's framework is useful for most domains where CBR is applied. There are of course other ways to represent uncertainty that might be more useful, i.e. having fuzzy borders for the elements in the time lines. However, this line of thought has not been pursued. We say that eventual uncertainty has been handled at a meta-level, in other words; before the data was encoded.

We decided to use quantitative temporal data in our representation; this also differs from the previous framework in JavaCreek which uses qualitative data. This use of quantitative data sets it apart from the other systems which uses quantitative data in some form or other. The rationale for using quantitative data is that most data can be time-stamped, and important information can be lost when going from a quantitative to a qualitative representation.

We hope that the restrictions we have put on the data above do not preclude it from being applicable across numerous domains. To sum up we have the following restrictions:

- The data is time-stamped.
- The temporal data can be expressed as events sorted into one or more time lines.
- We have absolute temporal knowledge. This means that we have no uncertainty in the temporal data.

Our representation will use a semantic network as its basis since this is the way knowledge is modelled in TrollCreek.

5.1 Components

The event is the smallest temporal element in our representation. One sequence of related events constitute a *time line*. In turn one or more time lines are attached to a *temporal case*. An example of a time line is shown in Figure 5.1.

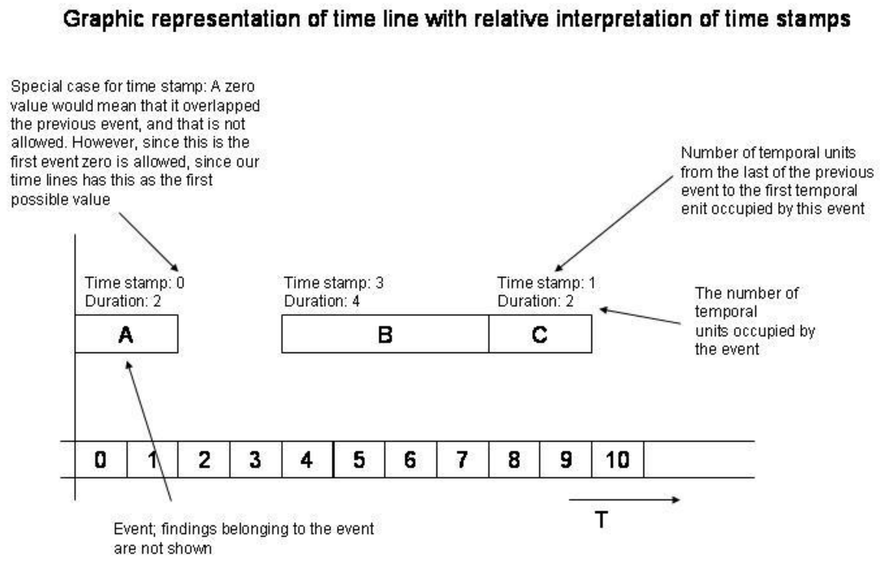

Graphic representation of time line with relative interpretation of time stamps
Special case for time stamp: A zero value would mean that it overlapped the previous event, and that is not allowed. However, since this is the first event zero is allowed, since our time lines has this as the first possible value
Number of temporal units from the last of the previous event to the first temporal enit occupied by this event
Time stamp: 0 Duration: 2
Time stamp: 3 Duration: 4
Time stamp: 1 Duration: 2
The number of temporal units occupied by the event
A
B
C
0 1 2 3 4 5 6 7 8 9 10
T
Event; findings belonging to the event are not shown

Figure 5.1: Example of a time line. Here we use relative time stamps, and it is time stamps of this type we will use in our example too. What a relative time stamp is will be explained in due time.

We will now go into details about the different components of the representation. The description follows a top-down approach; we start at the case level and then move on to the different elements that make up a temporal case.

5.1.1 Temporal Case

A temporal case has non-temporal findings and time lines attached to it. The non-temporal findings are attached to the case node with *has finding*-relationships, just as with regular cases.

These non-temporal findings can be thought of as providing the *context* of the case, see [Jac97]. Non-temporal findings provides a way for the cases to contain information that is not expressed in the time lines.

How a temporal case appears in the knowledge model is shown in Figure 5.2. This particular case has two non-temporal findings and one time line attached to it.

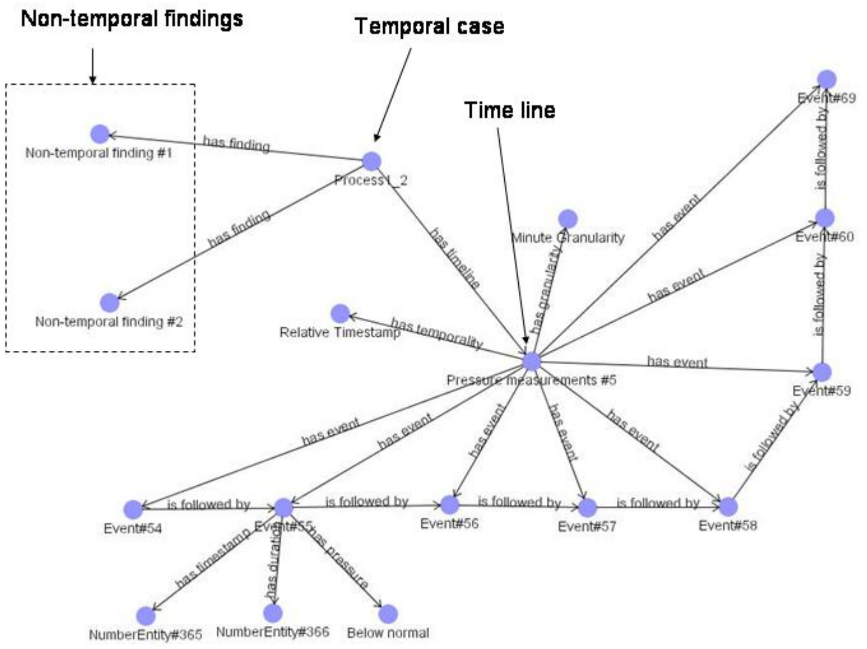


Figure 5.2: Example of a temporal case as it is represented in the knowledge model

A temporal case is assumed to be a subclass of the predefined concept *New Temporal Case* in the knowledge model. The name *New Temporal Case* was chosen to set it apart from the temporal cases in Jære's framework which are subclasses of the concept *Temporal Case*.

5.1.2 Time lines

A time line describes some time extended situation, i.e. the values of a parameter over period of time, a list of events and so on. A time line is made up of a sequence of non-overlapping events.

A time line needs to be a subclass or instance of the concept *TimeLine* in the knowledge model. The time line in Figure 5.2 is named *Pressure measurements#5*. It is not shown in the figure, but this time line is an instance of the concept *Pressure measurements*, which in turn is a subclass of the concept *TimeLine*.

In addition to having events attached to it a time line also has some other relations that show how it is to be interpreted, more specifically: The granularity and temporality.

The granularity defines the temporal unit used in the time line. Possible values are second, minute and day. The granularity is set by adding a *has granularity*-relation to the appropriate granularity concept.

The time line in Figure 5.2 has *Minute*-granularity and uses relative time stamps. We have not explained what a relative time stamps are yet, so we will start making up for this right now. Our representation of time lines rests on two assumptions:

- We know how the events that make up a time line are related to each other. We can for any two events say that one came so and so many time units after another.
- We know how long each of the events lasted.

There are several possible representations that fulfil these two assumptions. For instance could each event have a precise date and duration: Event 1 started at 12.10 on July 10 1992 and lasted for ten minutes. Or it could just be a time that is relevant to some other event: Event 2 started 20 minutes after Event 1 and lasted for 30 minutes.

Seeing that much data is time stamped with a precise date, we wanted to include the ability to represent explicit dates. However, most often we would probably not be interested in the actual dates, just how far apart the date values of two events are. Maybe we in some cases do not have the actual dates. Not that this is a big problem since we could just invent some dates. We would just have to remember to space the dates so that the intervals between the events is correct.

The best solution would be if we could choose how we wanted the time stamps to be interpreted. In the case where we have the actual dates we would use them as our representation, and in the case where we do not have dates we could for instance explicitly specify the space between two consecutive events in our representation.

This led us to allow for multiple representations in our framework. There is an abstraction layer between how the data is represented in the knowledge model and the form it is when it used by the reasoning mechanisms that makes this possible. The reasoning mechanisms operate on the *TimeLineInterface*-interface (see Appendix A.2.1), not on the actual knowledge model.

The temporality value determines how the timestamps of the events connected to the time lines are interpreted. It is specified by a *has temporality* relation from the time line to the concept that represents the desired interpretation. We have defined two possible interpretations:

- Absolute. This means that timestamps are to interpreted as Java *Date*-objects. The term “absolute” refers to the fact that the date of an event can be determined by just looking at its timestamp.
- Relative. The events are connected to each other in this case. The timestamp of the event is interpreted as a long value defining how long after

the end of the previous event this event started. What the long number means depends on the granularity of the time lines. I.e. if the granularity is *Minute* the number is interpreted as the number of minutes after the previous event. The events also need to have a follows relation to another event so that we know what event the long value refers to. It this type of interpretation we will be using in this thesis. Figure 5.1 gives some more information on how the interpretation functions.

And here comes our disclaimer: We have only implemented the *Relative*-nterpretation (the easier one) in practice. The *Absolute*-interpretation needs to use the granularity value to know what temporal level the focus is on (minute, day, year and so on), and we also need a definition of a zero point. When using relative time stamps we assume that the first event of a time line starts at point zero if nothing else is specified. We also assume that all the time lines under a case use the same temporal space. This means that all the time lines under the case start at the same zero point. For the *Absolute*-interpretation the zero point could be an explicit time point, i.e. January first 1972. It could also be the time of the event with the earliest date of all the events that are collected under the temporal case.

Some groundwork for the *Absolute*-interpretation has been done. All time stamps regardless of the interpretation are encapsulated *Long*-objects. A Java *Date*-object can be instantiated with a long value, and this is what we have planned to do: Make Date-objects out of all the time stamps, sort the events in chronological order, check for overlap between the events, which is not allowed, and then anchor the time line to a zero point.

The nice thing about the absolute time stamp interpretation is that it allows us to enter data from a database directly into the system without having to change the time format.

5.1.3 Event

An event has findings attached to it. These findings describe the event. In addition to findings the event has some duration, which is represented as an integer value. The minimum value is one. If the time line the event belongs has minute granularity the minimum duration would be one minute, if it is days, it would be one day and so on.

An event also has the time stamp value that we discussed in the previous section that is used to relate it temporally to the other events in the time line. We are focusing on the *Relative*-interpretation, so the time stamp is interpreted as the number of temporal units after the end of the previous interval until the start of this. Because we are using this interpretation we also need to specify a *follows*-relationship between the event and the one that it follows. A graphic description of an event is shown in Figure 5.3.

The event in the figure has a time stamp value that is represented by the node *NumberEntity#577*, and a duration represented by the node *NumberEntity#578*. This is the way numbers are represented in TrollCreek. These two nodes/concepts each encapsulate a subclass of the native Java *Number*-class, which in turn encapsulate a primitive number. If we want to know the numerical values of these nodes, we have to look at their frames.

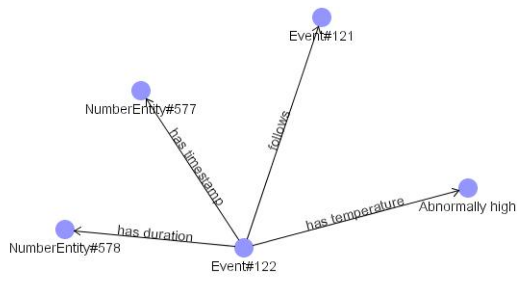


Figure 5.3: Example of an event

As mentioned are time stamps represented by *Long*-objects, while durations are represented by *Integer*-objects. The rationale behind this is given in Appendix A.2.1.

We can also mention that the system will allow the duration to be set to zero, but this can produce unwanted results, since we have not defined a meaning for an event with zero duration. Ma and Knight allow, as the only system that does this, their time elements to have zero duration. It is not clear what is achieved by this however.

5.2 Ontology

In Figure 5.4 the elements used by our representation are displayed as they appear in a knowledge model. Since the TrollCreek top level ontology has not been finally established yet, we have kept our placement of the concepts simple.

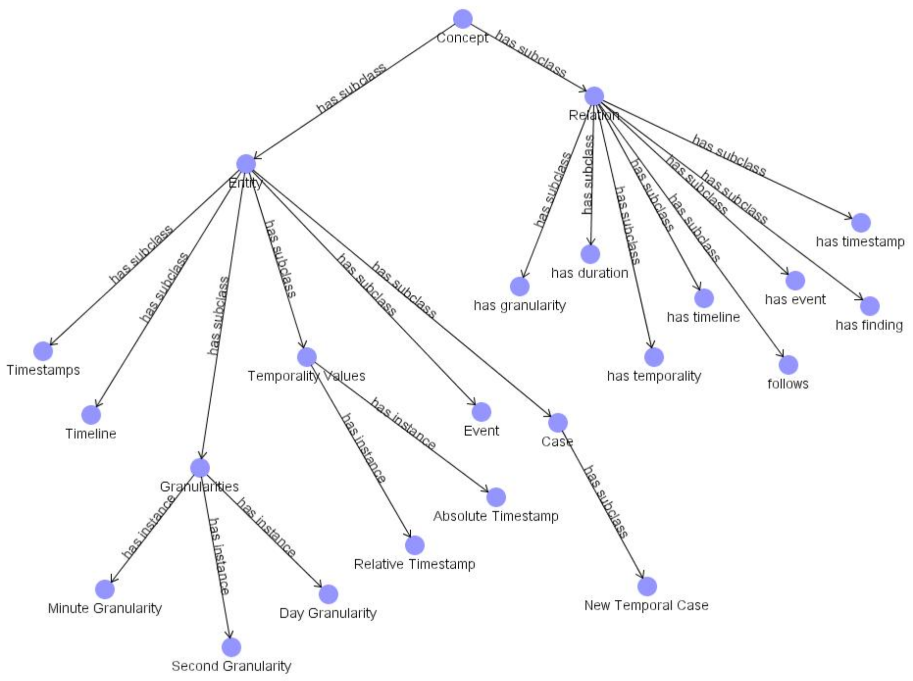


Figure 5.4: The elements used by our representation (inverse of relations not shown)

Chapter 6

Reasoning

In this chapter we will describe how the reasoning is done. We have split the reasoning task into two major parts: Non-temporal reasoning and temporal reasoning. The non-temporal part uses existing reasoning methods in TrollCreek, and is used as a "test" to check if the temporal reasoning should be initiated. The temporal reasoning mechanism is the core this thesis and will receive most of our focus.

6.1 Non-temporal reasoning

The non-temporal findings of two cases are first compared using the standard case-matching algorithm in TrollCreek. If the similarity measure of this comparison is above a given threshold a temporal similarity measure is also computed. The threshold value is set in the code (see Appendix A.2.2).

The rationale behind this is that it is not necessary to embark on a computation of temporal similarity if the cases do not share some other characteristics. This relationship between standard similarity measures and temporal similarity measures was present in Jære's framework too. This is also similar to *filtering on context* from Jaczynski's framework.

6.2 Temporal reasoning

We will focus on the *Retrieve* step of the CBR-cycle, so our goal is; given an input case find the most similar cases in the case-base. To do this we will develop a method that computes a similarity score between two temporal cases. Our framework decomposes the temporal reasoning task into three subtasks:

- Abstraction task: Compresses/abstracts all the time lines of a case into a single time line.
- Time line comparison task: Takes as input two time lines and computes a similarity score.
- Event comparison task: Takes as input two events and computes a similarity score. This task does not involve any use of temporal data.

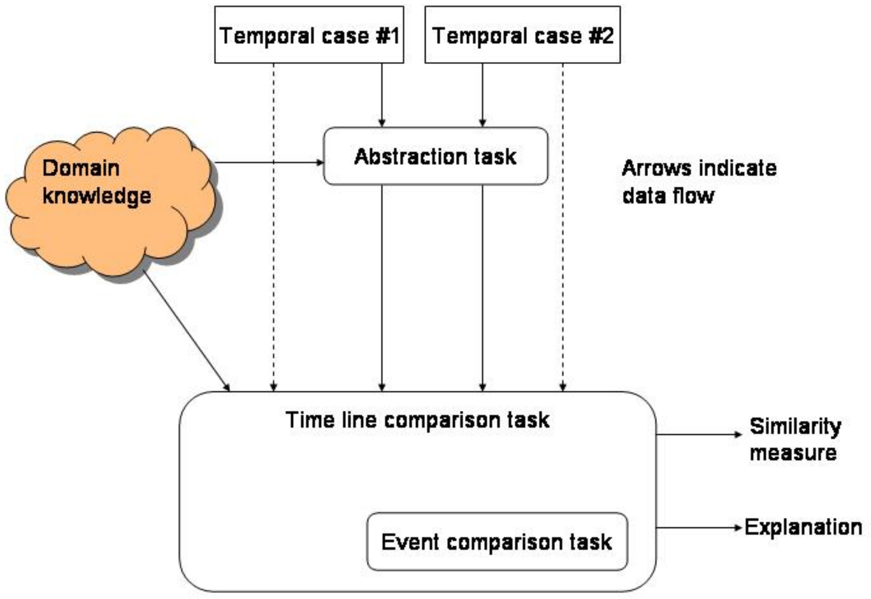


Figure 6.1: Subtasks of temporal reasoning

How the tasks are related is seen in Figure 6.1.

These are tasks, we do not say anything about the methods used for implementing them. We will in this thesis implement a method for solving each task.

From Figure 6.1 it can be seen that the event comparison task is embedded in the time line comparison task. We decided that the event comparison task should be represented as a separate task even though it is embedded in another task because it is quite elaborate.

The dotted lines from the orgin and target case to the time line comparison task point to the possibility of omitting the abstraction task if the cases only contain one time line each, we will discuss this below.

6.3 Abstraction task

When we made the choice that a case could contain several time lines, we added some complexity to the reasoning task. Instead of having to compare two time lines, we need a scheme that gives a similarity measure between two cases each consisting of an arbitrary number of time lines.

Some of the systems we surveyed have had to deal with this problem. Jaczynski's *elementary behaviours* can involve several time lines. Ma and Knight's proposed graph similarity methods for comparing two cases. ICONS abstracted several parameters into a single state.

Another solution is to do nothing special when a case contains several time lines. That is; compare pairs of time lines that monitor the same parameter.

The problem here is that the overall picture is lost. The interplay between the different parameters is not taken into consideration. If this is what is wanted it is actually possible to do so by using the methods described in this chapter by doing some tricks in the knowledge model. This comes as consequence of TrollCreek's structured comparator architecture. We will however assume that this is not what is wanted. We want our temporal reasoning mechanism to consider the interaction between different parameters.

We made the decision to use the idea from ICONS to handle the issue with multiple time lines: Abstracting them into a single time line. The approach used in ICONS has appeal since it can incorporate domain knowledge, and it provides flexibility in that we are free to say how the actual abstraction should be done.

6.3.1 Temporal abstraction

The question now is: How do we go from multiple time lines to one? We will from here on refer to the concept that does this many-to-one transformation as an *abstractor*.

There are several ways to abstract multiple time lines. One solution is to just add/compress all the time lines together as shown in Figure 6.2. Actually, the term abstractor is something of a misnomer when used here. Nothing is abstracted away. However, this is the way the abstraction task has been implemented in this thesis. This implementation shows that the basic idea functions, but it is not an optimal solution.

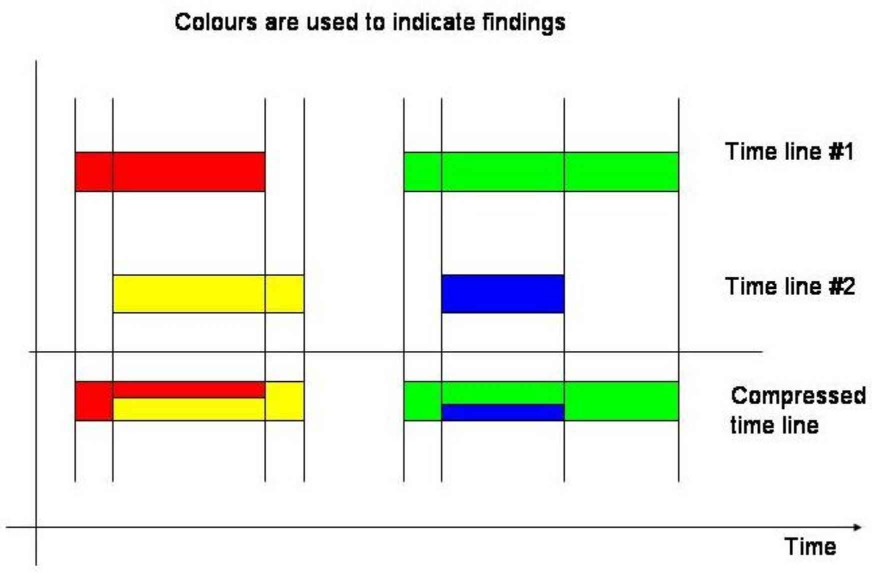


Figure 6.2: Compression of time lines

We will in the following describe how we would like the abstraction task to be implemented. The best way to do the abstraction is clearly different from domain to domain, and it is also knowledge-intensive. We would like to keep our tasks as domain independent as possible, so what is needed is an implementation of the abstraction task that is able to utilize domain knowledge provided by the knowledge engineer. In other words: The abstraction task should rest on a knowledge-based domain independent framework for temporal abstraction. Such a framework has actually been developed by Yuval Shahar [Sha97]. The framework has been implemented in the RÉSUMÉ system.

While it is too much to insist that the methods that implement our abstraction task should have the same power as the RÉSUMÉ system, it is still useful to examine it and the underlying framework, since many of the ideas present there are of potential use to us. The reason why this description comes here, and not in the chapter on relevant research, is that RÉSUMÉ is not a CBR-system. We will keep our focus on the "idea level " and not go into any practical details of the system.

Shahar also operates with the term "temporal-abstraction task," and he gives the following informal definition of the task (quoting from [Sha97]):

> The **temporal-abstraction (TA)** task can be viewed informally as a type of a generic *interpretation task*: Given a set of time-stamped data that interpret past and present states and trends and that are relevant for the given set of goals.

The RÉSUMÉ system creates many abstractions for a domain, and will also be able to answer queries related to the abstractions. Our task just has one goal: Create a single time line that captures the most important points of the time lines in a temporal case.

Shahar decomposes the temporal abstraction task into five subtasks:

1. Temporal-context restriction: This creates a frame of reference for the abstraction task. The context may for instance be therapy of insulin-dependent diabetes. What abstractions are done depends on the context. For the diabetes context it might be creation of intervals that show the insulin level.

2. Vertical temporal inference: Creates abstractions by using inference on parameter values that occur at the same time.

3. Horizontal temporal inference: Creates parameter intervals by inference on parameter propositions of the same parameter.

4. Temporal interpolation: Creates parameter intervals by joining disjoint parameter points or parameter intervals.

5. Temporal-pattern matching: Creates abstraction intervals by matching of patterns.

These tasks are domain independent. The methods that perform the tasks need the following kind of domain knowledge to function:

1. Structural knowledge. I.e. *ABSTRACTED-FROM* relationships.

2. Classification knowledge. I.e. definition of a parameter range as LOW.

3. Temporal-semantic knowledge. I.e. definitions of what intervals that are concatenable. Two consecutive periods of anemia can be summarized as one interval, while two consecuitive pregnancies cannot be summarized as an 18 month pregnancy.

4. Temporal-dynamic knowledge. I.e. the persistence of parameter values.

This kind of knowledge needs to entered by a domain expert, and after that has been done the system is ready [SM99].

In addition to the knowledge types the system also contains a temporal-abstraction ontology that defines what intervals, patterns, events and other terms actually mean. It should be clear from all this that the framework is large, and we will keep our discussion in the following section on a shallow level. What we will do is to sketch how the ideas behind some of the tasks in Shahar's framework can be used in our abstraction task.

Vertical temporal inference

In Shahar's framework the mechanism that performs this task accepts one or more parameter points or intervals as input and returns an abstraction point or interval. The mechanism that performs the task is called the *contemporaneous-abstraction mechanism*, and is used for two subtasks:

The first subtask is classification of parameters. This assumes the presence of *classification knowledge*. This can be value ranges (i.e. LOW) and definitions of parameter values that are to be placed in that range (i.e. *100mm-250mm*).

In Jære's thesis this kind of temporal abstraction was actually used when the cases from the oil domain was encoded. The abstraction was done manually prior to the modelling of the cases in the knowledge model however. This is something that could be automated, thereby omitting the need for a manual pruning of the parameter measurements before they are entered into the system. Our example that will be presented in Chapter 7 assumes that such classifications of parameter values have been done manually.

The best way to do this kind of classification may be to have a layered abstractor architecture. That is: Having abstractors defined at time line levels in addition to the one defined at the case level (see Figure 6.3).

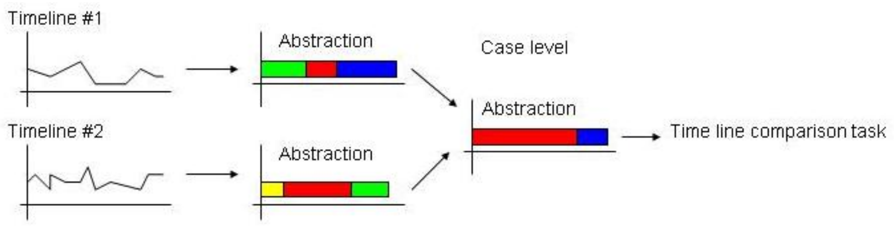


Figure 6.3: Example of a layered abstractor architecture

However, in our implementation we only work with a single abstractor at the case level. If we were to implement a layered abstractor architecture we would need two types of abstractors: One for the case level and one for the time line level since these two levels are different. The abstractor at the case level would have several time lines as input, while the one at the time line level would have a set of events as input. The output would however be same for the two types: A time line.

The second subtask is called *computational transformation*, and it maps values of one or more parameters into the value of another, abstract parameter. An illustration is given in Figure 6.4.

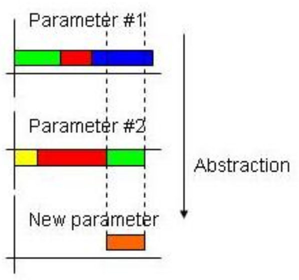


Figure 6.4: Example of computational transformation

As an aside: If we define functions that take as input all possible combinations the values of the measured parameters and that give as output a single state describing the overall situation at that point, we would automatically solve the problem of going from many time lines to one.

Horizontal temporal inference and temporal interpolation

We have grouped these two tasks together since they both operate on parameter values in the horizontal direction. Horizontal temporal inference creates abstractions where two intervals that measure a parameter value meet, see Figure 6.5.

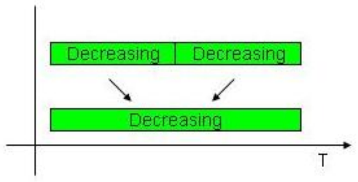


Figure 6.5: Abstraction where two intervals meet

Temporal interpolation on the other hand create abstractions where the intervals do not meet, see Figure 6.6.

This could be used to contract time lines by abstracting several events into one that covers the same interval. An example is shown in Figure 6.7.

Why would we want to contract time lines in this manner? If the method that performs the time line comparison task does not scale well we need to keep the input to it below certain thresholds. It is here this kind of contraction of time lines becomes useful.

As an example we can consider the oil domain which Jære used in her thesis. The raw data a case was based on consisted of about 2500 measurements of four

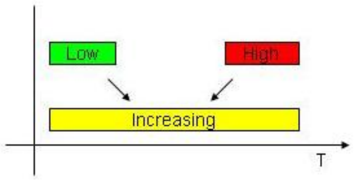


Figure 6.6: Abstraction between two disjoint intervals

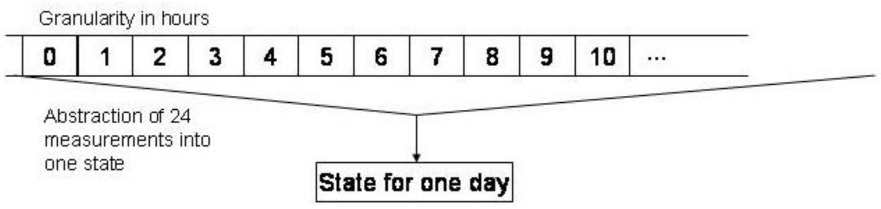


Figure 6.7: Abstraction of a sequence of events into a single event

parameters. If each measurement was represented as a separate event in the time line that is input to comparison task, the method that implemented it could not do any complex calculations, since the task would then have become intractable.

Temporal-pattern matching

This task creates abstractions of a more complex type than the other tasks. An example could be (again quoting from [Sha97]):

> ... an episode of drug toxicity from a strate of LOW(White bloodcell) count *lasting more than* 2 weeks and starting *within* 0 to 4 weeks of a state of LOW(Hemoglobin) *lasting more than* 3 weeks, in a patient who is receiving certain drugs.

This specification of temporal patterns has similarities with the *elementary behaviours* in Jaczynski's framework. We can quickly outline how the Jaczynski's idea of elementary behaviours might be implemented as temporal patterns: If we view a single case as a context we could define an abstractor for each case that would extract what we perceive as the important points. The same abstractor could then be applied to an unsolved case to see if it contained the same important points as the case the abstractor. This is illustrated in Figure 6.8.

This way of doing things would reduce the time line comparison task to just moving the abstractor from the target case to the origin case. Usage of abstractors in this way would require us to know what the important points in a case are.

In fact, this pattern matching task can be seen as a kind of rule based reasoning task. We could label some of the abstracted states as critical, and sound a warning when such a state was discovered. This is somewhat similar to the *alert* and *alarm* states that Jære uses. A difference is that she used the

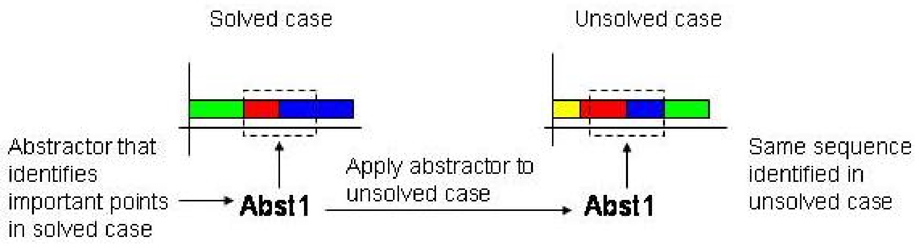

Solved case
Unsolved case
Abstractor that identifies important points in solved case
Abst1
Apply abstractor to unsolved case
Abst1
Same sequence identified in unsolved case

Figure 6.8: Each case has its own abstractor which is applied to an unsolved case

states in the case matching, while we propose to use such states prior to the actual case matching.

Omitting the abstraction task

Finally we will point out that not all scenarios involve multiple time lines. Sometimes there is just one time line attached to a case, i.e. a single sequence of notes, as in TempoExpress (see Chapter 3.2). In this case the designer of the system has a choice between omitting the abstraction task, or having an abstractor perform some action on the single time line. For instance: In the newest version of TempoExpress the single note lines go through an abstraction (the I/R model) before they are compared with each other.

Summary of temporal abstraction

If we were to prioritize the ideas laid out above, we would suggest that classification of parameter values and concatenation of events be investigated first. These tasks seem to be necessary to have in place if we are dealing with long time lines to avoid overburdening the time line comparison task.

The temporal-pattern matching task requires that methods that perform the other tasks are in place, since it uses output from these as its input.

We have not said anything about how the ideas can be implemented in TrollCreek in practice. The big picture is that domain knowledge will be entered into the knowledge model, possible in the form of the knowledge types Shahar uses, and then the abstractors will be use this knowledge when processing a case. We will leave the subject at this level.

6.3.2 Implementation of the abstraction task

As mentioned previously the method that performs the abstraction task does not do this in accordance with the ideas we just have presented. We have provided a single abstractor: *TimeLineCompressor*. *TimeLineCompressor* does not actually abstract anything, it just compresses the time lines it finds under a case. This is of course not knowledge-intensive in any way. The process was illustrated in Figure 6.2. The beginning or end of an event in any of the time lines in a case will trigger the creation of a new event in the abstracted time line. An event in the compressed time line will contain all the findings of the events on the other time lines that it shares time points with.

A temporal case can be assigned an abstractor via a *has abstractor*-relation. An abstractor encapsulates a Java class, and it is this class' responsibility to transform the time lines of the temporal case into a single time line. In the knowledge model an abstractor needs to be a subclass of the concept *Abstrator*. The time line generated by the abstractor is used by the comparator in the case matching.

6.4 Time line comparison task

Here two time lines are compared according to some scheme. The task can take as input time lines that are output by the abstraction task, or it can work directly on two non-abstracted time lines. Its output is a similarity score, and it should in addition be possible to get an explanation of the comparison.

How should the time lines be compared? From the survey of related research we did it is clear that there is no one method that is applicable under all circumstances. However, one idea was present in all systems: The order of the elements that are to be compared, events in our case, should have an effect on the similarity score.

We decided to investigate the use sequence comparison techniques closer. Such methods are usually based on dynamic programming methods [CLRS01][Smi91]. Two of the CBR-systems we surveyed used sequence comparison methods in their case comparisons: Ceaseless CBR and TempoExpress. We will now give an overview of what sequence comparison is.

6.4.1 Sequence comparison

Sequence comparison, as we use the term here, deals with the problem of comparing sequences where the correspondence between of the elements that make up them is not known in advance. Such methods are more complex than ones that compare sequences of equal length and/or only compare corresponding elements. Before we move on with the description of sequence comparison, we can mention that there of course exist scenarios where such simpler comparisons are the ones to use. In one of the systems we surveyed, WIND-1 (see page 10), it was a key point that only corresponding elements in two cases should be compared, and all the cases had the same length. A simplified version of the case comparison used in the WIND-1 system is shown in Figure 6.9.

In this case an algorithm that starts at T and moves backward while computing the similarity scores would be used. There is no problem involved in implementing such an algorithm in our framework, since it fits nicely within our task definition. When we have decided to investigate sequence comparison methods closer, it is partly because such methods are more complex than the one described above, and if our framework can support them it would be a plus. Also the problem sequence comparison addresses; not knowing the correspondence between elements of two sequences, is one that will arise in many domains.

The first full-scale work on sequence comparison was the book *Time Warps, String Edits, And Macromolecules: The Theory And Practice Of Sequence Comparison*, edited by David Sankoff and Joseph Kruskal [DS83]. The first edition

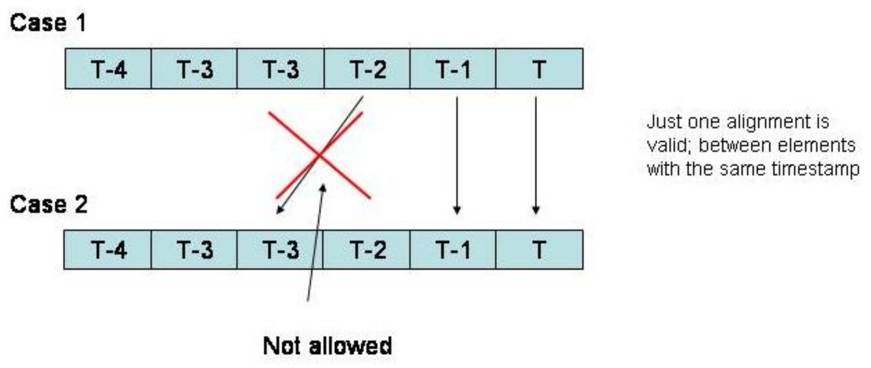

Case 1
T-4
T-3
T-3
T-2
T-1
T
Case 2
T-4
T-3
T-3
T-2
T-1
T
Just one alignment is valid; between elements with the same timestamp
Not allowed

Figure 6.9: Only corresponding elements are compared

came in 1983, so the book is getting on a bit in years. A newer overview of sequence comparison methods can be found in Dan Gusfield's book *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* [Gus97]. Gusfield's is more tuned to the use of sequence comparison within computational biology, while Sankoff and Kruskal's include papers that show the use of sequence comparison in such diverse areas as: Computational biology, speech recognition, study of bird songs and more.

A key point in sequence comparison is that sequences that in essence have similarities, differ from each other on the surface in a number of ways. These differences need to be specified, and the most used types are:

- Substitutions: Elements from the sequences may have been substituted.
- Deletions: Elements may have been deleted from the sequences.
- Insertions: Elements may have been inserted into the sequences.

Other types of differences may be defined, but we will focus on these. As an example Ceaseless CBR uses these three operators, while TempoExpress has defined some others too. The output of sequence comparison methods is information about the distance between the two sequences and how they differ. In our use we can view the distance as a measure of the strength of the comparison, and the analysis of the differences as the explanation. The differences between two sequences can be represented in at least three different ways: traces, alignments (also called matching) and listings. These types are illustrated in Figure 6.10. What mode of analysis that is best depends on the task at hand.

A trace can be viewed as a collection of arrows/lines going from one sequence to the other. The arrows that appear in the full trace are the ones that give the optimal score. An important point in a trace is that an arrow cannot cross another arrow. If this was allowed we would disregard the linear structure of the sequences. We would just have two unordered sets where there could be an arrow between any two elements in the two sets without regard to other arrows.

A matching is similar to a trace. Elements from the two sequences are aligned. However, some elements may not be aligned with any elements of the

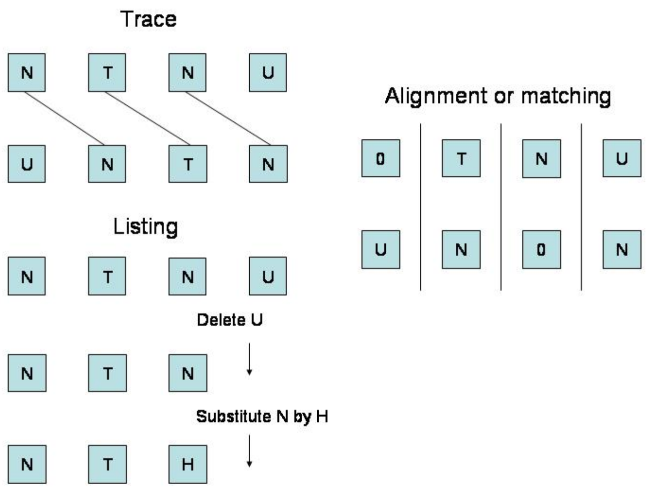


Figure 6.10: Different modes of analysis

other sequence, and these are then aligned with a special symbol (we have used 0 in Figure 6.10). A matching provides more information than a trace since it shows the size and placement of gaps, those parts that are aligned with a 0 in the other sequence. The higher information content is also shown by the fact the several different matchings may be transformed into the same trace. Such a transformation is done by placing an arrow between two aligned elements, and removing the 0's, so that some elements are possibly left "hanging," like the U's at the ends in our trace example in Figure 6.10.

A listing gives the sequence of operations that transform one sequence into the other while giving the optimal overall cost.

As mentioned sequence comparison methods are implemented using dynamic programming methods, and we will now examine this in more detail. The basic edit distance algorithm is given in Equation 6.1.

$$d(a^i, b^i) = \min \begin{cases} d(a^{i-1}, b^j) & +w(a_i, \phi) & \text{deletion of } a_i \\ d(a^{i-1}, b^{j-1}) & +w(a_i, b_j) & \text{substitution of } a_i \text{ by } b_j \\ d(a^i, b^{j-1}) & +w(\phi, b_j) & \text{insertion of } b_j \end{cases} \tag{6.1}$$

$d(a^i, b^i)$ is the edit distance function applied to two sequences a and b, with lengths i and j respectively. The $w(x, y)$-function gives the cost of the operation it represents. We see that substitutions, deletions and insertion are used in this case. What exactly do the operations mean? A substitution is a replacement of element a^i of sequence a with element b^j of sequence b. A deletion is the deletion of element a^i, and an insertion is the insertion of element b^j at point a^i in sequence a.

Besides selecting operators and weights, the matching can be manipulated by putting constraints on what is an acceptable analysis. This can for instance be limits on the number of consecutive deletions and insertions.

The actual computations are done using a matrix that is filled in a row major order. The value of a cell depends on the values of the cells above, to the left and above to the left (see Figure 6.11). The table can be viewed as stored function calls.

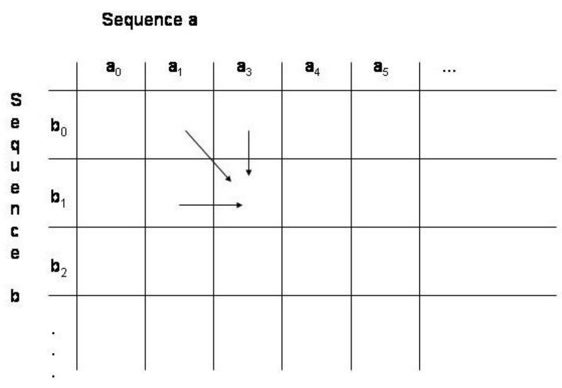


Figure 6.11: Matrix used in dynamic programming

The "final" edit distance value is stored in the cell with coordinates a_i, b_j. That is to say the lower right corner of the matrix.

To find the sequence of operations that produced the optimal edit distance value, the matrix in Figure 6.11 can be analyzed in a reverse fashion: Starting in cell a_i, b_i and for each cell deciding what operator was used, then backtracking to the upper left corner. Another solution is to store the operations used in a matrix of their own, as seen in Figure 6.12.

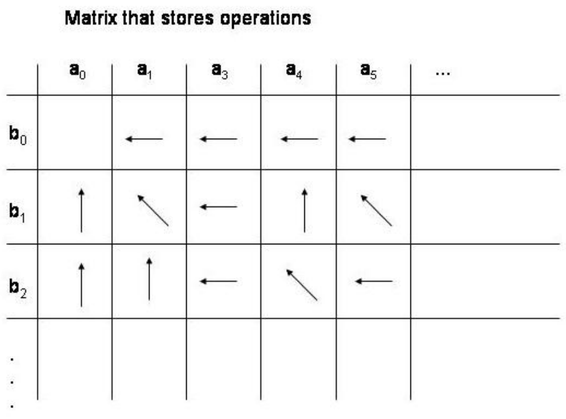


Figure 6.12: Operations matrix

Here it is just a matter of following the arrows from one end of the matrix to the other. Note that there can be several optimal edit distances, each having the same value. This is not a point we will concern us with here.

6.4.2 Implementation of the time line comparison task

This task takes care of the actual comparison between two cases, and the methods that perform it are therefore implemented as comparators (see Chapter 2.2.2). It is the temporal comparator that guides the whole matching process, and this includes activating the abstractor.

The comparator fetches the case's abstractor by following the *has abstractor*-relation attached to the case. If such a relation is missing it uses *TimeLineCompressor* as a default. Since we have provided *TimeLineCompressor* as the only abstractor, it does not in practice make a difference if our cases specify *TimeLineCompressor* as their abstractor or not.

As we have said there is not one single method of comparing two time lines that is right for all purposes. However, we have tried to make an architecture that will allow for easy implementation of different methods.

Behind the scenes the comparators that implement this task are assumed to be subclasses of the *TempMatchGeneric*-class (see Appendix A.2.2), which in turn is a subclass of the *EntityComparison*-class. The *TempMatchGeneric*-class is declared abstract, and its purpose is to enforce what the different matching methods needs to be able to do. The other elements in TrollCreek that represent the results of the comparison task work on *TempMatchGeneric*-objects to avoid having a GUI-element for each different method of comparing two time lines. It is instead the matching methods that must conform to what is given in the *TempMatchGeneric*-class.

We have implemented two different time line comparison methods in this thesis to show what sequence comparison can do. The first one is implemented in the comparator *TempMatchingScheme1*, and is an example of a trace. The second method is implemented in *TempMatchingScheme2* and is an example of a listing. *TempMatchingScheme2* uses the basic edit distance formulation given in Equation 6.1.

TempMatchingScheme1

This comparator finds the optimal trace between two time lines based on the following recurrence equation:

$$Alignment(TLO_i, TLT_j) = \max \begin{cases} Alignment(TLO_{i-1}, TLT_j) \\ Alignemnt(TLO_i, TLT_{j-1}) \\ NewTrace(TLO_{i-1}, TLT_{j-1}) \end{cases} \tag{6.2}$$

TLO (TimeLineOrigin) and TLT (TimeLineTarget) are time lines, and the subscript refers to the event at that position in the time line. I.e. TLO_i refers to event i in TLO. $NewTrace(TLO_i, TLT_j)$ is defined as follows:

$$\begin{aligned} NewTrace(TLO_i, TLT_j) &= max_{a<i,b<j}(\frac{1}{dist(i,j,a,b)} + NewTrace(a,b) \\ &+ sim(TLO_i, TLT_j) + sim(TLO_a, TLT_b)) \end{aligned} \tag{6.3}$$

The function $dist(i, j, a, b)$ is defined as:

$$dist(i, j, a, b) = (i - a) + (j - b) \tag{6.4}$$

We could use a more elaborate distance function that actually used the temporal distance between the events, but to keep it simple this was not done. We just count the number of events between two arrows. This has the consequence that the starting points and durations are not explicitly used by this scoring scheme, and the same will be true for the other matching scheme we present. $sim(a, b)$ computes a similarity value between two events, and uses the method described in the next section.

Notice that we are here looking for the max score, while the edit distance looks for the minimum score. We have an upper bound on the maximum score in this case. If TLO_i and TLT_j are identical the score produced by trace between them will be the maximum score possible.

What the scoring scheme does is to place arrows between events in the two sequences by considering the similarity of the events and the distance between the arrows. How much weight that should placed on similarity and how much should be placed on distance is dependent on the domain. A weight constant, or variable, could be attached to the $dist(i, j, a, b)$ and $sim(TLO_i, TLT_j)$ functions to gain control over how much weight is given to each measure. This comparison ensures that if two sequences share similar sections they will receive a higher score than two sequences that also share similar events, but where they are spread out differently in the two sequences. It is kind of similarity versus distance scoring metric. A graphic illustration of the matching scheme is given in Figure 6.13.

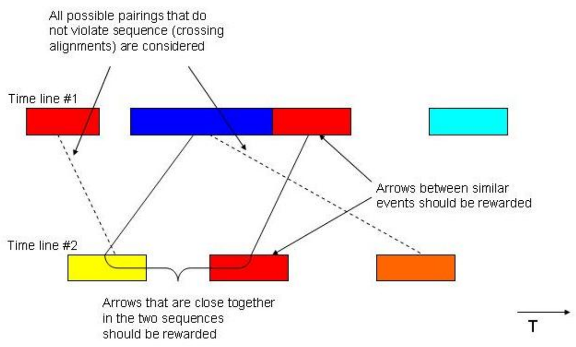

All possible pairings that do not violate sequence (crossing alignments) are considered
Time line #1
Arrows between similar events should be rewarded
Time line #2
Arrows that are close together in the two sequences should be rewarded
T

Figure 6.13: Illustration of matching scheme: Similarity vs. distance

If you examine the figure you might discover a problem with the $NewTrace(TLO_i, TLT_j)$ function. When we place the first arrow we do not have a previous arrow to calculate the distance to. In this case the function just returns the similarity measure for the two events and leaves out the distance.

The algorithm used in this section is quite different from the basic edit distance outlined in Equation 6.1, but it is nevertheless closely related to it conceptually. That was also the point of including this matching scheme here: To show the there is substantial flexibility in sequence comparison methods.

TempMatchingScheme2

This comparison method uses the basic edit distance equation shown in Equation 6.1. The weight of the substitution operation is calculated using the event comparison method described in the next section. If two events are similar, the substitution cost should be low. This makes intuitive sense. The substitution cost is given by:

$$w(TLT_i, TLO_j) = 1 - sim(TLT_i, TLO_j) \tag{6.5}$$

The $sim(a, b)$-function is the same as the one used by *TempMatchingScheme1* and will be described in the next section. Two identical events will get a similarity score of 1, which means that the substitution cost will be 0. What cost should be given to the deletion of an event in the origin sequence (the unsolved case)? It should cost more to delete an important event, but since the case is unsolved we do not necessarily know which events are important. We have provided the following cost:

$$w(TLO_i, \phi) = NumberOfFindings(TLO_i) * W_{fd} + Duration(TLO_j) * W_d \tag{6.6}$$

Where W_{fd} and W_d are weights. The insertion cost is calculated in similar manner:

$$w(\phi, TLO_j) = NumberOfFindings(TLO_j) * W_{fi} + Duration(TLO_j) * W_d \tag{6.7}$$

Where W_{fi} and W_d are weights. The rationale behind these equations is that it should cost more to delete or insert events that have a large number of findings than events with few findings. It should also cost more to delete or insert events with long durations.

The weights used in the operations are domain dependent, and finding suitable values may be something of an art. The creators of TempoExpress used genetic programming in finding suitable weight values. The weighting of the different operators is very involved in Ceaseless CBR, and the weights also change continually, but this is a consequence of the fact that the input is an ever-evolving sequence of alerts.

Of course the costs can be calculated in other ways, and probably should, than the ones demonstrated here. One way to add flexibility to the cost calculations is to involve the explanation strengths of the *has event*-relations. We have not defined a semantic for them, but we could for instance say that the explanation strength in one direction is a value that is used in the deletion operation, and the explanation strength in the other direction is used in the insertion operation.

It is also worth noticing that in this scoring scheme a low distance between two sequences mean that they are similar. We are in other words searching for the lowest score, while we in *TempMatchingScheme1* were searching for the highest score. If the distance is zero the two time lines are identical. A disclaimer is in place here: No, they do not need to be identical to get a distance value of zero since we do not include the durations in our substitution operation. This means that two events with the same findings but with different durations will match perfectly.

6.5 Event comparison task

It is very likely that a method that compares two time lines will need some sort of function that measures the similarity between two events. The two matching schemes we have implemented in this thesis use the same function for this task, and it is this we will describe in this section.

The event-to-event similarity assessment is done using a special comparator called *AbstractComparison*, which is very similar to the *CaseComparison*-comparator that is used in standard case matching (see Chapter 2.2.1).

The type of matching done by *CaseComparison* seems to fit the event matching task nicely. An event is described by a number of *has finding*-relations just as a case is, it can therefore be viewed as a sort of sub-case. Whether *AbstractComparison* should use spreading activation along the same relations as *CaseComparison* does, is something we will not dwell upon here. In the implementation the same set of relations is used. We will however remark that there might be some issues involved in including causal relations in the spreading activation, since these relations imply that something occurs before something else. As an example: *A* causes *B*, implies that A occurred before B, and it is not clear how this should be interpreted if A is part of one time line and B of another.

The main practical problem that precludes us from just using *CaseComparison* directly on the events is that the time line output by the abstraction task does not correspond to a time line that is explicitly represented in the knowledge model. What this means is that neither the time line nor the events it is made up of are represented explicitly as entities in the knowledge model. *CaseComparison* works on entities and therefore cannot be used here. It can however be used if we omit the abstraction task, as we can do if we are dealing with a domain where the temporal data is specified on a single time line. With the abstraction task omitted the reasoning mechanisms works directly on the time lines as they are represented explicitly in the knowledge model.

AbstractComparison is a class that works on a set of *has finding*-relations instead of on two entities. This works in our case since our abstractor just makes new events out different groupings of existing relations. It would probably not work as well in the more general case where the abstractor makes new events with findings that where not present in the existing time lines.

The problem is solvable however. We could for instance make an abstractor that actually makes a new time line in the knowledge model. If we did this we could dispense with the *AbstractComparison*-class and use the *CaseComparison*-class, since we then would have the abstracted time line represented in the knowledge model.

6.6 Summary

Now that we have described all the tasks we can take an overview of the ideas. Two temporal cases are input to the reasoning mechanism. Both target and origin cases go through the abstraction task. The abstraction method used can be identical for both cases, or it can differ. The point is that the abstractor is to use whatever domain knowledge we have to extract what is important from the cases. The output from the abstraction task is a time line which in theory is of the same form as the time lines that are attached to cases explicitly in the knowledge model.

The time line comparison task then takes over and compares the time lines. The scoring scheme it uses needs to be adjusted from domain to domain. We have given two examples that use sequence comparison methods. These methods also make use of an event comparison task. The method that performs the event comparison task is knowledge-intensive and is built on the existing method of comparing cases in TrollCreek. Since the time line comparison methods we have implemented incorporate the event comparison method, the comparisons become knowledge-intensive through the operators they use.

If we look at our comparison methods at the level above the operators, they basically find a sequence of operations that applied to the time lines give an optimal score. This is not knowledge-intensive. Whether the comparison task could be enhanced by using knowledge is something we will leave as an open question.

After the comparison is complete we are presented with a value that gives a measure of the similarity and an explanation. The explanation is in our case made up of the abstracted time lines and the sequence of operations applied to the time lines by the comparison method.

Chapter 7

Example

This chapter contains an example of the framework in use. As mentioned in Chapter 4 we decided not to use any of the datasets we examined, but instead went with an imaginary domain. We will now describe what this domain is and how it is modelled, and then we will show an example of temporal case matching.

The example is meant to illustrate the practical representation of temporal data in TrollCreek and how the reasoning works. It is not meant to be a solid simulation of a real world domain.

7.1 The domain

The domain is that of a simple industrial process. A few parameters are monitored, and if something wrong happens in the process the values of these parameters a certain period back are stored as a case. The idea is that an unsolved case is a sample of the most recent measurements process, and it is matched against the case-base to see whether the current situation resembles any of the stored faulty situations.

The process is monitored through three parameters: temperature, pressure and energy consumption. Measurements of these values over a period of time are represented as cases. We assume that some abstraction has already been done on our temporal data, more specifically: That the values of the different parameters have been abstracted into one of the following ranges: *Normal range*, *Above normal*, *Below normal*, *Abnormally high* and *Abnormally low.*

This means that we have to specify these ranges in our knowledge model somehow. How this has been done in our case is shown in Figure 7.1.

We have to specify what time lines we are going to use in the knowledge model. This is done by creating a concept for each the different types of time lines and making them instances of the *TimeLine*-concept. This is shown in Figure 7.2.

In addition to this we have to specify some subrelations to the *has finding*-relation, more specifically: *has pressure*, *has temperature* and *has energy consumption.* The situation is shown in Figure 7.3.

Why do we need to specify these relations? The reason lies in the behaviour of our abstractor, which as mentioned is not really an abstractor at all. The ab-

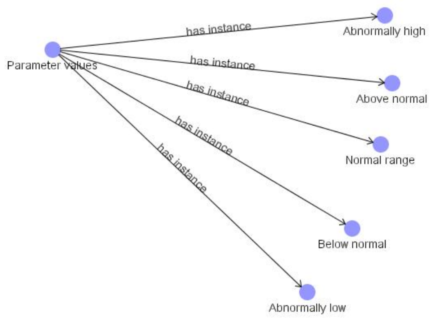


Figure 7.1: Possible parameter values

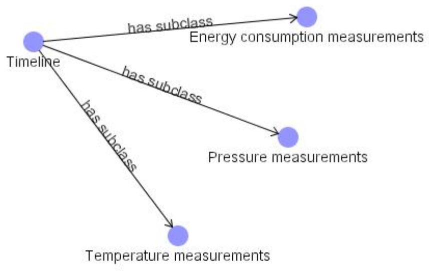


Figure 7.2: Different time lines within our domain

stractor just adds time lines together, so if we just used the standard *has finding* to specify the findings under each event we would get a collection of findings under each event, but we would not know which time line they came from. This would lead to a perfect match between a *Below normal* value for the temperature parameter and a *Below normal* value for the pressure parameter, something which is not what is intended. What we would like is that temperature values should be matched against temperature values, pressure against pressure and so on. To make this happen we define subclasses to the *has finding*-relation. A finding relation will only be matched against relations of its own kind, or relations of which it is a subclass.

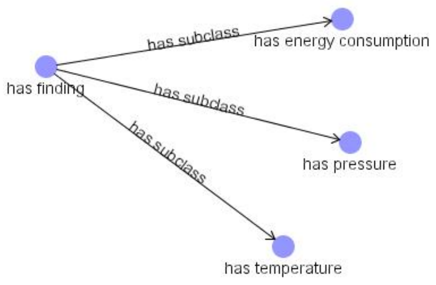


Figure 7.3: Subclasses of *has finding*

This also means that creating different types of time lines for our parameter does not have any practical consequences. We could just specify that our cases contained three time lines of the same type, and identify the parameters they monitored by their type of *has finding*-relations. However, this would be bad knowledge modelling.

7.2 Example of a case

In Figure 7.4 we see the how the temporal case *Process1_1* is represented in the knowledge model. The findings of the events are not shown to avoid the view from being any more cluttered than it already is.

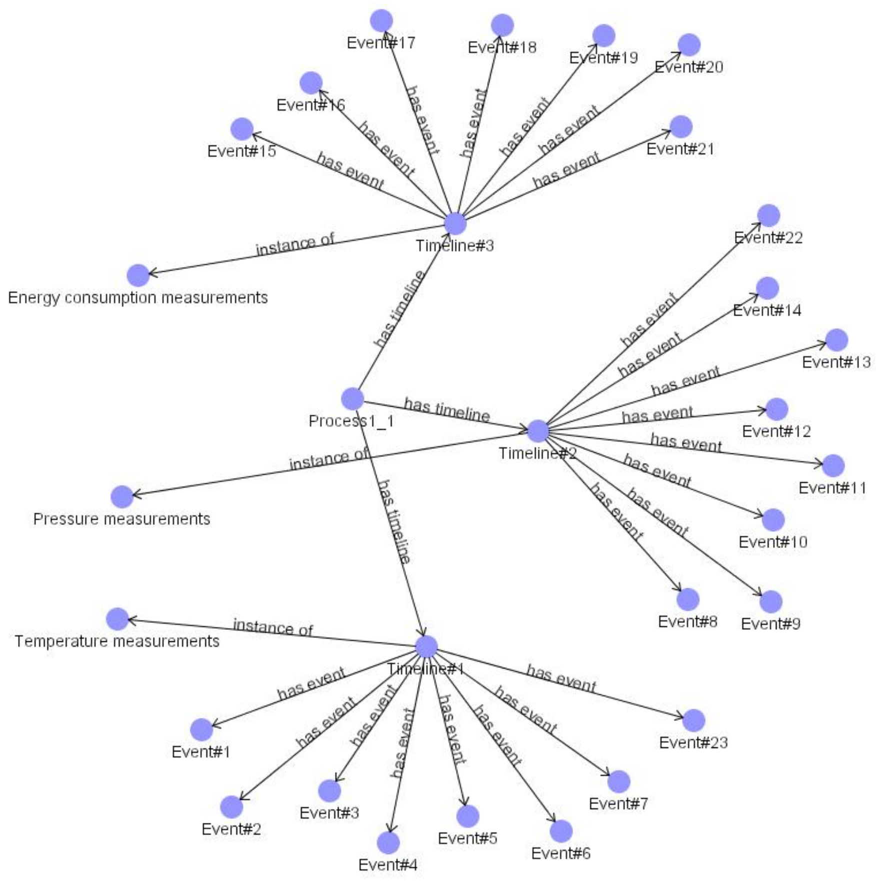


Figure 7.4: Example of time lines within a case

A less frustrating way of viewing a case is through the time line viewer. How *Process1_1* appears there is shown in Figure 7.5.

The three time lines explicitly represented in the knowledge and the abstracted time line (courtesy of *TimeLineCompressor*) is seen in the viewer. The colours used in the viewer are random, and do not serve any other function than

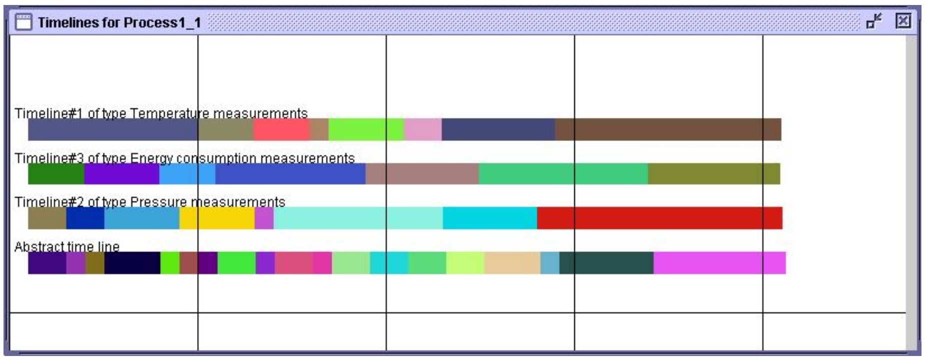

Timelines for Process1_1
Timeline#1 of type Temperature measurements
Timeline#3 of type Energy consumption measurements
Timeline#2 of type Pressure measurements
Abstract time line

Figure 7.5: Process1_1 seen through the viewer

to allow us to tell where an event begins and where it ends. We can also mention that there are ten temporal units between two vertical bars in the viewer.

The viewer does not show the findings of the events. If we want to see the findings we have to search in the knowledge model, more precisely: First we need to see where the event is placed in the time line in the viewer, and then find this event by locating the node that represents it in the knowledge model. The events of the abstracted time line do not exist explicitly in the knowledge model, so we will not find them there. When we view the time lines of case a summary of the abstracted time line is written to the standard output, and it is this we will have to use to figure out which findings go with which event in that case.

In our example every event of the abstracted time line will normally have three findings; one for each the three parameter values. This is just as expected since the abstractor just adds the three time lines together, and all the events of the time lines have just one finding describing the state of the parameter.

7.3 Case-generator

To be able to create cases quickly a case-generator was built. It creates cases according to some simple rules:

1. All cases contain the three time lines mentioned above.
2. An event immediately follows another. This is achieved by setting the time stamp value to 1 for all events. We assume that the data streams continuously from the process, so that there will be no gaps in the time lines.
3. All the time lines within a case are of equal length.
4. An event cannot have the parameter value that the event it follows has. There is no point in creating a new event if there is no change in the parameter value. It would of course be different if we had the restriction that all the events should have equal length (i.e. ten minutes), but this is not the case here.

With the exception of the constraint under point 4, the parameter values are chosen randomly.

The case generator also has a method for copying cases. This was meant to aid in the testing by first copying a case and then altering it a little to see how this would affect the matching score. As the generator is set up now, it makes a copy of each case it creates and stores this in the knowledge model with the original case. Changes to the copy must be done manually at this stage.

The cases do not contain any non-temporal findings, which mean that the normal case matching method that is invoked before the temporal matching will deem all the cases as having nothing in common. It will return a comparison strength of zero in other words. We still want the temporal matching to go ahead, so we have set the threshold value for starting the temporal matching to zero in the code.

7.4 Case matching

Two new elements have been added to the popup menu that appears when the user right-clicks on a node in the TrollCreek knowledge model editor. The elements are shown in Figure 7.6. *View timeline* becomes active when the user clicks on a node representing either a temporal case or a time line, and brings up a viewer, which we already have mentioned, with the time lines displayed.

The other new element, *Temporal matching*, becomes active when the user clicks on a temporal case. This command initiates the temporal case matching.

Now we will examine some cases matches. We will first specify *TempMatchingScheme1* (see Chapter 6.4.2) as the comparator for all the temporal cases, we will return to *TempMatchingScheme2* later.

In Figure 7.7 we have run a temporal matching on the case *Process1_1*. We are currently viewing the matching with the highest score, which is between *Process1_1* and *Copy of Process1_11*. In this case the origin and target case are identical, so we should expect a high score. In fact, since the cases are identical this is highest score we can get.

The matching screen contains the following information. We see the two time lines that are compared and the trace between them. In this case each event is matched to the one directly opposite it on the other time line. The component under the time lines shows the recurrence tables (it is actually just the output from the comparators *toString()*-method). The output from *TempMatchingScheme1* lists three tables. Two of these are recurrence tables, while the third shows the cached comparison strengths between different events (calculated by *AbstractComparison*).

These tables are as good an explanation we are going to get from the scoring scheme. From them we can see all the steps in the calculations. The content of the tables becomes useful when one is tweaking the scoring scheme so that it will perform as intended.

In Figure 7.7 we see part of one of the recurrence tables, more specifically the ones that stores the optimal scores. To the lower right we see the optimal value for the whole sequence: 46.

The text box at the bottom contains a summary of the two abstracted time lines. For each event its time stamp value, duration and findings are listed, in

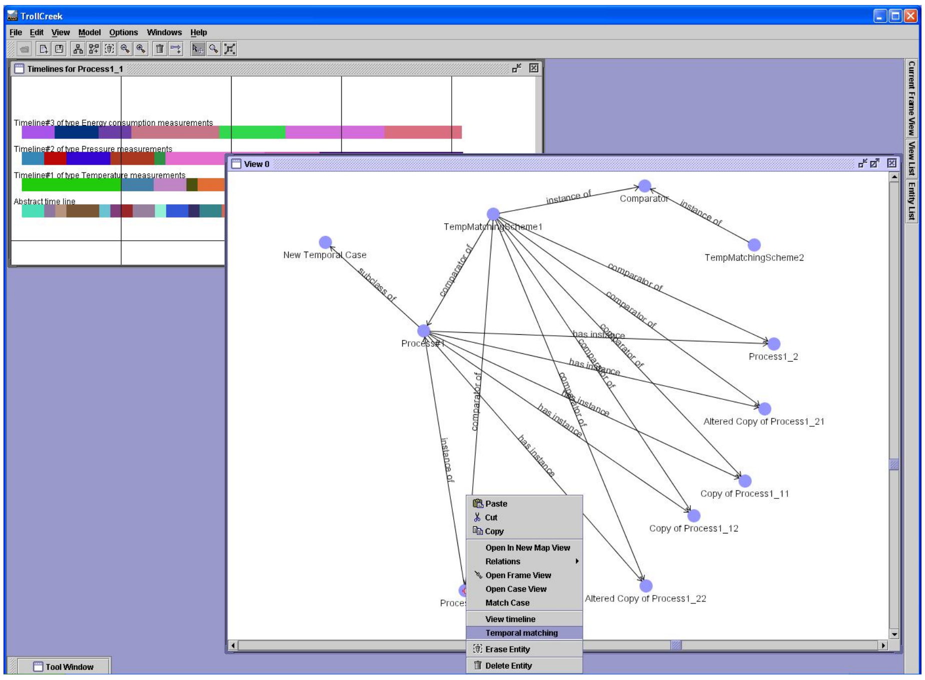


Figure 7.6: Temporal elements on the popup menu: *View timeline* and *Temporal matching*. Behind the popup-menu we see a view that shows the temporal cases we have defined in the current knowledge model. It can also be seen that we have specified *TempMatchingScheme1* as the comparator of all the cases

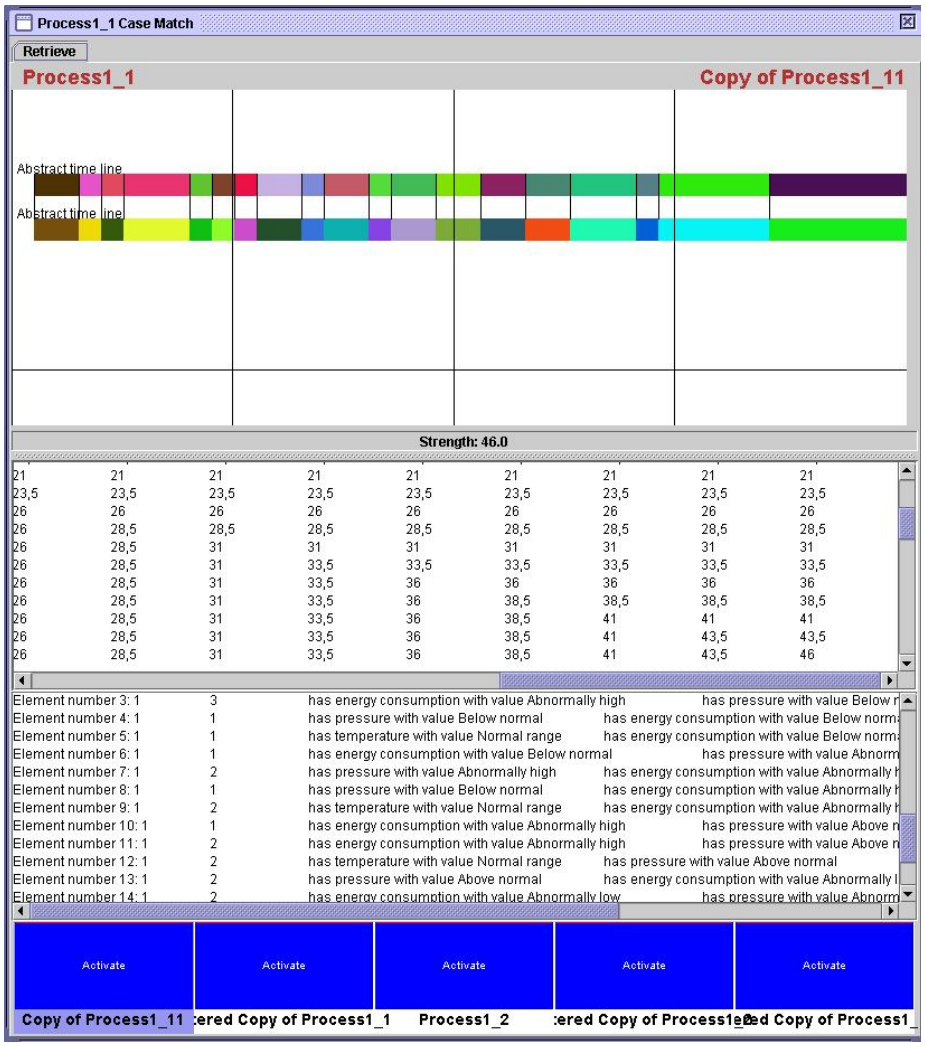
that order. Since the abstract time lines do not exist explicitly in the knowledge model this is the only way we can get information about the abstract time lines.

In Figure 7.8 we see part of the matching results between *Process1_1* and *Process1_2*. This is not a perfect match. We see that the score is lower and the trace is skewed.

We saw that identical cases get the highest score, as we would expect/hope. It is however harder to get a grip on how the matching score changes as the cases become more and more dissimilar. Of course we know which factors influence the score: Length and location of events, and the findings connected to each event. It is the interplay between the factors it is difficult to figure out. The explanation given by the system regarding similarity score just tells us that this is optimal value given the scoring scheme. It is up to us to adjust the weights and operators of the scoring scheme so that it will produce good results.

We will now move on to *TempMatchingScheme2* (see Chapter 6.4.2). We have now set *TempMatchingScheme2* as the comparator for all the cases, and again we have run a matching on *Process1_1*. The same GUI-elements are used so on the surface the comparison looks quite like the one we just saw.

Notice that a trace is not displayed in this case, for the simple reason that we do not have one. Again the two identical cases get the highest score. Also notice that the scores have been inverted and normalized in this case. An edit distance of 0 will give a strength of 1. The columns that indicate the strength of the comparisons at the bottom of the screen are shorter here than was the



```
21      21      21      21      21      21      21      21      21
23,5    23,5    23,5    23,5    23,5    23,5    23,5    23,5    23,5
26      26      26      26      26      26      26      26      26
26      28,5    28,5    28,5    28,5    28,5    28,5    28,5    28,5
26      28,5    31      31      31      31      31      31      31
26      28,5    31      33,5    33,5    33,5    33,5    33,5    33,5
26      28,5    31      33,5    36      36      36      36      36
26      28,5    31      33,5    36      38,5    38,5    38,5    38,5
26      28,5    31      33,5    36      38,5    41      41      41
26      28,5    31      33,5    36      38,5    41      43,5    43,5
26      28,5    31      33,5    36      38,5    41      43,5    46
```

```
Element number 3: 1     3    has energy consumption with value Abnormally high         has pressure with value Below r
Element number 4: 1     1    has pressure with value Below normal         has energy consumption with value Below norm
Element number 5: 1     1    has temperature with value Normal range      has energy consumption with value Below norm
Element number 6: 1     1    has energy consumption with value Below normal            has pressure with value Abnorm
Element number 7: 1     2    has pressure with value Abnormally high      has energy consumption with value Abnormally h
Element number 8: 1     1    has pressure with value Below normal         has energy consumption with value Abnormally h
Element number 9: 1     2    has temperature with value Normal range      has energy consumption with value Abnormally h
Element number 10: 1    1    has energy consumption with value Abnormally high         has pressure with value Above n
Element number 11: 1    2    has energy consumption with value Abnormally high         has pressure with value Above n
Element number 12: 1    2    has temperature with value Normal range      has pressure with value Above normal
Element number 13: 1    2    has pressure with value Above normal         has energy consumption with value Abnormally l
Element number 14: 1    2    has energy consumption with value Abnormally low          has pressure with value Abnorm
```

Figure 7.7: Matching two identical cases

case for *TempMatchingScheme1*. This is not because *TempMatchingScheme1* thinks the cases are more similar than *TempMatchingScheme2*, but because its scores are not normalized, and the mechanism in TrollCreek that generates the columns expects a number between 0 and 1 as input. Any value greater than 1 will result in a column of max height.

In the text box under the time lines we see the operators needed to transform the origin sequence into the target sequence plus the cost of the operators. Since the two sequences are identical the transformation is simple enough: Each element of the origin sequence is substituted with its corresponding element in the target sequence at zero cost. *TempMatchingScheme2* also uses tables, but we have not listed them in the text box.

In Figure 7.10 we see another case matching involving *Process1_1*. This time the cases are not identical, and this reflected in the cost and choice of the

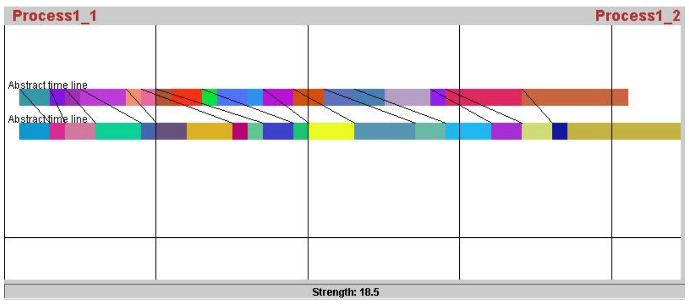
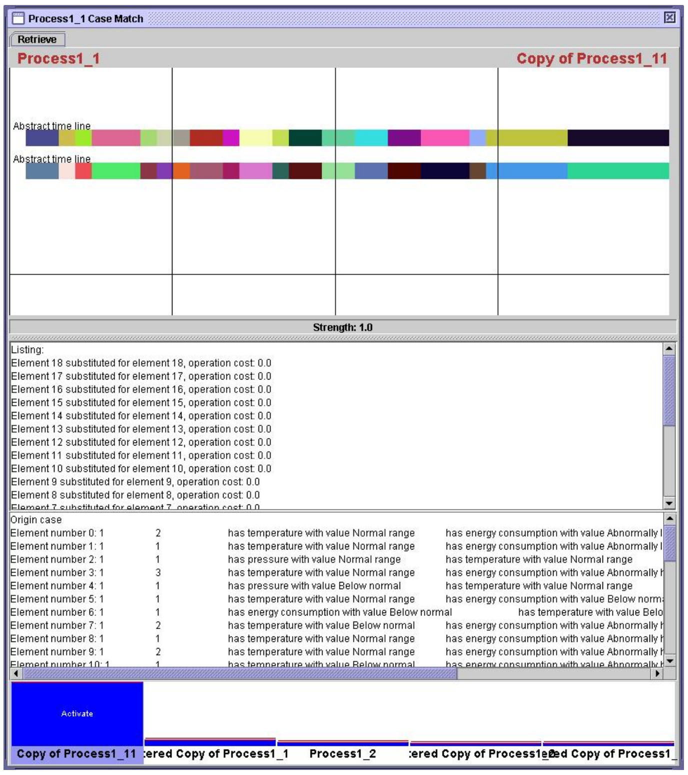


Figure 7.8: Another example of a trace between two time lines

operations.

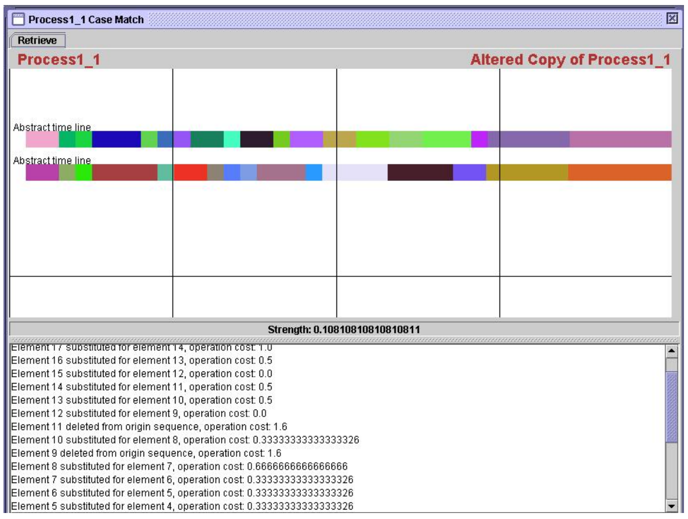
The two abstracted time lines do not contain the same number of events, so some deletions and/or insertions are needed to transform one sequence into another. However, we see that most operations are substitutions. This is a sign that the substitution operation generally is cheaper than deletions and insertions. If this is what is want everything is OK. If we feel that this is not OK we can adjust the weights associated to the operations. At the extreme other hand of the current setup we have the scenario where the cost of deletions and insertions are very small compared to the substitution cost. This may lead to a situation where the cheapest transformation of one sequence into another consists of deleting all of the origin sequence and then inserting the whole target sequence.



```
Listing:
Element 18 substituted for element 18, operation cost: 0.0
Element 17 substituted for element 17, operation cost: 0.0
Element 16 substituted for element 16, operation cost: 0.0
Element 15 substituted for element 15, operation cost: 0.0
Element 14 substituted for element 14, operation cost: 0.0
Element 13 substituted for element 13, operation cost: 0.0
Element 12 substituted for element 12, operation cost: 0.0
Element 11 substituted for element 11, operation cost: 0.0
Element 10 substituted for element 10, operation cost: 0.0
Element 9 substituted for element 9, operation cost: 0.0
Element 8 substituted for element 8, operation cost: 0.0
Element 7 substituted for element 7, operation cost: 0.0
```

```
Origin case
Element number 0: 1      2      has temperature with value Normal range      has energy consumption with value Abnormally l
Element number 1: 1      1      has temperature with value Normal range      has energy consumption with value Abnormally l
Element number 2: 1      1      has pressure with value Normal range         has temperature with value Normal range
Element number 3: 1      3      has temperature with value Normal range      has energy consumption with value Abnormally h
Element number 4: 1      1      has pressure with value Below normal         has temperature with value Normal range
Element number 5: 1      1      has temperature with value Normal range      has energy consumption with value Below norm
Element number 6: 1      1      has energy consumption with value Below normal          has temperature with value Belo
Element number 7: 1      2      has temperature with value Below normal      has energy consumption with value Abnormally h
Element number 8: 1      1      has temperature with value Normal range      has energy consumption with value Abnormally h
Element number 9: 1      2      has temperature with value Normal range      has energy consumption with value Abnormally h
Element number 10: 1     1      has temperature with value Below normal      has energy consumption with value Abnormally h
```

Figure 7.9: Temporal case matching using *TempMatchingScheme2*



```
Element 17 substituted for element 14, operation cost: 1.0
Element 16 substituted for element 13, operation cost: 0.5
Element 15 substituted for element 12, operation cost: 0.0
Element 14 substituted for element 11, operation cost: 0.5
Element 13 substituted for element 10, operation cost: 0.5
Element 12 substituted for element 9, operation cost: 0.0
Element 11 deleted from origin sequence, operation cost: 1.6
Element 10 substituted for element 8, operation cost: 0.33333333333333326
Element 9 deleted from origin sequence, operation cost: 1.6
Element 8 substituted for element 7, operation cost: 0.6666666666666666
Element 7 substituted for element 6, operation cost: 0.33333333333333326
Element 6 substituted for element 5, operation cost: 0.33333333333333326
Element 5 substituted for element 4, operation cost: 0.33333333333333326
```

Figure 7.10: *TempMatchingScheme2* used on two cases that are not identical

Chapter 8

Discussion

8.1 Conclusions

We have developed a framework for temporal reasoning in TrollCreek. The framework contains a way of representing temporal data and a reasoning mechanism.

The representation is based on quantitative data. We have implemented one way of specifying the quantitative data, using what we call *relative time stamps*. The representation also opens for other ways of specifying the data, i.e. actual dates and time points (January 1st 1995 12.00 AM) if that is desired.

The temporal reasoning mechanism consists of two major tasks: The abstraction task and the time line comparison task. The time line comparison task also encompasses an event comparison task.

We have implemented a simple method for the abstraction task, and have outlined how this task can solved better by a knowledge intensive abstraction method. For the comparison task we have implemented two scoring schemes based on sequence comparison methods. The event comparison task is performed by using a method that builds on the existing method for comparing cases in TrollCreek.

We have also provided an example of the framework in use. The example used an imaginary domain and is meant to show that our framework actually functions, not that it functions well. Considerable effort may be needed to develop, or tune the existing, methods so that they will produce good results for a real-world domain.

8.2 Theoretical and practical issues

In this section we will review some issues that have arisen in the development process.

8.2.1 The abstraction task

Our most basic assumption is that we will be able to abstract all of the time lines within a case into a single time line. The assumption actually goes a little further than that, we also imply that the abstraction will pick out the important points that should be used in the comparison task.

Domain knowledge will be of use in finding the important points as described in Chapter 6.3. How the practical implementation of such a temporal abstraction framework should be done was left unsaid, we just sketched some ideas of possible functionality. A lot can be done in this area, so much that questions of where to put the effort and how much effort should be put in it become relevant.

If new methods for performing the abstraction task are going to be developed it might be best to start with an actual domain and work out what methods of abstraction one thinks will produce the best results, and then try to put this functionality into TrollCreek (preferably in a domain-independent manner).

8.2.2 Structure of the case-base

We have not said anything about how we imagine the case-base is structured, but this is an important point that will affect what methods that are best suited to solve the tasks in our framework.

Do all the cases in the case-base have fixed length, or are they of varying length? Do we know the correspondence between the events of the origin and target time line or not? Do the cases contain irrelevant information/noise? Our framework can be applied in all these instances, so it is up the user how he wants to model the domain. We can in general say that the more variables we can pin down the better. I.e. if we know the correspondence between events of the origin and target time lines then we do not need sequence comparison methods, but can use a less complex method.

8.2.3 Scalability

The cases in our example were quite small. What would happen if our abstracted time lines had 1000 events instead of about 20? Our sequence comparison methods would probably fare poorly if we lengthened the time lines in our example and did nothing to speed up the comparison methods. This does not however mean that sequence comparison methods scale so badly that it will be impossible to use them when the time lines are of a certain length. The amount of computation needed depends on many factors. What takes time is performing the operations we use in our scoring scheme (i.e. substitution, deletion and insertion) over and over again at each stage of the comparison. The method we have used for performing the event comparison task, which is used in both scoring schemes, is quite costly. But there is some good news too: TrollCreek caches comparisons of entities. If we keep down the number of different concepts used in the time line that is input to the sequence comparison task, all the comparison may be cached. This means that the event comparison task might be reduced to mainly a table-lookup operation after some event comparisons have been done in the beginning of the time line comparison.

Another method to speed up the event comparison task is to remove the explain step and just use the activate step. This means that we only look for direct matches between findings in two events.

A point is that the methods that perform the different tasks should be adjusted to function well together, and this includes the amount of computation. If the method that performs the abstraction task produces long time lines, the method that solves the comparison task may need to be less complex. If the event comparison task is computationally expensive, the methods that solve the abstraction and comparison task should try to keep the number of comparisons low, and so on.

8.3 Further work

We have presented some ideas and shown that they function; the next step is to use the temporal framework on a real domain to see how well they work. We have already mentioned several times that the methods that perform abstraction and comparison task needs to be tuned to the domain at hand. The methods for performing the abstraction have potential to be knowledge-intensive, and it would be natural to investigate this closer in conjunction with applying the framework to a real-world domain.

In addition to uncovering issues that we may have overlooked when developing the framework, application to a real world domain will also put us in position to do a performance test of the framework.

Better ways of viewing the time lines also need to be developed. The simple viewer used in this thesis only shows where events start and end. Additional features could include:

- Letting the user set the time scale. This also means that the viewer needs to support scrolling.
- Displaying the findings of an event by clicking on it. Now the user has to search through what is written to the standard output to determine which findings hold for a particular event.
- Using the colours more constructively. The colours used by the viewer are random and serve no other purpose to allow us to tell the events apart. This could be enhanced in numerous ways. We could have the colour red associated with critical situations, green when everything is OK and so on. A prerequisite for this is that we have some kind of framework for associating colours with events, which we at present do not have.
- It is frustrating for a human to manipulate the temporal cases through views in TrollCreek. Time lines can contain many events, and a temporal case can in turn contain several time lines. It would be easier if the time line viewer also allowed the user to manipulate the time lines instead of just viewing them. A human will find the representation used by the viewer easier to use than the one in the knowledge model.

Appendix A

Code overview

This appendix gives an overview of how the different parts of the framework has been implemented. For more details than is given here, consult the JavaDoc.

Care has been taken to alter as little of the original code as possible.

A.1 Changes to existing code

This section documents the changes that have been done to the existing code.

New menu elements Two new actions have been added to the popup-menu: *View timelines* and *Temporal matching.* This implies some additions to the following files:

- MoveTool
- ZoomTool
- BuildTool
- Creek.properties

Updating numbers in frame view Some changes were done to be able to change the value of NumberEntities in the TrollCreek knowledge editor. These were done to be able to change time stamps and durations quickly, but are not to be seen as part of the temporal framework.

Comparison panes The following lines were added to comparisonpanes.properties:

```
jcreek.reasoning.TempMatchingScheme1 = jcreek.gui.reasoning.TemporalComparisonPane
jcreek.reasoning.TempMatchingScheme2 = jcreek.gui.reasoning.TemporalComparisonPane
```

Framework in new models When a new model is created the concepts needed by the temporal framework are automatically added to the empty model. This has been done by adding a call to NewTemporalFramework.generateNewTemporalModel(*model*) in CreekDocumentGroup.newKM().

A.2 New classes

In this section we will describe all the new classes that have been added to the jCreek-packages.

If the following classes are added to a TrollCreek-distribution, but the changes in the previous section are not implemented, it is still possible to use the temporal framework. However, it can then only be accessed through the code, not by using the knowledge model editor.

A.2.1 Representation

The following classes deal with the representation.

LongEntity (jcreek.representation.LongEntity) The timestamp values of the events are represented with the long data type. The rationale behind this is that Java uses a long value as a fundament for the Date-class. Instead of representing the timestamp values as encapsulated Date-objects we made the decision to use long values. This makes the makes the representation a little simpler since we only operate with Number-objects. A downside is that when we use the Relative-timestamp interpretation we are wasting space, since the timestamp values are assumed to within the Integer-range. This can be changed (hopefully) without any complications so that Relative-timestamp values use Integer-objects in the representation.

NewTemporalModel (jcreek.representation.NewTemporalModel) This class contains all the definitions our framework uses. It also has a method that takes a knowledge model as input and adds all the definitions used by the framework to that model. It is assumes that the input knowledge model is an IsoPod-model.

Subclasses of EntityType These are convenience classes for manipulating the different elements of the representation, there is really not much to say about them.

- NewTemporalCase
- TimeLine
- Event

TimeLineInterface (jcreek.representation.TimeLineInterface) This interface needs to be implemented by all classes that represent time lines in some form or other. The EntityType-subclass TimeLine does not implement this interface since we have an abstraction layer between the representation in the knowledge model and the representation internally. InternalTimeLine and AbstractTimeLine implement this interface.

InternalTimeLine (jreek.representation.InternalTimeLine) The reasoning mechanisms do not work directly on the time lines as they are represented in the knowledge model. They work on the TimeLineInterface-interface, which InteralTimeLine implements. InternalTimeLine encapsulates a TimeLine-object, and gives the reasoning mechanisms access to it through its methods.

This opens for having multiple representations of the actual time lines, i.e. absolute and relative dates.

AbstractTimeLine (jcreek.representation.AbstractTimeLine) Whereas InternalTimeLine encapsulates a TimeLine-object that exists in the knowledge model, AbstractTimeLine encapsulates an abstract time line that does not exist in the model. It also implements the TimeLineInterface-interface, so the reasoning mechanisms cannot tell the difference between InternalTimeLine and AbstractTimeLine.

In the code it used by TimeLineCompressor to represent the abstracted time line. There are some unresolved issued related to AbstractTimeLine. At the present time an abstract time line is constructed by adding pieces of different exisiting time lines; *has finding*-relations and their values to be more specific (see Chapter 6.5). This is not what we want in the long run. An abstract time line should be created from scratch. That is to say: With its own relationships.

The thought is that this class can be used by all abstractors to represent the abstracted time line. It contains methods for constructing events and placing them on a time line.

A.2.2 Reasoning

The following classes deal with the reasoning.

NewRetrieveResult (jcreek.reasoning.NewRetrieveResult) The original RetrieveResult-class had some references to the Case-class, which caused it to not accept the NewTemporalCase-class. NewRetrieveResult is a little more general (using EntityComparison in stead of CaseComparison here and there), but a few references to NewTemporalCase confines it to only working with this class of cases.

AbstractionInterface (jcreek.reasoning.AbstractionInterface) To allow for multiple implementations of the abstraction task, we have defined this interface that all abstractors must implement.

TimeLineCompressor (jcreek.reasoning.TimeLineCompressor) Our method for solving the abstraction task. How it functions is described in Chapter 6.3.2.

TempMatchGeneric (jcreek.reasoning.TempMatchGeneric) This is a subclass of EntityComparison that is supposed to be a superclass for all methods that implement the time line comparison task. The GUI-elements for viewing the results of the temporal case matching work with this class.

The class is declared abstract since the thought is that the subclasses are to do the actual reasoning. This class also takes care of non-temporal reasoning,

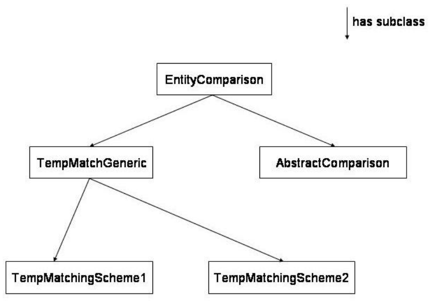


Figure A.1: The different subclasses of EntityComparison that are apart of the temporal framework

that is: It runs a regular case comparison and checks if its strength is above the threshold value, and if it is a temporal comparison is also done.

AbstractComparison (jcreek.reasoning.AbstractComparison) This is a derivative of the CaseComparison-class. It used to compare two events from two different time lines. The CaseComparison-class could be used for this if all events were represented in the knowledge model, but because of our abstraction task this is not given. It is basically a stripped down CaseComparison-class that operates on arrays of *has finding*-relations instead of on entities.

The class is a subclass of EntityComparison. However, the constructor of EntityComparison is not called. This is because this class works on arrays of findings, so we have no entities to give the constructor of EntityComparison. A consequence of this is that the comparisons by this class are not cached. However, the comparisons between the findings use CaseFindingComparison, just like CaseComparison, so those comparisons are cached.

TempMatchingScheme1 (jcreek.reasoning.TempMatchingScheme1) This is one of the comparators we developed. How it functions is described in Chapter 6.4.2.

TempMatchingScheme2 (jcreek.reasoning.TempMatchingScheme2) This is the other comparator we developed. It is described in Chapter 6.4.2.

A.2.3 GUI-related

TemporalRetrieveResultPane (jcreek.gui.reasoning.TemporalRetrieveResultPane) This class is based on RetrieveResultPane, which could not be used because of some references in it. In theory there is nothing that precludes ordinary and temporal cases of using the same RetreiveResultPane-class, one just needs to make some generalizations here and there.

TemporalComparisonPane (jcreek.gui.reasoning.TemporalComparisonPane) A class for viewing the results of a TempMatchGeneric-object. It is to TempMatchGeneric what CaseComparisonPane is to CaseComparison.

MultipleTimeLineViewer (jcreek.gui.representation.MultipleTimeLineViewer) Viewing the time lines in the form of a semantic net can be an ungrateful chore, so this class will provides a simple time line viewer. It can take as input several time lines and will show them all. It is an extension of the JPanel-class.

ColorScheme (jcreek.gui.representation.ColorScheme) This class is used to generate colours for the events that are displayed in the time line viewer. At the present the colours are random. This class is only accessed by the MultipleTimeLineViewer-class.

A.2.4 Other

ViewTimeLineAction (jcreek.cke.command.ViewTimeLineAction) This actions opens time line viewer that shows a single time line if the action was invoked on a TimeLine-entity, or all the time lines if it was invoked on a NewTemporalCase-entity.

TempMatch (jcreek.cke.command.TempMatch) This is the action that initiates the temporal case matching.

CaseGenerator (jcreek.util.CaseGenerator) This is meant to be general class for generating temporal cases. It does not actually create cases, but implements methods that will of use to such a task.

CaseGeneratorProcess1 (jcreek.util.CaseGeneratorProcess1) This is a subclass of CaseGenerator that create the cases used in our example. What rules it follows is described in Chapter 7.3.

ModelUtilities (jcreek.util.ModelUtilities) Contains methods for opening and saving a knowledge model. These methods were taken from the CreekExample-class. The thought behind extracting the methods from CreekExample and placing them in its own class, was that it seemed strange that classes should refer to methods that were used in a class that was an example. The ModelUtilities-class is used by the CaseGenerator-class, and has also been used in the debugging of several other classes.

Bibliography

[Aam91] Agnar Aamodt. *A Knowledge-Intensive Integrated Approach to Problem Solving and Sustained Learning.* PhD thesis, University of Trondheim, Department of Electrical Engineering and Computer Science, 1991.

[Aam93] Agnar Aamodt. Explanation-driven case-based reasoning. In Stefan Wess, Klaus-Dieter Althoff, and Michael M. Richter, editors, *EWCBR*, volume 837 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 1993.

[Aam04] Agnar Aamodt. Knowledge-intensive case-based reasoning in creek. In Funk and González-Calero [FGC04], pages 1–15.

[AB03] Kevin D. Ashley and Derek G. Bridge, editors. *Case-Based Reasoning Research and Development, 5th International Conference on Case-Based Reasoning, ICCBR 2003, Trondheim, Norway, June 23-26, 2003, Proceedings*, volume 2689 of *Lecture Notes in Computer Science*. Springer, 2003.

[AGdM03] Josep Lluís Arcos, Maarten Grachten, and Ramon López de Mántaras. Extracting performers' behaviors to annotate cases in a cbr system for musical tempo transformations. In Ashley and Bridge [AB03], pages 20–34.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[AP94] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.

[BM98] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[Bre04] Tore Brede. Time sequences in case-based reasoning (a reimplementation in jcreek), 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[CP02] Susan Craw and Alun D. Preece, editors. *Advances in Case-Based Reasoning, 6th European Conference, ECCBR 2002 Aberdeen, Scotland, UK, September 4-7, 2002, Proceedings*, volume 2416 of *Lecture Notes in Computer Science*. Springer, 2002.

[Cun98] Padraig Cunningham. CBR: Strengths and weaknesses. In *IEA/AIE (Vol. 2)*, pages 517–524, 1998.

[DS83] Joseph B. Kruskal David Sankoff, editor. *Time Warps, String Edits, And Macromolecules: The Theory And Practice Of Sequence Comparison*. Addison-Wesley, 1983.

[FGC04] Peter Funk and Pedro A. González-Calero, editors. *Advances in Case-Based Reasoning, 7th European Conference, ECCBR 2004, Madrid, Spain, August 30 - September 2, 2004, Proceedings*, volume 3155 of *Lecture Notes in Computer Science*. Springer, 2004.

[GAdM04] Maarten Grachten, Josep Lluís Arcos, and Ramon López de Mántaras. Tempoexpress, a cbr approach to musical tempo transformations. In Funk and González-Calero [FGC04], pages 601–615.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Han00] Bjarne K. Hansen. Weather prediction using case-based reasoning and fuzzy set theory. Master's thesis, Dalhousie University, Daltech, 2000.

[HR01] Bjarne K. Hansen and Denis Riordan. Fuzzy case-based system for weather prediction. *Engineering Intelligent Systems*, 3:139–145, 2001.

[Jac97] Michel Jaczynski. A framework for the management of past experiences with time-extended situations. In Forouzan Golshani and Kia Makki, editors, *Proceedings of the Sixth International Conference on Information and Knowledge Management (CIKM'97), Las Vegas, Nevada, November 10-14, 1997*, pages 32–39. ACM, 1997.

[Jær01] Martha Dørum Jære. Time sequences in case-based reasoning. Master's thesis, Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science, 2000, 2001.

[JAS02] Martha Dørum Jære, Agnar Aamodt, and Pål Skalle. Representing temporal knowledge for case-based prediction. In Craw and Preece [CP02], pages 174–188.

[JT99] Michel Jaczynski and Brigitte Trousse. Broadway: A case-based system for cooperative information browsing on the world-wide-web. In Julian A. Padget, editor, *Collaboration between Human and Artificial Societies*, volume 1624 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 1999.

[Ker95] Elpida T. Keravnou. Modelling medical concepts as time-objects. In Pedro Barahona, Mario Stefanelli, and Jeremy C. Wyatt, editors, *AIME*, volume 934 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 1995.

[KLBL97] John D. Hastings Karl L. Branting and Jeffrey A. Lockwood. Integrating cases and models for prediction in biological systems. *AI Applications*, 11(1):29–48, 1997.

[Mar04] Francisco J. Martín. *Case-Based Sequence Analysis in Dynamic, Imprecise, and Adversarial Domains.* PhD thesis, Universitat Politècnica De Catalunya, 2004.

[MK94] Jixin Ma and Brian Knight. A general temporal theory. *Comput. J.*, 37(2):114–123, 1994.

[MK03] Jixin Ma and Brian Knight. A framework for historical case-based reasoning. In Ashley and Bridge [AB03], pages 246–260.

[MP04] Francisco J. Martín and Enric Plaza. Ceaseless case-based reasoning. In Funk and González-Calero [FGC04], pages 287–301.

[PA02] Enric Plaza and Joseph-Lluis Arcos. Constructive adaption. In Craw and Preece [CP02], pages 306–320.

[Per98] Francisco C. Pereira. Composing music with case-based reasoning. In *Proc. Of the European Conference of Artificial Intelligence (ECAI98)*, 1998.

[RS97] Ashwin Ram and J. C. Santamaria. Continuous case-based reasoning. *Artificial Intelligence*, 90(1-2):25–77, 1997.

[SG01] Rainer Schmidt and Lothar Gierl. Temporal abstractions and case-based reasoning for medical course data: Two prognostic applications. In Petra Perner, editor, *MLDM*, volume 2123 of *Lecture Notes in Computer Science*, pages 23–34. Springer, 2001.

[SG02] Rainer Schmidt and Lothar Gierl. Case-based reasoning for prognosis of threatening influenza waves. In Petra Perner, editor, *Industrial Conference on Data Mining*, volume 2394 of *Lecture Notes in Computer Science*, pages 99–108. Springer, 2002.

[Sha97] Yuval Shahar. A framework for knowledge-based temporal abstraction. *Artificial Intelligence*, 90(1-2):79–133, 1997.

[SHPG96] Rainer Schmidt, Bernhard Heindl, Bernhard Pollwein, and Lothar Gierl. Abstractions of data and time for multiparametric time course prognoses. In Ian F. C. Smith and Boi Faltings, editors, *EWCBR*, volume 1168 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 1996.

[SM99] Yuval Shahar and Mark A. Musen. Evaluation of a temporal-abstraction knowledge-acquisition tool. In *Twelfth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada*, 1999.

[Smi91] David K. Smith. *Dynamic Programming A Practical Introduction.* Ellis Horwood, 1991.

[Sør00] Frode Sørmo. Plausible inheritance; semantic network inference for case-based reasoning. Master's thesis, Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science, 2000.

[TW03] Asmir Tobudic and Gerhard Widmer. Playing mozart phrase by phrase. In Ashley and Bridge [AB03], pages 552–566.

[WG93] Andreas S. Weigend and Neil A. Gershenfeld, editors. *Time Series Prediction: Forecasting the future and understanding the past*, volume Proc. Vol. XV of *SFI Studies in the Sciences of Complexity.* Addison-Wesley, 1993.