

AD-A245 600

Netherlands organization for applied scientific research



TNO Physics and Electronics Laboratory

P.O. Box 96864  
2509 JG The Hague  
Oude Waalsdorperweg 63  
The Hague, The Netherlands  
Fax +31 70 328 09 61  
Phone +31 70 326 42 21

TNO-report

report no.  
FEL-91-B308

copy no.

9

title

Data Fusion: Temporal Reasoning and Truth Maintenance

TP 91-3885



Nothing from this issue may be reproduced and/or published by print, photoprint, microfilm or any other means without previous written consent from TNO. Submitting the report for inspection to parties directly interested is permitted.

In case this report was drafted under instruction, the rights and obligations of contracting parties are subject to either the 'Standard Conditions for Research Instructions given to TNO' or the relevant agreement concluded between the contracting parties on account of the research object involved.

© TNO

author(s):

Ir. A.P. Keene  
Drs. M. Perre

DTIC  
ELECTE  
FEB 04 1992  
S D D

date:

November 1991

TDCK KAPPORTCENTRALE  
Frederikkazerne, Geb. 140  
van den Burchlaan 21  
Telefoon: 070-3166394/6395  
Telefax : (31) 070-3166202  
Postbus 90701  
2509 LS Den Haag



classification

title : unclassified  
abstract : unclassified  
report text : unclassified  
appendices A-C : unclassified

no. of copies : 35

no. of pages : 99 (incl. append., excl. RDP & dist. list)

appendices : 3

All information which is classified according to Dutch regulations shall be treated by the recipient in the same way as classified information of corresponding value in his own country. No part of this information will be disclosed to any party.

92-02826



92 2 03 165

Reproduced From  
Best Available Copy

20000 831000

Report no. : FEL-91-B308  
 Title : Data Fusion; Temporal Reasoning and Truth Maintenance  
 Author(s) : Ir. A.P. Keene, Drs. M. Perre  
 Institute : TNO Physics and Electronics Laboratory  
 Date : November 1991  
 NDRO no. :  
 No. in pow '91 : 704.2  
 Research supervised by : Drs. M. Perre  
 Research carried out by : Ir. A. P. Keene

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability codes	
Dist	Avail. and/or Control
A-1	



### Abstract (unclassified)

This report contains a survey of two techniques that can be used in the field of data fusion: temporal reasoning and truth maintenance. The automatic fusion of intelligence reports necessitates taking into account the factor time. Incoming messages can lead to new interpretations of the current battlefield situation, changing previously made hypotheses. A data fusion system must also be able to make a prediction of what sightings are to be expected, e.g. in the case of columns of vehicles moving past different sensors. This report describes a temporal database system that can capture (some part) of the volatility of the intelligence processing domain. While processing intelligence reports there is always an amount of uncertainty and incompleteness that has to be dealt with. So there is a need for maintaining different lines of reasoning or hypotheses pertaining to the battlefield situation concurrently, and incorporating new information as it becomes available. In this report an assumption-based truth maintenance system provides a framework in which this problem can be solved. A prototype has been developed to demonstrate the applicability of the aforementioned techniques. This prototype, called Mefisto (Modular Environment for Fusion and Interpretation of Sensor data in Tracking Opposing forces), is a simple knowledge-based system integrated with a temporal truth maintenance facility.

---

Rapport nr. : FEL-91-B308  
Titel : Data Fusion: Temporal Reasoning and Truth Maintenance  
  
Auteur(s) : Ir. A.P. Keene, Drs. M. Perre  
Instituut : Fysisch en Elektronisch Laboratorium TNO  
Datum : November 1991  
  
HDO-opdrachtnummer : -  
Nr. in IWP91 : 704.2  
Onderzoek uitgevoerd o.l.v. : Drs. M. Perre  
Onderzoek uitgevoerd door : Ir. A.P. Keene

---

## Samenvatting (ongerubriceerd)

Dit rapport bevat een overzicht van twee technieken die kunnen worden gebruikt op het gebied van datafusie: "temporal reasoning" en "truth maintenance". De automatische fusie van inlichtingenrapporten brengt de noodzaak met zich mee om de factor tijd in de beschouwing te betrekken. Binnenkomende berichten kunnen leiden tot nieuwe interpretaties van de actuele situatie op het gevechtsveld, waarbij eerder opgestelde hypothesen worden aangepast. Een datafusie-systeem moet ook in staat zijn om een voorspelling te doen over te verwachten waarnemingen van bijvoorbeeld kolonnes voertuigen die zich langs verschillende sensoren voortbewegen. In dit rapport is een "temporal database system" beschreven dat (een deel van) de "vluchtigheid" van het inlichtingenverwerkingsproces kan vangen. Bij het verwerken van inlichtingenrapporten moet altijd rekening gehouden worden met een bepaalde mate van onzekerheid en onvolledigheid. Hierdoor ontstaat de noodzaak om tegelijkertijd verschillende redeneringen of hypothesen, ten aanzien van de situatie op het gevechtsveld te onderhouden en deze, zodra nieuwe informatie beschikbaar komt, overeenkomstig te wijzigen. In dit rapport is een "assumption-based truth maintenance system" beschreven, dat een oplossing biedt voor deze problemen. Er is een prototype ontwikkeld om de bruikbaarheid van de eerder genoemde technieken te demonstreren. Dit prototype, genaamd Mefisto (Modular Environment for Fusion and Interpretation of Sensor data in tracking Opposing forces), is een eenvoudig kennisstelsel geïntegreerd met faciliteiten voor "temporal truth maintenance".

	<b>ABSTRACT</b>	<b>2</b>
	<b>SAMENVATTING</b>	<b>3</b>
<b>1</b>	<b>INTRODUCTION</b>	<b>6</b>
<b>2</b>	<b>TEMPORAL REASONING AND TRUTH MAINTENANCE</b>	<b>8</b>
2.1	Introduction	8
2.2	Temporal reasoning	9
2.2.1	Approaches to temporal reasoning	9
2.2.2	Temporal reasoning and data fusion	11
2.2.3	A temporal taxonomy	13
2.3	Truth maintenance	19
2.3.1	Non-monotonic reasoning	19
2.3.2	The Assumption-based Truth Maintenance System	20
<b>3</b>	<b>PROTOTYPING MEFISTO</b>	<b>25</b>
3.1	Introduction	25
3.2	The battlefield environment	26
3.2.1	The area of intelligence responsibility	26
3.2.2	The order of battle	27
3.3	System overview	29
3.3.1	The structure of the process	29
3.3.2	The structure of the data	31
3.3.3	The structure of the dialogue	34
3.4	Technology overview	37
3.4.1	Forward chaining production system	37
3.4.2	Truth maintenance and temporal reasoning in Mefisto	39
3.4.2.1	Keeping track of reason	39
3.4.2.2	Keeping up with time	40
3.4.3	Inference and temporal truth maintenance	41
3.4.4	Data fusion in Mefisto	47

4	CONCLUSIONS AND RECOMMENDATIONS	51
	ACRONYMS AND ABBREVIATIONS	53
	REFERENCES	54
	APPENDIX A: THE TEMPORAL DATABASE SYSTEM	
	APPENDIX B: INFERENCE AND INTERACTION TDB-ATMS	
	APPENDIX C: THE ASSUMPTION-BASED TRUTH MAINTENANCE SYSTEM	

## 1 Introduction

This report is a sequel to [Keene & Perre, 1990], which gave a general overview of various approaches to data fusion in the military intelligence processing domain. In the following chapters we will focus on two techniques which are of importance to an automatic data fusion system: temporal reasoning and truth maintenance.

Temporal reasoning is "reasoning about time". In itself this statement may not seem that surprising. Of course automated *real-time* systems exist that interact with physical processes in the real world: these systems have to keep track of time, in one way or another. However, an important observation can be made in the command and control domain: not many systems are in use that can *reason* about time. Especially the process of data fusion is tightly connected with time. Sensor data arrives at different points in time, without the assurance that it can be interpreted and processed sequentially. It can be stated that in data fusion applications the non-monotonic characteristics prevail.

Truth maintenance is a method to monitor the *truth* status of elements in a data base system. This status may depend on assumptions which lay at the roots of these data elements. Should these assumptions become invalid, then the truth status of conclusions which they support (i.e. other data elements) also changes. A truth maintenance system provides a framework in which *dependencies* between data elements can be represented explicitly. If there is some change in one element, then the consequences for other elements which are dependent on it, will be deduced.

While processing intelligence reports temporal reasoning and truth maintenance are to an extent complementary. Suppose that at one point in time there is a certain amount of information available from which conclusions about the battlefield situation can be drawn. This information and the ensuing conclusions could be called "time-stamped". If at some later point in time additional information becomes available, contradicting or augmenting information already received, it could be necessary to adapt earlier made conclusions concerning the battlefield.

The following topics are presented in the remainder of this report. Chapter 2 gives a more general description of temporal reasoning and truth maintenance. The interest is focused on different approaches to temporal reasoning and the relation between this technique and data fusion. A

concise treatment of non-monotonic reasoning forms the starting point of the discussion on truth maintenance systems, culminating in a presentation of the assumption-based variety. These theoretical notions are operationalized in chapter 3, which contains a description of the analysis and design process of the Mefisto prototype (Modular Environment for Fusion and Interpretation of Sensor data in Tracking Opposing forces). After defining the battlefield environment, two main points are addressed: Firstly, the structure and functionality of Mefisto. Secondly, the extent to which the theoretical notions of the previous chapter have been incorporated into this prototype system. Chapter 4 contains the conclusions and recommendations based on our experiences while building Mefisto. After summing up the acronyms, abbreviations and references, this report concludes with appendices containing excerpts from the Mefisto code for the temporal query language, the assumption-based truth maintenance system and the interaction between these two.

## 2 Temporal reasoning and truth maintenance

### 2.1 Introduction

The main goal of data fusion is to combine the available data on a certain area of interest to achieve as best an estimate as possible of the objects in the area, their groupings, movements and combat activities. The data may come from different sensors, is governed by uncertainty and incompleteness, and are "snapshots" of a continuously changing domain. We note the distinction between sensor fusion, which refers to the correlation of low-level sensor data (e.g. radar, infra-red), and data fusion, which is mainly concerned with combining data from the vehicle level up to complex unit aggregations.

Generally, information in the form of incoming reports will define some activity associated with a unit of an opposing force, describing vehicle types, equipment and personnel sighted, movements with an associated direction and speed, location and time of the sighting. It must be decided what type of unit the report describes and possibly the identity of the unit. This *classification* associates the report with a specific unit type. An attempt is made to confirm the sighting by checking units established from earlier reports, to see if and how the information matches the current known situation. It will often be the case that there is no unique solution for the *correlation* of the reported information, a report may refer to a unit associated with an earlier report, but it may be another unit, not currently known to exist on the battlefield. Thus it will be necessary to keep track of units in both space and time. When separate units have been distinguished, the next step will be the *aggregation* of units into larger unit formations. The propagation of assumptions when e.g. a unit is taken to be such and such or assumed to be part of a certain encompassing unit has to be tracked as well. Consequently, it will be necessary to maintain multiple lines of reasoning about the area of interest.

Temporal aspects predominate in data fusion. In the first place, the situation at time now based on the current information is important. Typically, the currently available information taken at face value will not be sufficient for determining a coherent picture of the actual situation. It is necessary to be able to engage in some form of prediction. An example is the tracking of units in the system to answer queries such as "Which units could be in the vicinity of location X within



one hour from now?" An answer involves estimating the zone that is relevant, establishing the units that are in that zone at time now, associating information as to velocities and movement capabilities, direction of movement, etc. This also serves to indicate that temporal reasoning is very closely related to spatial reasoning.

Apart from prediction, a retroactive adaptation of the current situation will often be necessary. This is the case when for instance an incoming report provides information contradicting a previous report, but also applies to sensors such as a drone, an unmanned airborne sensor that may provide information only after its flight. Both predictive and retroactive adaptations therefore require a mechanism for recording assumptions that may at a later stage become invalid. Temporal reasoning thus requires a form of truth maintenance, and as outlined above so does the very nature of data fusion. The assumption-based truth maintenance system (ATMS) is recognized as the most promising means for this purpose. This chapter addresses techniques for both temporal reasoning and truth maintenance.

## 2.2 Temporal reasoning

### 2.2.1 Approaches to temporal reasoning

Time is a significant factor in common-sense reasoning, yet it is not actually dealt with in conventional database systems. The contents of the database are considered to be timelessly true, defied only by the explicit deletion from the database. The idea behind a temporal database is to represent the notion that information about the world is generally incomplete and continuously changing. An important aim is to take into account that there are many possible states of affairs (worlds, contexts, situations), based on conditional predictions to fill gaps resulting from the incompleteness of information. The known information is stored in a temporal database, and a problem solver constructs a number of possible completions of the knowledge, choosing the most likely solution. This can not be represented by the "timelessly true" facts in e.g. a relational database, and without some method of completing this knowledge, because this would add up to exactly *one* state of affairs.

Taking the relational database model as the current convention, we have a collection of relations, each relation consisting of a set of tuples with an equivalent set of attributes, represented as tables. The current contents of the tables form the state of the database, adapted by the operations insert,

delete and update. Only the current state of the database is retained, past states are discarded. An extension of this is the representation of the historical state per relation in a *historical* database. Previous states of the database are not retained as such, but are represented in the history of each relation in the database. Modifications can be made to the relations when errors are detected or in answer to update requests. This is accomplished by adding historical records each time an entry is updated. Thus, historical databases support arbitrary modification and represent current knowledge about the present, as do conventional databases, but they also represent current knowledge about the past. An example of a (laboratory) database management system (DBMS) that offers facilities for the implementation of a historical database is POSTGRES ([Stonebraker et al., 1986; Stonebraker et al., 1990]). A further extension is outlined in [Snodgrass & Ahn, 1986], for a bibliography on temporal databases we refer to [Stam & Snodgrass, 1989].

A more formal view is the approach taken when applying temporal logics (modal logics) by extending the predicate calculus with temporal operators, describing notions such as "always from now on", "some time in the past", etc., formulated in the context of a possible worlds semantics. For instance, the notion "P was valid at some time T" would be true if P is true in all possible worlds "covering" T. The strict formalism ensures that logic programs governed by temporal logic are consistent and complete, however this formalism is a major drawback as well, due to the unnatural representation. There is a large number of theories on temporal logic, we refer to [Galton, 1987].

Taking in mind our domain, we are interested foremost in the following sequence of activities: the recording of incoming messages, which are transferred into battlefield entities, which in turn are grouped into formations corresponding to known data on order of battle. A data fusion system needs the functionality of retroactive adaptation of database records. The history of the actual situation at some time is extremely important. When new information points out that what was thought to be a tank battalion was in fact a complete regiment it must be possible to update this in the corresponding (historical) tuple in the database. This is further clarified by taking in mind situation assessment and anticipation of future enemy movements.

It is our view that temporal information in the context of data fusion can be sufficiently represented by facilities supporting historical queries. We advocate an approach that lies somewhere in between the above, such as used in the planning system Time Map Manager (TMM) of Dean [Dean, 1985; Dean, 1986; Dean & McDermott, 1987; Firby & McDermott,

1987]. It is a computational approach, centering around techniques for managing a database of assertions corresponding to the occurrence of events and the persistence of their effects over time. The approach takes into account that temporal information is incomplete and defeasible. Therefore, the approach stresses the necessity that a problem solver has to be able to make predictions on the basis of certain assumptions, that these assumptions may at some later time become invalid and hence the predictions based on these assumptions as well. Moreover, it is recognized that most common-sense reasoning involves reasoning about temporal events and that durations of events are often known within metric bounds. This is reflected in the choice of "bringing time into the relations", replacing classic assertions by data structures incorporating interval representations of temporal validity. We will take this approach as a starting point, with the main objective of determining if and how it can be utilized for the representation of temporal dependencies in the domain of data fusion.

### 2.2.2 Temporal reasoning and data fusion

The strategy behind temporal reasoning is what [Dean & McDermott, 1987] refer to as *shallow temporal reasoning*. Shallow temporal reasoning is characterized by breaking down the reasoning process into the following steps:

1. Generate a set of candidate hypotheses;
2. Select one hypothesis from among the candidates;
3. Use the selected hypothesis as a basis for prediction;
4. Respond to unforeseen consequences noticed in the course of prediction.

The hypotheses correspond to the possible states of affairs, which are the result of the known information on events, their effects, their time of occurrence and duration. For instance, a hypothesis based on the sightings of certain units in each other's vicinity could be the indication of the presence of a larger encompassing unit.

The selected hypothesis is then the basis for making predictions depending upon the hypothesis. Predictive inferences can take the form of what Dean calls *controlled forward inferences* or *automatic projections* [Dean, 1986]. Controlled forward inferences are achieved by the application of forward chaining rules, directly adding deductions to the database (for example, the deduction that a new report refers to a tank company because the conditions as to vehicle types

and quantities are satisfied). Automatic projection is a mechanism that responds to trigger-events. If antecedent conditions are satisfied and a trigger event occurs then after some delay a consequent effect is added to the database.

Finally, the fourth step is described as responding to unforeseen consequences noticed in the course of prediction. It entails foremost the need for truth maintenance, for keeping track of the assumptions underlying a certain hypothesis, and when assumptions become invalid, invalidating the hypothesis and replacing it by another.

The above cyclic description of shallow temporal reasoning is in effect a concise formulation of the essential process underlying data fusion. As reports containing sensor data are transferred into associated units by a classification process, hypotheses are generated concerning the type of unit involved. A unit is assumed to have some unit type at say time  $T_0$ , at a later time  $T_1$  the unit is aggregated into a larger unit, which will be further used for concluding aspects about the battlefield situation at time  $T_2$ . When information at time  $T_3$  allows the conclusion that the classification of the unit was incorrect after all, all inferences made since then using the unit classification as a condition must be defied. This work is done by a truth maintenance system, which must reply with a list of the conclusions added to the database that are thus defied. Effectively, the database situation must then be turned back to time  $T_0$  in the sense that the conclusions are removed from the database (in fact, they will not be deleted but "clipped" with  $T_3$  as endtime, we refer to the next paragraph) and an optional re-run of the inference mechanism with the new unit type at  $T_0$  will then result in new conclusions, effective from some time  $T_4$ .

A temporal reasoning application will typically require a temporal database, a (temporal) query language, an inference mechanism and some form of truth maintenance system [Dean & McDermott, 1987]. A problem solver will incorporate a temporal query language and an inference mechanism. Temporal assertions are stored in a database, which can be queried by the problem solver. The temporal assertions and derivations based on these assertions are passed to a truth maintenance system (TMS), which performs the bookkeeping of assumptions and justifications, and has access to the database as well. The TMS notifies the problem solver of changes in the validity of nodes. The main notions concerning the temporal database, a temporal query language and inference will be addressed in the next paragraph. Truth maintenance is addressed in section 2.3.

### 2.2.3 A temporal taxonomy

The basic temporal notion is the *interval*. An interval is a pair of points, consisting of a begin- and an endpoint, such that the beginpoint precedes or coincides with the endpoint. In the latter case it is a single timepoint, which is thus represented in two dimensions as an interval (point,point).

An *occasion* (Dean also refers to "time token") is a fact or instantiated proposition. It has associated with it a temporal distance statement that we will call *timedist*, which is the central data structure contained in the *temporal database*. The *timedist* structure represents the duration of an occasion, stating the begin- and endpoints, constrained within a lower bound and an upper bound. The following representation of the *timedist* structure is the notation in Prolog:

```
timedist(begin(Occasion),end(Occasion),Low,High).
```

The *timedist* structure records relative time, and absolute time is calculated using only relative temporal distances [Dean & McDermott, 1987]. The reason is that Dean's Time Map Manager concerns the domain of planning, which deals mostly with relative time, e.g. the duration of task1 is known to have a lower bound of 20 minutes, and an upper bound of 35 minutes, whereas the precise begin- and endtimes of task1 are not known beforehand, but the end of task1 must precede the beginning of task2, etc. Thus the minimum and maximum durations of an occasion are recorded, related to the begin- and endpoints, which are expressed as "begin(Occasion)" and "end(Occasion)". The domain of data fusion also incorporates relative temporal information, e.g. it may be known that the displacement of a unit from location A to location B has a certain duration, or that given the sighting of some unit this implies that a certain other unit is expected to pass within an hour.

The *timedist* structure explicitly contains relative temporal information. However, it implicitly represents absolute temporal information, which will be explained further on. This allows the representation of default information, because given a timepoint and an occasion such that its begin-to-end interval does not contain that timepoint, this implies that the occasion was not valid at that time. This is a powerful way of recording historical information. It is comparable in its intention to a two-dimensional indexing method for geographic database applications using "minimal bounding rectangles" for the storage of geographic information, as opposed to the conventional one-dimensional indexing methods.

To calculate absolute time from relative time it is necessary to adopt some reference point. The registration of temporal information for an occasion of which the begintime is known is then accomplished by asserting two *timedist* entries in the temporal database, the first containing the entries corresponding to the reference point and the beginpoint of the occasion, the second containing the entries for the begin- and endpoint of the occasion. For example, an occasion valid from timepoint  $t_1$  until timepoint  $t_2$  is entered in the temporal database as follows:

```
timedist(ref,begin(Occasion),t1,t1).  
timedist(begin(Occasion),end(Occasion),t1,t2).
```

The first entry has  $t_1$  as lower and upper bound, stating that the occasion is known to exist since timepoint  $t_1$ . The second *timedist* entry contains the information related to the endpoint of the occasion. The above representation suffices for the calculation of all temporal distances, acquiring a powerful mechanism for the calculation of temporal distances between occasions with relative timepoints, while obtaining absolute time from these relative temporal dependencies by the calculation of the shortest possible path from A to B.

There is a visual representation of the temporal notions (*temporal imagery*), the construction of *time maps*. In figure 2.1 a time map is shown. A time map is a graph with vertices referring to points in time, and distance constraints between the points of the graph along the directed edges. The notation [A,B] in fig. 2.1 corresponds to the minimal and maximal distance between two points. These may be negative distances, indicating "travelling" in the past. Thus, we can calculate the distance bounds in going from say  $pt_1$  to  $pt_2$ . In the example there are three possible paths leading from  $pt_1$  to  $pt_2$ , two of which traverse  $pt_3$  on route.

There is an important distinction between three types of (temporal) distances: a simple distance function connecting two separate points (as captured by the *timedist* data structure), the distance function corresponding to the distance of a path, and the distance that represents the greatest lower bound (GLB) and least upper bound (LUB) of the distances of these paths, thus the shortest path. In the case of figure 2.1, the *timedists* are the four separate distances. The three available paths from  $pt_1$  to  $pt_2$  have respective distances [5,9], [6,8] and [-1,11]. On aggregation of the minimum and maximum bounds of these paths the distance representing the GLB and LUB is [6,8].

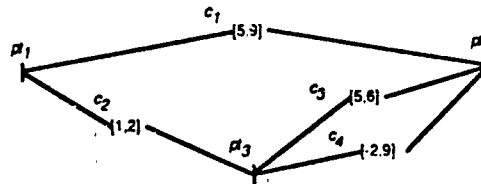


Figure 2.1: Time map illustrating distance constraints [D an & McDermott, 1987]

We will further illustrate the temporal notions with an example. We refer to Appendix A for a listing of the Prolog source code of the predicates used below. We have two occasions, occ1 and occ2. We add the following entries to the temporal database:

```

timedist(ref,begin(occ1),100,100).
timedist(begin(occ1),end(occ1),0,inf).
timedist(ref,begin(occ2),150,150).
timedist(begin(occ2),end(occ2),10,30).

```

This implies that occ1 is valid from timepoint 100 on, and occ2 is valid from timepoint 150 with a duration between 10 and 30 time units. We have implemented a temporal distance function with a shortest path calculation as explained above. When we query the distance from begin(occ1) to end(occ2) the result is the following:

```

| ?- temporal_distance(begin(occ1),end(occ2),Min,Max).
Min = 60,
Max = 80.

```

This illustrates that absolute time is derived from relative temporal instances, via the reference point "ref". The temporal dependency between occ1 and occ2 is not explicitly entered into the temporal database, but calculated via the reference point.

Querying the lower and upper bounds of an occasion results in the following, stating that occ2 starts at time point 150, and ends between time points 160 and 180.

```
| ?- temporal_bounds(occ2).  
[ref,begin(occ2),150,150]  
[ref,end(occ2),160,180]
```

A *temporal query language* is necessary to further use the temporal database for determining e.g. whether a certain point comes before another point, whether an occasion is true throughout a specific interval, whether two occasions are overlapping. Effectively these requirements above are met by predicates that represent notions such as "timepoint less than" and "occasion true throughout an interval" that can be used in temporal queries. These predicates concern the temporal relations between occasions and time points, among occasions, and between occasions and intervals. They are implemented as backward chaining rules. In this way a powerful mechanism is developed for querying the implicit temporal dependencies among the various occasions. Beside the timedist instances entries of predicates of the temporal query language such as "during" and "true\_throughout" can be asserted directly to the temporal database as well.

The following is an example of a predicate of the temporal query language, which when called upon calculates those occasions that satisfy the condition that the period in which Occ1 is valid lies completely inside the period that Occ2 is valid. We refer to Appendix A for a complete list.

```
during(Occ1,Occ2):-  
    temporal_distance(begin(Occ1),end(Occ1),Min1,Max1),  
    temporal_distance(begin(Occ2),end(Occ2),Min2,Max2),  
    not(Occ1 = Occ2),  
    less_equal(Min2,Min1),  
    less_equal(Max1,Max2).
```

For the example above, this query results in the following answer, meaning that occ2 is found to occur during occ1.

```
| ?- during(X,Y).  
X = occ2, Y = occ1.
```



There is a distinction between two types of occasions. Firstly, an *event* refers to an occasion with a duration that can be reasonably predicted, such as the take-off of an airplane, or the movement of a tank company from location X to location Y. In the example above, *occ2* is an event. Secondly, a *persistence* is an occasion applicable to change over time. Persistences reflect what is believed to have occurred, and by default the upper bound of a persistence is set to infinite (the occasion is assumed to be persistent). In the example above *occ1* is a persistence, with its upper bound set to infinite.

A mechanism referred to as *clipping* changes persistences into clipped persistences when called for, indicating that the occasion is no longer valid as from the inserted cliptime. This clipping is essentially the replacement of an endtime (generally, with an earlier endtime), and an accompanying replacement of the upper bound. Often, the clipped occasion will be a persistence with an upper bound "infinite" between begin- and endpoint. The upper bound is replaced by the time corresponding to the beginpoint of some later occasion replacing it. To further illustrate the clipping, we list the temporal bounds of *occ1* from the example above and clip the occasion:

```
| ?- temporal_bounds(occ1).  
    [ref,begin(occ1),100,100]  
    [ref,end(occ1),100,inf]  
  
| ?- clip_node(occ1,190).
```

Due to the relative notions involved, the result of the clipping is that the second timedist entry is updated. Initially, the lower and higher bound were 0 and inf, these are replaced by the difference between the cliptime and the begintime. In the case of *occ1*, the timedist entries become as follows:

```
timedist(ref,begin(occ1),100,100).  
timedist(begin(occ1),end(occ1),90,90).
```

Figure 2.2 below illustrates the clipping mechanism for the example that was given here.

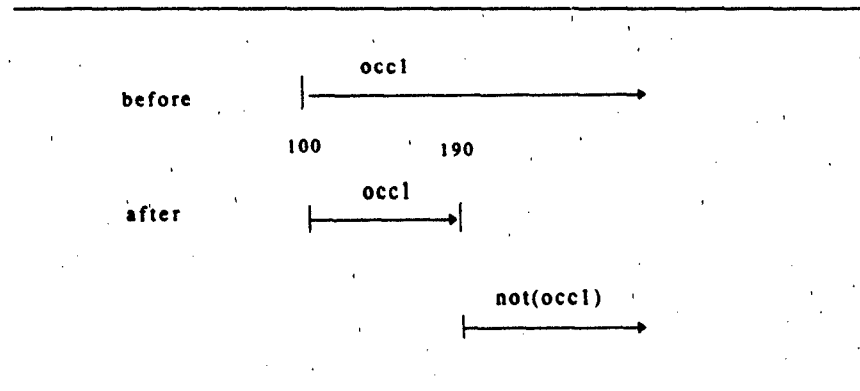


Figure 2.2: Time map illustrating the clipping of an occasion

Now, when the temporal bounds are calculated, the result is the following:

```
?- temporal_bounds(occl).  
[ref,begin(occl),100,100]  
[ref,end(occl),190,190]
```

The inference mechanism in a temporal database application uses three types of rules: *backward chaining* rules, *forward chaining* rules and *clipping* rules. The backward chaining rules are used, as stated above, for the implementation of the temporal queries. The forward chaining rules contain the domain knowledge and will be used to perform tasks corresponding to classification and prediction. Clipping rules respond to the occurrence of *apparently contradictory* occasions - for instance, "unit X non-active" and "unit X active" - by clipping the earlier occasion, thus forcing its endtime to precede the beginning of the later occasion.

The basic temporal taxonomy described provides a quite natural common-sense representation of time. The history of database entries is implicitly contained in the representation of begin- and endpoints in an accompanying temporal relation. A problem is posed by the question of control: how should the various rules be applied. The application of temporal reasoning will be further addressed in the next chapter, in combination with the application of truth maintenance, which is the subject of the next paragraph.

## 2.3 Truth maintenance

### 2.3.1 Non-monotonic reasoning

The ability to reason about and adapt to a changing environment is an important aspect of intelligent behaviour. Given a system that performs reasoning (if <condition> then <conclusion>), adaptation requires the ability to alter conclusions when conditions are contradicted or no longer met. This implies that monotonic behaviour, i.e. the strict growth of derived facts in a reasoning system, is not satisfactory. The conclusions in a context of incomplete knowledge are tentative, meaning that they can be withdrawn when new information makes this necessary. The reasoning system has to keep record of all tentative conclusions reached and whether they are believed or disbelieved.

There are quite a few mechanisms that perform non-monotonic reasoning, among them default logics such as established by [Reiter, 1980], non-monotonic logics of e.g. [McDermott & Doyle, 1980], methods such as circumscription [McCarthy, 1980], and, most notably, truth maintenance systems. Of these, the original TMS [Doyle, 1979] has been extended with assumptions by [DeKleer, 1986a] to what is generally known as the ATMS, the assumption-based truth maintenance system. The latter has since gained a widespread recognition and is in our view the best instrument for non-monotonic reasoning, due to the appealing representation of assumptions and the implicit ability of representing multiple contexts, entailing the representation of complex hypotheses. For an overview of non-monotonic reasoning we refer to [Ginsberg, 1987], a review of the literature on truth maintenance is contained in [Stakenborg, 1989], extensions of the ATMS are given by DeKleer in [DeKleer, 1986b; DeKleer, 1986c].

An assumption-based truth maintenance system (ATMS) is meant to cooperate with a problem solver. The problem solver gives the ATMS "inferences". The ATMS in turn gives the problem solver "beliefs". The ATMS is a cache for all the inferences, made by the problem solver. It also allows for non-monotonic reasoning, and it ensures that the database is free of contradictions. Figure 2.3 illustrates the basic architecture. We note that the terms "reason maintenance" or "belief revision" are also used; we adhere to truth maintenance.

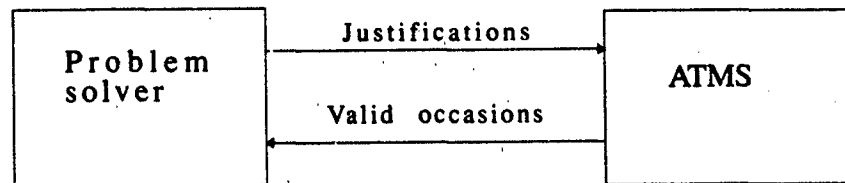


Figure 2.3: Basic problem-solving architecture

### 2.3.2 The Assumption-based Truth Maintenance System

The idea behind an ATMS is the maintenance of a tree-structure of which the nodes are the statements that a problem solving program uses. These statements may be premises, assumptions, i.e. statements assumed to be true by default in order to engage in predictive reasoning, or derived nodes. The results of the problem solver are passed as "justifications" to the ATMS, which records these along with the assumptions they rely on. When new information points out that an assumption is no longer valid, this is passed to the ATMS as well, which then proceeds to track all justifications that used the assumption as an antecedent, and records these as being no longer valid. Implicitly, the ATMS contains all possible hypotheses derivable from combinations of all the valid assumptions.

The ATMS constructs a dependency network consisting of *nodes* and *justifications*. The problem solver associates a datum (a fact, complex proposition, etc.; but more importantly an instantiation) with the node, the ATMS maintains the belief status of the node. An *assumption* results in an *assumed node*, a special type of node that normally remains unjustified (justifies itself), a *premise* is a node corresponding to a basic fact. Other node types are based on justifications representing the derivations made by the problem solver, describing how *derived nodes* depend on other nodes. The dependency network is maintained by the construction of *environments*, being sets of assumptions. Environments are internal representations created by the ATMS, and nodes receive a

*label* as a pointer to the environments the node is valid in, i.e. the assumptions the node is dependent on.

Justifications are expressed in the form:  $A_1, \dots, A_n \rightarrow C$ , with  $C$  the consequent and  $A_1$  to  $A_n$  the antecedents, which can be assumptions or non-assumption nodes. We further define a node as valid in an environment  $E$  if and only if it can be derived from the current set of justifications using only the assumptions in  $E$ . An environment is called *inconsistent* if a contradiction is derivable from it. In the ATMS this is called a *nogood* environment.

We will illustrate the working of the ATMS with an example. We have used an implementation of the ATMS in Prolog listed in [Guoxing, 1989], which was developed at the University of Twente, the Netherlands. The essential source code pertaining to the ATMS algorithm is listed in Appendix B, as well as adaptations we have made to the code to eliminate errors and to enable the interaction with the temporal database. The example below does not incorporate the interaction with the temporal database, this will be addressed in the next chapter. The same example will be used, however, therefore the treatment here is kept concise.

We have several (abstract) nodes, corresponding with the four occasions *occ1* to *occ4*. We add the following to the ATMS, meaning that *occ1* is a premise, *occ2* an assumption, *occ3* derived from *occ1* and *occ2*, and *occ4* is in turn derived from *occ3*:

```
l ?- add_premise(occ1),  
    add_assumption(occ2),  
    add_justification(occ3, [occ1, occ2]),  
    add_justification(occ4, [occ3]).
```

This results in the creation of the premise, assumption, and the two justifications in the ATMS. The ATMS constructs environments as well, and attributes labels to the nodes. The nodes, justifications and environments can be listed. We will illustrate the propagation of a *nogood* assumption through the ATMS with the following simple example. We pass the node *occ2* as a *nogood* assumption to the ATMS. This results in the passing back of a list of the nodes set to *out* by the ATMS, a feature which we have added to the implementation of [Guoxing, 1989].

The result is the following:

```
| ?- pass_nogood(occ2,List).
```

```
Setting nogood: occ2
```

```
Nodes set to out by ATMS:
```

```
List = [occ2,occ3,occ4].
```

This shows that the nodes occ3 and occ4 are set to *out* as well as a result of passing occ2 as a nogood assumption. The result for the environment consisting of the assumption occ2 is that the environment is set to be contradictory by the addition of the node "Contradiction".

Each environment induces a *context*, being sets of nodes (assumptions and non-assumptions) valid in the environment. Thus a context consists of the assumptions valid in a consistent environment, and all nodes derivable from those assumptions. A characterizing environment for a context is the set of assumptions from which every node of the context can be derived. Now, the main task of the ATMS is to determine whether or not a node is valid in a given context. This is managed by the maintenance of labels.

Labelling is a crucial mechanism in an ATMS. A label is a set of environments  $\{E_1, \dots, E_k\}$  associated with each node  $N$ . A label has to fulfil four requirements: it must be *consistent*, *sound*, *complete* and *minimal*. Consistency means that no  $E_i$  is a nogood environment, implying that all environments containing a nogood environment as a subset must be removed. Soundness means that node  $N$  is valid in each  $E_i$ . Completeness implies that every environment  $E$  in which  $N$  is valid is a superset of some  $E_i$ . Finally, minimality is the property that no  $E_i$  is a subset of any other, which implies that all environments that are subsumed by (are supersets of) other environments must be removed.

Figure 2.4 is an example of an environment lattice, the result of only five assumptions. An environment lattice contains all the environments in the ATMS, with the empty node as root node, the assumptions in the ATMS as the next layer, and all possible combinations of these assumptions in the intermediate layers, up to the top node which consists of all assumptions. Given  $n$  assumptions, there are  $2^n$  environments.

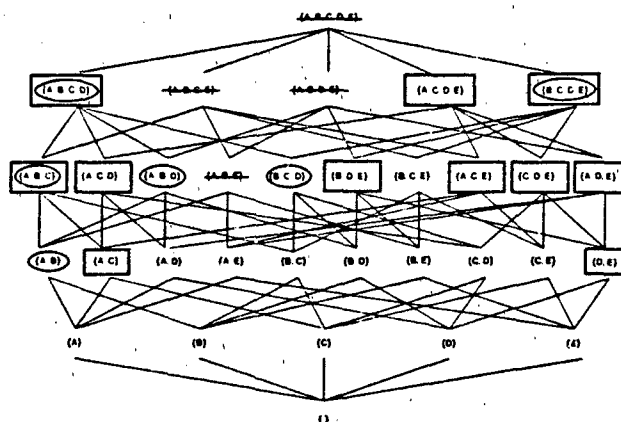


Figure 2.4: Environment lattice [DeKleer, 1986a]

However, not all of these environments will be used for derivations, and the passing of nogood assumptions causes all supersets of these environments to be removed from the lattice. This implies a lesser complexity when using an ATMS than might be expected when considering the complexity of the environment lattices.

The crossed out environments in the lattice of figure 2.4 correspond with the result of passing the environment  $\{A,B,E\}$  as a nogood environment. The environments that have been crossed out are all supersets of environment  $\{A,B,E\}$ . They necessarily become nogood as well because they completely *subsume* the latter environment. As stated above, each environment induces a context, the nodes that are contained in or can be derived from the assumptions contained in that environment. The oval nodes in figure 2.4 represent the context environments of an ATMS node with label  $\{(A,B),\{B,C,D\}\}$ . This means that a node that is valid in the environment with that label will also be valid in all environments that are supersets of the two environments contained in the label. In the same way, the square nodes represent the context environments of a node with  $\{(A,C),\{D,E\}\}$  as label.

The basic ATMS-cycle is the creation of a new node that initially gets the empty label {}. A problem solver datum is associated with the node, and the status is set to *out*. The next step is to either create a premise, create an assumption, or to add a justification. A premise node keeps label {}, but the status is set to *in*. An assumption A is given label {[A]} and status *in*, and is in fact a justification with itself as antecedent *and* consequent. As stated before, the antecedents of a justification must be either premises, assumptions, or other previously justified nodes.

Thus, the ATMS's primary objective is finding a consistent and well-founded assignment of the states (*in* or *out*) of the nodes which are neither premises nor assumptions. The addition or removal of a justification triggers a recomputation of the environment labels of some subset of the set of nodes. It is not our intention to go into the further technicalities of the reason maintenance mechanism here, the algorithm is described in [DeKleer, 1986a].



## 3 Prototyping Mefisto

### 3.1 Introduction

This chapter deals with a demonstration program we have developed that uses a combination of assumption-based truth maintenance, a temporal reasoning mechanism and a forward-chaining rule-firing strategy to fuse information contained in sample reports pertaining to a battlefield situation. The name "Mefisto" is an acronym for "Modular Environment for Fusion and Interpretation of Sensor data in Tracking Opposing forces". The combination of the above techniques is demonstrated to be a powerful tool in addressing some major problems in data fusion: the incorporation of temporal reasoning facilities, a retroactive adaptation of the (temporal) database and the maintenance of concurrent hypotheses concerning the current battlefield situation.

We stress that the intention was to investigate the application of the above techniques to the domain of data fusion. This implies that assumptions have been made beforehand that restrict the notion of "data fusion" in the context of the application described in this chapter. The domain is hypothetical in the sense that it may not be compared to the existing domain that an intelligence officer has to deal with. The contents of the sample reports must therefore not be considered as relevant, they were constructed solely as a means of illustrating the techniques involved. We have not incorporated geographical aspects. We have furthermore restricted the number of sensors to two, being human observers reporting groups of sighted vehicles and secondly electronic warfare units reporting radio-transmittals perceived. The examples used will therefore be admittedly simple, but suffice to illustrate the strength of the combination of the above techniques.

We acknowledge the use of an implementation of the basic ATMS [DeKleer, 1986] developed at the University of Twente, The Netherlands, listed in [Guoxing, 1989], which we have enhanced with temporal reasoning facilities. The data structure used as the foundation for the temporal database stems from [Dean & McDermott, 1987]. We have implemented a basic forward chaining production system as described in [Merritt, 1989]. As stated, the domain is hypothetical, but as guidelines we have used [VS 2-1351, 1988] for a simple order of battle, as well as [VS 30-5, 1989], describing combat intelligence. Structured Analysis (Yourdon) was used for the

representation of the process structure. Nijssen's Information Analysis Method (NIAM) was used for the representation of the data structures. The application was developed on a SUN Sparc-2 workstation and written in Quintus Prolog.

We will first address the domain involved. Then we will provide an overview of the application, describing its structure and specifying the user interface. Paragraph 3.4 will detail the techniques used: the forward chaining algorithm and the rule format, the ATMS and the temporal reasoning facilities that have been constructed, the merger of the latter two, and the data fusion strategy.

## 3.2 The battlefield environment

### 3.2.1 The area of intelligence responsibility

The battlefield environment in our application is, as indicated in the previous section, kept relatively simple. The area of intelligence responsibility is assumed to be a rectangular area. We have divided this region into six sectors, each having the same dimensions located some distance ahead of the FLOT (Front Line Own Troops). In each of these sectors we have a human observer post at a fixed location, the central point of the sector. These observer posts (H1 to H6) report vehicles sighted passing in the vicinity of their locations. The reports from the human observers contain the position (being the location of the observer post), the time of sighting, the vehicle types and quantities sighted and the direction of movement.

Apart from the human observer posts there are two comint posts at the own side of the FLOT. These comint posts (part of an EWU) are assumed to provide detailed information on the front sectors, using ESM systems for the interception of transmissions and taking bearings to determine the positions. They are able to provide order of battle information based on the patterns perceived in the intercepted transmissions. This will lead to the inclusion of either approximate vehicle groups or a reference to a unit type in the comint reports. As a result the information reported by the comint posts is less reliable than the information reported by the human observers. Beside the position the comint reports contain the time of interception, the vehicle types and quantities or a reference to a unit type.

We have limited the spatial aspects in the application to a minimum. We have incorporated several time-dependent context parameters to describe a simplified meteorological condition. This

entails a distinction between day and night, between clear or misty circumstances, and specifically for the comint reports the circumstance that the interceptions of transmissions may have been subject to jamming, possibly causing errors in content and reliability of the comint reports.

### 3.2.2 The order of battle

The basis for the order of battle is taken from [VS 30-1, 1987], however we have adapted this to simplify matters, therefore the order of battle used should not be matched with an existing one. The top level unit is the regiment. The regiments can be either a tank regiment or a mechanized infantry regiment.

---

```
Tkreg
  3 Tkbat
    3 Tkcmp
      3 Tkplt
        3 T80
  1 Mechinfbat_BMP
    3 Mechinfcmp_BMP
      3 Mechinfplt_BMP
        4 BMP
  1 AAAsect
    4 ZSU
  1 Minesweepcmp
    3 Minesweepplt
      10 KMT
```

---

Figure 3.1: A tank regiment

The unit types are tank (Tk), mechanized infantry (Mechinf\_BTR, Mechinf\_BMP), anti-tank (AT), mine sweeping (MS), artillery (ARTY) and anti air artillery (AAA). As vehicle types we have tank (T80), armoured cars (BMP, BTR), anti-tank (SA), anti air artillery vehicles (ZSU), mine sweeping vehicles (KMT), trucks carrying equipment for target acquisition (POLEDISH) and heavy artillery vehicles (2S3).

The mechanized infantry units can be "mechinf\_BMP" or "mechinf\_BTR", depending on the vehicle type. The main force of the regiments consists of four battalions. A battalion consists of several companies and platoons. We assume a pre-combat situation some distance ahead of the FLOT, implying that companies will generally move in a column formation, with a short distance between the vehicles. Generally, the companies of the battalions will be spread out across the

breath of the sectors, the companies move in columns of platoons, allowing a fast deployment if necessary.

A tank regiment has three tank battalions and one "mechinf\_BMP" battalion. Also, a mine sweeping platoon is added to the regiment. This is illustrated in figure 3.1. The numbers listed in the figure are not totals, they should be read as follows: a tank regiment has three tank battalions. Each tank battalion consists of three tank companies. Each company in turn consists of three platoons. Finally, a tank platoon consists of three T80's.

---

```

Mechinfreg_BMP
  3 Mechinfbat_BMP
    3 Mechincmp_BMP
      3 Mechinfplt_BMP
        4 BMP
  1 Tkbat
    3 Tkcmp
      3 Tkplt
        3 T80
  1 AntiTkcmp
    9 SA
  1 ARTYsect
    1 Targetacqplt
      3 POLEDISH
    1 Mechartcmp
      9 2S3
  1 Minesweepplt
    10 KMT

```

---

Figure 3.2: A mechanized infantry BMP regiment

There are two types of mechanized infantry regiment, corresponding with the vehicle types BMP and BTR. The mechinf\_BMP regiment consists of three mechinf\_BMP battalions and one tank battalion. Beside these battalions the regiment consists of an anti-tank company, a mine sweeping platoon and an artillery section with target acquisition means and heavy artillery. The mechinf\_BTR regiment consists of three mechinf\_BTR battalions and one tank battalion. It also has an anti-tank company, an anti air artillery section and a mine sweeping platoon.

---

```
Mechinfreg_BTR
  3 Mechinfbat_BTR
    3 Mechinfcmp_BTR
      3 Mechinfplt_BTR
        4 BTR
  1 Tkbat
    3 Tkcmp
      3 Tkplt
        3 T80
  1 AntiTkcmp
    9 SA
  1 AAAsect
    4 ZSU
  1 Minesweepplt
    10 KMT
```

---

Figure 3.3: A mechanized infantry BTR regiment

### 3.3 System overview

#### 3.3.1 The structure of the process

The overall structure of the application is as follows: we have a domain representing a simple battlefield as explained in the previous section, with sources sending in reports on sighted vehicle groups and perceived radio-transmissions. The information in the reports is converted into unit structures. By default each report is first assumed to refer to a separate unit on the battlefield. The further structuring of the units is accomplished by classifying, correlating and aggregating units by the application of rules containing knowledge of the properties of the domain objects and the order of battle. The derivations and the underlying assumptions concerning these units and their attributes are passed to the ATMS. The temporal information concerning the units is passed to a temporal database. The rules use the combination of information contained in the ATMS and the temporal database. The top-level process structure consists of the following functions:

1. Process report;
2. Classify unit;
3. Correlate units;
4. Aggregate units;
5. Process ATMS-request;
6. Process TDB-request.

The functions are carried out in response to events, being requests from the operator of the application. The processing of a report is the conversion of a report into a unit (the assumption that each report refers to a separate unit), for which an initial "unit frame" is constructed with attributes filled in as much as possible from the reported information. The report is stored as a premise in the ATMS, a timedist entry for the report is stored in the temporal database (TDB). The unit that is created is stored in the ATMS as an assumption. Two timedist entries are stored in the TDB (see paragraph 2.2.3), one from the reference point to the begintime (the sighting time reported), and one from the begintime to the endtime, whereby the upper bound of the duration is initially set to infinity.

The three main phases are the classification, correlation and the aggregation of units. The classification is aimed at determining the unit type. A classify request triggers the attempt to classify a certain vehicle type group as referring to a specific unit type by the application of classification rules. The result is stored as a justification in the ATMS and accompanying timedist entries are stored in the TDB. Among the classification rules are default rules, which also derive a unit type, but the latter is stored as an assumption in the ATMS. The assumed unit type will possibly be used later to correlate and/or aggregate. When new information indicates that the unit is not a tank company but a mechanized infantry company, this will cause a recalculation of the ATMS states corresponding to the validity of derived statements based on the previous assumption.

The correlation of reports with units is more complicated. Given two observer posts at some distance from each other, it is possible that the same vehicle groups pass both observer posts. In the case of information from electronic warfare units this may hold as well because the comint posts may report the same group more than once. The correlation is aimed at eliminating duplicate counts of the same units, which would indicate a stronger force than would actually be present on the battlefield. This phase thus encompasses the correlation of data from like sensors as well as from different sensors. Correlation rules aid in determining whether two reports in fact refer to the same object. When this occurs the unit structure corresponding to the earlier report is clipped, passed to the ATMS as a nogood assumption, triggering the recalculation of the consistency of the database.

Aggregation is the combination of two or more units into a higher-level unit. When an aggregation takes place, the resulting unit is stored separately, leaving the units used for the

aggregation intact because there may be several options for the aggregation of units, and units might be correlated later with other units. Aggregation rules also perform a classification task because they will derive values for the unit type of the resulting aggregation.

We stress that the control of the application is to a large extent in the hands of the operator. We have chosen to allow the phases classification, correlation and aggregation to be initiated by the operator and not by the application itself. This implies that the application is not highly "automated". The reason is the focus on techniques for truth maintenance and temporal reasoning; the aim was not to process incoming reports in real time. To further support the interaction with the ATMS and the TDB a direct manipulation of the ATMS-database and the temporal database is possible as well. The processing of requests to the ATMS and the TDB are contained in the two remaining functions in the aforementioned list. In paragraph 3.3.1 this will be dealt with in more detail.

### 3.3.2 The structure of the data

The main data structures in the application are "report", "unit\_type" and "unit". The ATMS further uses "tms\_node", "assumption", "justification" and "environment" as data structures. The temporal database is built with "timedist" as data structure. The data structures will be outlined below.

The *report* contains a report number as the unique reference, and the source of the report. In the case of a human observer post, the location of the sighted vehicles and the direction they were moving in is reported. The vehicles are reported as "sighted\_vehicle\_type\_group", represented in the database as tuples of vehicle types and quantities. In the case of a comint report, the location and orbit information are reported. In both cases the time of sighting is contained in the report.

---

```
Report_nr      : 1
Source_type    : humint
Source_nr      : 3
Direction_from : east
Direction_to   : west
Sighting_time  : 101
Vehicle_group  : t80 , 9
```

---

Figure 3.4: Report structure

The "unit\_type" is the representation of units contained in the standard order of battle. A "unit\_type" is represented as a unit class and unit size, e.g. respectively "tank" and "company", implying that the unit type is a tank company. This distinction is made because it may for example be evident what class a unit belongs to, but not what the size of the unit is. The following example clarifies the representation of a "unit\_type":

```
unit_type(mechinf_btr,cmp).  
unit_type(mechinf_btr,plt).  
unit_type_group(mechinf_btr,cmp,mechinf_btr,plt,3).  
orbat_vehicle_type_group(mechinf_btr,plt,btr,4).
```

The "unit\_type\_group" above means that a mechanized BTR infantry company (the notation above is in lower case letters due to a convention in Prolog) consists of 3 mcchinf platoons, these in turn consisting of 4 BTR vehicles. The unit type data is used in the rules. In effect, a match is carried out between the "sighted\_vehicle\_type\_groups" contained in the reports and the "orbat\_vehicle\_type\_groups" of the known order of battle to classify the unit type.

The *unit* is the central data structure, it contains the information on the units that (are assumed to) exist on the battlefield. Apart from information on the source(s) that reported the unit, the (last known) position and direction of movement, a unit consists of vehicle type groups and has a unit class and a unit size. The vehicle type groups may be the result of several sighted vehicle type groups and are therefore distinct from the latter. A rating is attributed to the unit structure, based on several context parameters concerning the time of day, the weather and jamming, as described in paragraph 3.2.1.

---

```
Unit_nr      : 1  
Source_type  : humint  
Source_nr    : 3  
Last_position : [250,250]  
Sector       : 3  
Direction_from : east  
Direction_to  : west  
Vehicle_groups : [t80,9]  
Unit_class   : tk  
Unit_size    : cmp  
Rating       : a
```

---

Figure 3.5: Unit structure



The ATMS uses four main data structures. The following are based on the ATMS-implementation in Prolog by [Guoxing, 1989] that we have used, which will be addressed in paragraph 3.4.2:

```
tms_node (Index, Occasion, Status, Label, Justifications, Consequents,  
          Rules, Nodetype) .  
assumption (Index, Occasion, Environments) .  
justification (Index, Type, Consequent, Antecedents) .  
environment (Index, N_assumptions, Assumptions, Nodes, Contradictory) .
```

The *tms\_node* data structure contains first of all an index and the occasion it corresponds with. Furthermore, the status of the node (*in* or *out*), a label representing the indices of environments (sets of assumptions) in which the node can be proven using all justifications known to the ATMS, an index of the justification(s) describing how this node is derivable from other nodes, an index for the justifications using this node as an antecedent, a Rules field reserved for the problem solver and finally the type of node (assumed node, premise, or derived node).

The *assumption* data structure has the occasion as second argument, and provides a list of the environments wherein the assumption holds.

The *justification* data structure lists the type of justification (supplied by the problem solver), the index of the consequent node and the indices of the antecedent nodes.

The data structure *environment* includes the number of assumptions in this environment, the (indices of the) assumptions themselves, the nodes in whose label the environment appears, and the field contradictory indicates whether this environment is inconsistent.

We have used a temporal data structure that we call *timedist* that originates from [Dean & McDermott, 1987] as the data structure for the temporal database. The data structure contains the names of the begin- and the endpoint and the lower and upper bounds of the duration between the begin- and endpoint:

```
timedist (Beginpoint, Endpoint, Low, High) .
```

The temporal representation implies that nothing is deleted from the database. All data stored in the ATMS and in the internal (Prolog) database receives a temporal entry. In the case that an occasion is no longer valid, the temporal entry is clipped, but the occasion as well as the clipped temporal entry are retained.

### 3.3.3 The structure of the dialogue

In its current form, the application consists of a main program containing the forward-chaining engine, predicates governing temporal reasoning, the interaction with the ATMS and the TDB, and the definition of the interface. Furthermore, there are separate files containing the ATMS, a compiled report database, a compiled domain database and a compiled rules knowledge base. The Prolog database is used as the working storage means. Due to the concentration on techniques involving truth maintenance and temporal reasoning, we have not incorporated any graphical facilities whatsoever.

The control of the application is to a large extent in the hands of the operator. The main menu therefore corresponds with the functions described in paragraph 3.3.1 and has the following form:

1. Report generator
2. Classify report
3. Correlate units
4. Aggregate units
5. ATMS request
6. TDB request
7. Exit

The report generator can be called; it fetches and prints reports one by one:

Generate report? [y/n]: y.

Report\_nr : 5  
Source\_type : humint  
Source\_nr : 3  
Direction\_from: east  
Direction\_to : west  
Sighting\_time : 105  
Vehicle\_group : btr , 3

For each report, a new unit is asserted with as yet no unit class or unit size, but with vehicle type groups filled in if these were reported. Transparent to the user, corresponding occasions and temporal entries are passed to the ATMS and the TDB.

Classification, correlation and aggregation are all initiated by the operator. A request results in the application of rules. The application of the rules is sequential, stepping through the rules in the order in which they are contained in the knowledge base. It is possible to fire applicable rules one at a time or all at once. The inference mechanism and the rule format will be addressed in paragraph 3.4.1. When a rule fires this is communicated to the user, stating the rule that was fired and the interactions with the ATMS and/or the TDB that resulted from it.

Apart from the above a direct interaction with the ATMS and the TDB is possible. The ATMS database can be queried for `trns_nodes`, assumptions, justifications and environments. It is possible to additionally pass instantiations of these structures to the ATMS, providing direct control over the ATMS. The temporal database can be queried and `timedist` instantiations may be asserted. However, the interaction with the ATMS and the TDB will generally be triggered by the application of the rules. The details of the interaction will be further dealt with in paragraphs 3.4.2. and 3.4.3.

We will indicate the flow of control between the main functions. The functions initiated by the main menu can be called independently, however, they do depend on each other. The report generator will generate reports one at a time. A unit frame is constructed for the report and filled in with the report information. The report generator will ask whether more reports are required. In general, when a number of reports have been generated, the functions classification, correlation and aggregation are called sequentially. This is illustrated in figure 3.8 below. From each of these phases the report generator can be called again, after which the above process is repeated.

During the four phases above the ATMS and the TDB are inspected and adapted continuously by the system, triggered by the rules that are fired. However, this inspection and adaptation may take place manually as well.

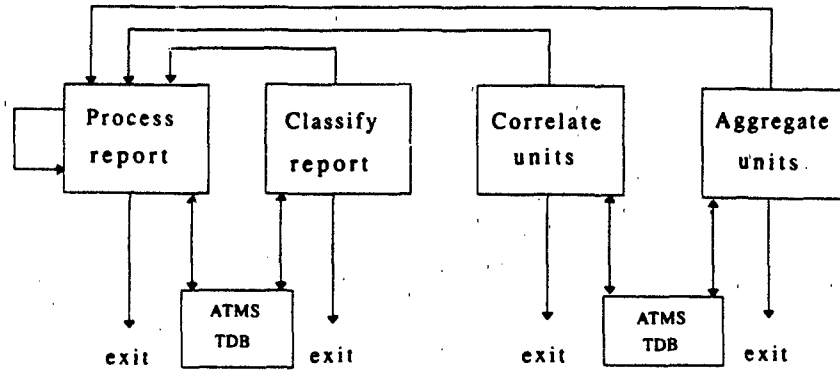


Figure 3.6: Control flow

## 3.4 Technology overview

### 3.4.1 Forward chaining production system

We have used a simple forward chaining production system for the implementation of the rule-application. The algorithm is taken from [Merritt, 1989]. It is a basic failure-driven loop carrying out a "match-and-process" cycle, firing all rule-instantiations, one rule at a time.

We have chosen to use a forward-chaining rule-firing strategy for the fusion of information from the reports. Rule-based reasoning is a widespread technique and it provides a quite natural representation for stating e.g. some simple heuristics pertaining to order of battle. The forward-chaining strategy was also chosen because reports are entered one by one and continuously and the information that feeds the rules becomes available in bits and pieces. This as opposed to for instance the domain of diagnosis, where a set of symptoms is entered at once, and a backward chaining strategy is used that finds the rule(s) containing the diagnosis that best matches the symptoms. Also, the ATMS cannot store variables and thus all ATMS nodes must be instantiations of, in our case, Prolog clauses. This implies that a backward-chaining rule-firing strategy, whereby instantiations are only "virtual" in the sense that they are not explicitly stored, does not suffice.

The rule format that we have used is as follows:

```
rule(Rule_type,Rule_nr,Description)::  
    [Condition1,  
     Condition2, etc]  
==>  
    [Action1,  
     Action2, etc].
```

A simple example of a rule is the following, stating that a unit is classified as being a tank company when the unit consists of between 8 and 13 tanks. The "derive\_and\_assume" statement is a call to the ATMS, triggering the passing of a justification for the unit class and unit size classification to the ATMS.

```
rule(classify(Unit_nr),5,tkcmp)::
    [tms(unit_vehicle_type_group(Unit_nr,t80,Qty)),
     Qty > 8,
     Qty < 13
    ]
==>
    [temporal_distance(ref,end(unit(Unit_nr)),Low,High),
     derive_and_assume(unit_class(Unit_nr,tk),_,Low),
     derive_and_assume(unit_size(Unit_nr,cmp),_,Low)
    ].
```

The rule types are divided into classification, correlation and aggregation rules. The above rule is an example of a classification rule.

The derived unit class and unit size are set to be an assumption in the ATMS after being passed as a derivation. This implies that if at a later time the unit class or unit size of this unit are derived to be something other than "tk" and "cmp", the passing of the assumption as a "nogood" to the ATMS will result in tracking the conclusions that have since made, based in part on the assumption that the unit was a tank company.

The conditions and actions of the rules are contained in lists, enabling efficient processing in Prolog. We distinguish between three types of "calls" in the conditions and actions of the rules. These can be calls to Prolog, calls to the ATMS and calls to the temporal database. These three types of calls are the building blocks of the rules. They are carried out by "match" and "process" clauses, which use predicates corresponding to the calls. The Prolog calls may be calls to functions in the main program, queries, assertions and retractions of clauses in the Prolog database, as well as built-in Prolog predicates. The interaction with the ATMS and the use of the temporal database will be the subject of the next section.

### 3.4.2 Truth maintenance and temporal reasoning in Mefisto

#### 3.4.2.1 Keeping track of reason

We have used an implementation of the ATMS that is listed in [Guoxing, 1989], written in Prolog. De Kleer's original ATMS does not incorporate negations, nor disjunctions, nor does it allow to explicitly contain default justifications. The implementation of the ATMS of [Guoxing, 1989] concerns the basic ATMS and therefore does not include these either. Instead of a time-consuming adaptation of Guoxing's implementation, we have chosen to bring negations, disjunctions and defaults into the interaction with the ATMS triggered by the application of the rules, letting the problem solver deal with them. An adaptation of Guoxing's implementation was necessary, however, to remove several errors mainly concerning the subsumption of environments and the allocation of contradictory environments as a result of passing nogood assumptions. Further adaptations concerned the interaction with the main program and the temporal database. Details can be found in the appendices.

The consultation and manipulation of the ATMS is triggered by the application of the rules; the ATMS-calls as explained in the previous paragraph. The ATMS is queried in the conditions of the rules. For instance, in the case of the following condition, used in the example of the previous paragraph:

```
tms(unit_vehicle_type_group(Unit_nr,t80,Qty)).
```

The possible ATMS-calls are the following:

```
asktms(X, Status).  
add_premise(X).  
add_assumption(X).  
add_justification(Consequent, AntecedentList).  
pass_nogood(X).
```

An occasion can be asserted as a premise. A new assumption can be created resulting in the assertion of an assumed node in the ATMS, which also triggers the creation of a new environment. Justifications pass derived nodes and their antecedents in the ATMS. When passing an assumption or set of assumptions as a nogood environment in the ATMS, the environment is

set to "contradictory", triggering a recomputation of the labels. The ATMS checks which nodes have labels containing environments that are subsumed by this nogood environment. It changes these environments to "contradictory" as well and adjusts the labels, so that these nogoods are removed from the respective labels. If as a result the label becomes empty, the status of that node is set to *out*, the node is no longer valid. This process results in an "outlist", i.e. a list of the nodes that the ATMS has set to *out*.

#### 3.4.2.2 Keeping up with time

Each occasion has a *timedist* entry representing the duration of the occasion. We repeat the basic temporal data structure here:

```
timedist (begin (Occasion) , end (Occasion) , Low , High) .
```

This represents that the duration of Occasion is known to have a lower bound Low and an upper bound High. This is a relative duration, not related to an absolute reference point. In the case that the begintime of the occasion is known as well, another *timedist* entry is asserted into the temporal database to allow the calculation of absolute time. We have chosen this absolute reference point (*ref*) to be the timepoint 0. This second *timedist* entry represents the begintime of the occasion and will have both lower and upper bound equal to the begintime of the occasion. For a persistence, the *endtime* in the above *timedist* entry is set to infinity, indicating that the occasion is believed to be valid indefinitely, to be defied only by information stating the contrary.

The interaction in the rules allows to query the temporal database, calling functions that calculate temporal distances, asserting, updating and clipping *timedist* entries. The following calls to the temporal database are possible:

```
timedist (X, Y, Low, High) .  
add_timedist (X, Y, Low, High) .  
update_timedist (X, Y, Low, High) .  
clip_node (A, T) .  
clip_nodelist (List, T) .
```



The *clipping* mechanism changes persistences into clipped persistences (clips) when called for. This clipping is essentially the replacement of the upper bound of the occasion to be clipped. Due to the relative notions involved in the *timedist* entries the new upper bound will be the difference between the *cliptime* and the *begintime* of the occasion. Let's take the following example:

```
timedist (ref, begin (unit (1)), 120, 120) .  
timedist (begin (unit (1)), end (unit (1)), 0, inf) .
```

When the occasion *unit(1)* is clipped at time 200, the result of the clipping will be that the first *timedist* entry is kept intact and that the second *timedist* entry is updated to the following:

```
timedist (begin (unit (1)), end (unit (1)), 80, 80) .
```

When querying the temporal distance from the reference point *ref* to *end(unit(1))* the result will now be 200.

We have furthermore implemented an elaborate collection of temporal queries. The queries enable to determine whether a certain timepoint comes before another timepoint, whether an occasion is true throughout a specific interval, whether two occasions are overlapping, etc. We refer to the appendices for a more complete list.

### 3.4.3 Inference and temporal truth maintenanc

Given techniques for assumption-based truth maintenance and temporal reasoning, the question now arises how to combine them. Both the temporal data structure and the representation of an occasion in an ATMS contain information concerning the validity of an occasion. The ATMS is used to keep track of assumptions and conclusions based on these assumptions. The temporal database must be updated when the ATMS has defied earlier derived conclusions. The ATMS *in/out* truth status represents the current status of an occasion in the sense that it happens to be the result of the latest adaptation to the ATMS, but it is in fact time-independent. The temporal database represents the complete period during which the occasion is valid. The combination is comparable to a four-dimensional space-time representation, in the sense that it allows to represent the state of the battlefield (which is described by the occasions) at each moment in time.

Moreover, it does not represent just one state, but implicitly contains a multitude of states due to the representation based on assumptions.

The effectuation of the combination of truth maintenance and a temporal database is triggered by the application of the rules. Adaptations to the implementation of the ATMS of [Guoxing, 1989] were made to effect the passing of information to and from the ATMS, necessary to be able to adapt the timedist entries when an environment is passed as "nogood" to the ATMS. When an occasion is passed to the ATMS, at least one timedist entry is asserted to the temporal database, as outlined in previous paragraphs. When an assumption or set of assumptions is passed as a nogood environment to the ATMS the timedist entries for the assumption(s) are adapted simultaneously, clipped by some cliptime. The ATMS returns a list of the nodes invalidated as a result of the ensuing label recomputation. These nodes can then be clipped accordingly.

To accomplish the above we have implemented a set of predicates that combine the functions listed in the previous two paragraphs. These are used in the actions of the rules and are the following, the relevant source code for the predicates listed below is contained in Appendix A:

```
asktms (X, Status) .
add_premise (X, T) .
assume (X, T) .
derive (X, List, T) .
derive_and_assume (X, List, T) .
replace_node (X, Y) .
set_node (X, Status) .
add_timedist (X, Y, Low, High) .
update_timedist (X, Y, Low, High) .
clip_node (X, T) .
clip_nodelist (List, T) .
pass_nogood (X, OutList) .
set_nogood (X, T) .
```

These perform selections, additions, updates and "deletions", the latter in effect the replacement of (truth) status *in* by status *out*. A derived occasion results in passing a justification to the ATMS with the nodes, that were used as conditions in the rule, as the antecedents of this justification. It

is also possible that a derived occasion is set to be an assumption as well, a feature intended to allow the tracking of how defaults were derived (e.g. when assuming unit types) which is not incorporated in the basic ATMS. A node can be replaced, or set to another truth status. The *set nogood* predicate is a combination of "pass nogood", "clip node" and "clip nodelist". The clipping of a nogood assumption results in the passing back of a list of the nodes set to *out*, the "clip nodelist" predicate carries out the clipping of this list.

We will illustrate the above for the example that was already contained in chapter 2 for the four occasions *occ1* to *occ4*. We instantiate *occ1* as a premise from timepoint 0 on, and *occ2* as an assumption from timepoint 100 on:

```
| ?- add_premise(occ1,0).  
| ?- add_assumption(occ2,100).
```

The ATMS now contains two nodes, a justification for the assumption *occ2*, as well as an environment consisting of the assumption. Now, we will apply two rules, resulting in the derivation of occasions *occ3* and *occ4*. The occasion *occ3* is for example the result of firing the first rule below. The rule states that *occ3* is derived if *occ1* and *occ2* are valid in the ATMS. The *derive* statement adds *occ3* as a derived node to the ATMS, and simultaneously adds *occ3* to the temporal database, with a begintime corresponding to the later of the begintimes of *occ1* and *occ2* (application of the rule will find this to be the timepoint 100).

```
rule(example,1,occ3)::  
    [tms(occ1),  
     tms(occ2)]  
    ==>  
    [later(occ1,occ2,Time),  
     derive(occ3,_,Time)]
```

The second rule results in the derivation of *occ4*. It is instantiated in the ATMS as a derived node, derived from the validity of the earlier derived *occ3*. As the begintime for *occ4* the begintime of *occ3* is taken, the latter being calculated by the temporal distance clause contained in the rule.

```
rule(example,2,occ4)::
  [tms(occ3)]
  ==>
  [temporal_distance(ref,end(occ3),Time,_),
   derive(occ4,_,Time)]
```

The following is the result of firing the rules:

```
=> fire(example,_).
```

```
Adding justification for: occ3
```

```
Starting at time: 100
```

```
Rule fired: example, 1, occ3
```

```
Fire rule? y.
```

```
Adding justification for: occ4
```

```
Starting at time: 100
```

```
Rule fired: example, 2, occ4
```

The ATMS can be consulted for lists of the nodes, justifications and environments contained in the ATMS. When we inspect the justifications the ATMS has constructed due to the firing of the two rules above, the result is the following:

```
Justification: 2 -> Type: example
```

```
Consequent   : 3
```

```
Antecedents  : [1,2]
```

```
Justification: 3 -> Type: example
```

```
Consequent   : 4
```

```
Antecedents  : [3]
```

The first justification listed means that a justification has been added with "Type" indicating the rule type that caused the justification. The consequent of the justification is node 3, the

antecedents are nodes 1 and 2. For this example, the numbers of the nodes correspond with the numbers of the occasions, as the list below will show. This also illustrates the "bookkeeping" nature of the ATMS; the derivations are registered without any reference to the meaning of the nodes involved. The second justification above corresponds to the derivation resulting from the second rule.

Now, when we inspect the ATMS for the nodes with truth status *in*, the following list is generated:

```
! ?- tell_node(_,in).
Node 1:  occ1, premise node
Node 2:  occ2, assumed node
Node 3:  occ3, derived node
Node 4:  occ4, derived node
```

The internal representation of these nodes in the ATMS further contains the label, being the set of environments that the node is valid in. For example, we will list the representation for node 3 here:

```
Node   : 3 -> Datum: occ3
Status: in
Label  : [1]
Just   : [2]
Cons   : [3]
Type   : [derived]
```

This implies that occ3 is contained in the ATMS with environment 1 as label. In our example, there is only one environment, containing the assumption occ2. The field "Just" contains justification 2 as the justification that instantiated occ3, and the field "Cons" states that justification 3 contains node 3 as an antecedent.

The clipping of nodes is also instigated by the application of rules. We will use the following clipping rule which does not contain any conditions, as these are not relevant here. The condition would be some state that would lead to the clipping of the assumption occ2 at timepoint 190.

The rule is as follows:

```
rule(clipping,1,occ2)::  
  []  
  ==>  
  [set_nogood(occ2,190)].
```

```
| ?- fire(clipping,1).
```

Setting nogood: occ2

Nodes set to out by ATMS are:

occ2, clipped at time 190

occ3, clipped at time 190

occ4, clipped at time 190

Rule fired: clipping, 1, occ2

The nodes corresponding with occ2, oc3 and occ4 are set to *out* in the ATMS, environment 1 above is set to "contradictory", and a justification is added for the nogood assumption added to the ATMS. When we query the absolute temporal bounds for the occasions the result is the following:

```
| ?- time(_).  
[occ1,0,inf]  
[occ2,100,190]  
[occ3,100,190]  
[occ4,100,190]
```

When we ask which nodes have truth status *in*, the ATMS responds as follows:

```
| ?- tell_node(_,in).  
Node 1:  occ1, premise node
```

An issue is what to pass to the ATMS. If all data that is usually stored in databases is now stored in an ATMS, not considering their meaning, the overhead might not be worthwhile. Moreover, the

nature of an ATMS is that it performs bookkeeping, it keeps track of *how* the nodes were derived and thus *why* something is valid or not. The problem solver keeps track of *what* it means and *when* it is valid. The representation of semantical content in an ATMS is therefore a waste of resources. For this reason the timedist entries are kept outside the ATMS. Secondly, the timedist entries implicitly represent the validity of occasions and storing them in the ATMS database would entail redundancy, leading to integrity problems. The ATMS needs only explicitly contain the occasions as nodes. In this way, the ATMS keeps track of the occasions, but the history of the occasion is not represented in the ATMS.

#### 3.4.4 Data fusion in Mefisto

In paragraph 3.3 the main functions, data structures and flow of control in Mefisto were outlined. Here we will indicate step by step what actions can be taken in the application to perform the fusion of reported information.

The strategy for performing data fusion is governed by the four main phases that we will repeat here:

1. Generation of reports;
2. Classify unit types from report information;
3. Correlate (seemingly) identical units;
4. Aggregate units into higher level unit structures.

The first phase is the generation of reports. When the report generator is called to generate a report the first action taken is the construction of a unit structure as described in paragraph 3.3.2. This unit structure is filled in as much as possible and a reference to the report is made to enable tracing the reported information. We stress that this is an intermediate structure, resulting from the underlying assumption that each report refers to a separate unit on the battlefield. These unit structures may be altered or eliminated in the later correlation phase.

The unit structure contains the following separate elements:

```
unit(Unit_nr).  
origin(Unit_nr,Source_type,Source_nr,Report_nr).  
last_known_movement(Unit_nr,Position,Sector,Direction).  
unit_vehicle_type_group(Unit_nr,Vehicle_type,Quantity).  
unit_class(Unit_nr,Unit_class).  
unit_size(Unit_nr,Unit_size).  
rating(Unit_nr,Rating).
```

The "unit\_vehicle\_type\_group" may be repeated if the unit consists of more than one vehicle type group. An occasion for the report is passed to the ATMS as a premise, a timedist entry for the report (sighting time) is asserted to the temporal database. An occasion for unit(Unit\_nr) is passed to the ATMS as an assumption. The unit class and unit size (together: the unit type) will not be filled in yet. The unit vehicle type group(s) is (are) adopted from the report. The unit vehicle type groups are passed to the ATMS as assumptions as well. The reason is that the reported sightings are not at first hand assumed to be completely reliable. The occasions are passed to the temporal database as persistences. This implies that two timedist entries are asserted to the temporal database. The first registers the begintime of the occasion, the second registers the duration of the occasion with the upper bound set to infinity, as explained in paragraph 3.4.2.

The second phase is the classification of unit types. The unit structures resulting from the reports must now be attributed with a unit class and a unit size. This is accomplished by the application of classification rules. These classification rules contain knowledge on the order of battle, starting at the vehicle level. For instance, a report from a human observer of a column of ten tanks will be classified as referring to a tank company. Classification will often take place on the basis that a specific vehicle type or group of vehicle types is characteristic for a certain unit type. Also, the absence of a certain vehicle type may indicate values for unit class and possibly size. The occasions for the unit class and size are passed as "derived assumptions" to the ATMS, allowing that they may be set to nogood at a later time. The timedist entries for unit class and size will be adopted from the timedist entries of the unit structures.

The third phase is correlation. The aim is to distinguish references to the same battlefield unit by various reports and to unite the reported information into one unit structure. Here, the



intermediate unit structures resulting from the report generation are inspected and duplicate references to the same battlefield objects are eliminated, if possible. Correlation therefore requires criteria to eliminate duplicate counts. The criteria are applied by correlation rules.

Important is that criteria are used to perform the correlation of like-sensor data as well as for the correlation of data from different sensor types. In our application, this means both data reported by human observers (humint) and data from electronic warfare units (comint). The strategy we have chosen for correlation is that we correlate like-sensor data first, data from different sensor-types after that. Secondly, we restrict the correlation to be applied pairwise. We will illustrate this for the case that four reports refer to the same battlefield unit. Let's assume that the unit structures resulting from these reports are called  $u_1$  to  $u_4$ . The first three are humint reports, the last one is a comint report. Application of the correlation rules will result in a sequence of three correlations:  $u_1$  with  $u_2$  (result  $u_{12}$ ),  $u_{12}$  with  $u_3$  (result  $u_{123}$ ), and finally  $u_{123}$  with  $u_4$ , resulting in  $u_{1234}$ . The strategy of pairwise correlation restricts the complexity in the correlation rules. More importantly, it will also restrict the computational complexity as the number of occasions grows. We stress here again that we use the term correlation for attributing reported information to battlefield objects already sighted, not as the correlation of low level sensor tracks.

The correlation of two unit structures will result in the clipping of the earlier unit structure. The later unit structure (the result of a later report) is kept. Attributes of the later unit structure may be filled in, for example when correlating a humint sighting with an earlier comint report the frequency will be added to the unit structure that resulted from the humint sighting. Also, the vehicle type groups may be adapted to incorporate the information from both reports. Generally, correlation will imply an accumulation of information concerning battlefield objects. In this way, the units "move ahead" on the battlefield, so to speak. The earlier unit structure is passed as a no-good assumption to the ATMS. This may lead to the situation that conclusions based on the existence of this earlier unit are set out by the ATMS. The unit is clipped in the temporal database.

The last phase is the aggregation of units. The units remaining after the correlation phase are input to the aggregation phase. The aggregation is aimed at grouping units into higher level units. Platoons are combined into companies, companies will in turn be aggregated into battalions, battalions into regiments. Aggregated units are represented separately in the ATMS. The reason we keep them apart is that units may be part of more than one aggregation, therefore the single

units remain in the database and are not clipped. The aggregation is a kind of overlay on the units on the battlefield. Aggregations are hypothesized by the application of aggregation rules, and justified in the ATMS with the units (and their specific attributes) grouped together as the underlying assumptions.

Aggregations represent the situation hypotheses. When a new report entails an addition of a unit, this may result in another aggregation. This aggregation may contradict an earlier one, giving rise to a second hypothesis of the perceived situation. Another case is when a newly generated report results in the correlation with some unit structure used in an aggregation. The result may be a change in the classified unit type, and it may well be that what was thought to be the unit type is now passed as a nogood assumption to the ATMS. This may in turn lead to the removal of an aggregation and thus the elimination of a situation hypothesis.

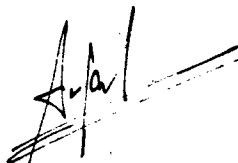
## 4 Conclusions and recommendations

The application described in this chapter is in an experimental stage. It does however demonstrate the usefulness of the combination of a forward chaining rule firing mechanism, an assumption-based truth maintenance system and a temporal database to support a strategy for data fusion. The facilities for predictive reasoning and a retroactive adaptation of the database created with this combination allow to maintain multiple hypotheses concerning a battlefield environment with the maintenance of assumptions as the key point. The temporal database furthermore supports a "non-deletion policy", allowing data to be kept at relative low cost because one timedist entry for each occasion suffices to maintain its history.

An important aspect of using an ATMS is computational efficiency. Especially the mechanism for updating (e.g. making sure that the list of environments connected to some fact is indeed without contradictions) and the process of handling nogoods can become very tricky from an implementation standpoint [Morgue & Chehire, 1991]. Again we stress the fact that we used an implementation of the ATMS by [Guoxing, 1989], we refer to Appendix C. However this did have a drawback, because the implementation needed code adaptations in order to perform adequately. Most notably, the subsumption of a nogood environment by existing environments in the ATMS was not calculated properly, entailing that the effect of passing a nogood did not propagate "far enough" causing occasions to remain valid when they were not supposed to. Apart from remedial work, the adaptation to allow for the interaction with the main program and the temporal database was time-consuming as well.

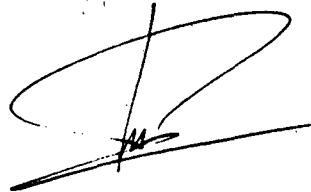
The application and the techniques used could be extended in several ways, we also refer to [Keene & Perre, 1990]. As stated earlier, geographical complexity was not incorporated. Nor have we used any probabilistic techniques to represent uncertainty. This could be done by using for instance the Dempster-Shafer theory of evidence as is described in [Brogi et al., 1984]. Incorporation of fuzzy logic techniques could be even more promising [Sombé, 1990]. The forward chaining engine could be enhanced with certainty factors as well, which would be a relatively simple extension. The rule-firing strategy could be extended, for instance by using the RETE match algorithm to perform conflict resolution among applicable rules [Merritt, 1989].

---



Ir. A.P. Keene  
(author)

---



Drs. M. Perre  
(project leader)

## Acronyms and abbreviations

ASIC	All Sources Information Center
ATMS	Assumption-based Truth Maintenance System
COMINT	Communications Intelligence
DBMS	Database Management System
DFD	Data Fusion Demonstrator
DFD	Data Flow Diagram
DTG	Date Time Group
EWU	Electronic Warfare Unit
ELINT	Electronic Intelligence
FEBA	Forward Edge of the Battle Area
FLOT	Front Line Own Troops
GLB	Greatest Lower Bound
HUMINT	Human Intelligence
ISD	Information Structure Diagram
LUB	Least Upper Bound
MEFISTO	Modular Environment for Fusion and Interpretation of Sensor data in Tracking Opposing forces
NIAM	Nijssens Information Analysis Method
ORBAT	Order of Battle
POSTGRES	Post Ingres
PROLOG	Programming in Logic
RDBMS	Relational Database Management System
RMS	Reason Maintenance System
RPV	Remotely Piloted Vehicle
TDB	Temporal Database
TMM	Time Map Manager
TMS	Truth Maintenance System

## References

[Bonasso, 1985]

Bonasso,R.P., "Capturing domain primitives for knowledge-engineering", Proceedings IEEE Conference on Circuits, Systems and Computers, 1985.

[Bonasso et al., 1984]

Bonasso,R.P., Antonisse,H.J., Laskowski,S.J., ANALYST I,II, Reports MTP-83W00002, MTP-84W00220, Mitre Corporation, 1984.

[Brogi et al., 1988]

Brogi,A., Filippi,R., Gaspari,M., Turini,F., "An expert system for data fusion based on a blackboard architecture", in: [Engelmore & Morgan, 1988].

[Bratko, 1986]

Bratko,I., *Prolog programming for artificial intelligence*, Addison-Wesley, 1986.

[Byrne et al., 1989]

Byrne,C.D., Miles J.A.H., Lakin,W.L., "Towards knowledge-based naval command systems", in: *Proceedings Third International C3MIS Conference*, IEE, Bournemouth, 1989.

[Dean & McDermott, 1987]

Dean,T.L. & McDermott,D., "Temporal database management", *Artificial Intelligence* 32 (1987), pp. 1-55.

[Dean, 1985]

Dean,T.L., *Temporal Imager: An approach in reasoning about time for planning and problem planning*, Yale University, RR #433, October 1985.

[Dean, 1986]

Dean,T.L., "Handling shared resources in a temporal database management system", *Decision Support Systems* 2 (1986), pp. 135-43.

[DeKleer, 1986a]

De Kleer,J., "An Assumption-Based Truth Maintenance System", *Artificial Intelligence* 28 (1986), pp. 127-62.

[DeKleer, 1986b]

De Kleer,J., "Extending the ATMS", *Artificial Intelligence* 28 (1986), pp. 163-96.

[DeKleer, 1986c]

De Kleer,J., "Problem Solving with the ATMS", *Artificial Intelligence* 28 (1986), pp. 197-224.

**[Denholm, 1988]**

Denholm,P., "An Enemy Contact Report Expert System (ECRES)", in: *Intelligence generation and its integration into military command & control systems*, AFCEA Europe Seminar, Rome, 1988.

**[Doyle, 1979]**

Doyle,J., "A truth maintenance system", *Artificial Intelligence* 12 (1979).

**[Engelmore & Morgan, 1988]**

Engelmore,R.S. & Morgan,A.J., *Blackboard Systems*, Addison-Wesley, 1988.

**[Essens, 1989]**

Essens,P.J.M.D., *Orientation command and control research First Army Corps* (Dutch), TNO Institute for Perception, Report IZF 1989-28, 1989.

**[Firby & McDermott, 1987]**

Firby,R.J. & McDermott,D.V., "Representing and solving temporal planning problems", in: Cercone,N. & McCalla,G. (eds.), *The knowledge frontier: essays in the representation of knowledge*, Springer, 1987, pp. 353-413.

**[Galton, 1987]**

Galton,A. (ed.), *Temporal logics and their applications*, Academic Press, 1987.

**[Ginsberg, 1987]**

Ginsberg,M. (ed.), *Readings in non-monotonic reasoning*, Morgan Kaufmann, 1987.

**[Guoxing, 1989]**

Guoxing,H., *Truth maintenance systems and their implementation in PROLOG*, University of Twente, Report UT-KBS-89-01, February 1989.

**[Hanks & McDermott, 1985]**

Hanks,S. and McDermott,D., *Temporal reasoning and default logics*, Yale University, RR #430, October 1985.

**[Harris, 1988]**

Harris,C.J. (ed.), *Application of artificial intelligence to command & control systems*, Peter Peregrinus Ltd., 1988.

**[Keene & Perre, 1990]**

Keene,A.P. & Perre,M., *Data fusion: A preliminary study*, TNO Physics and Electronics Laboratory, Report FEL 90-B356, December 1990.

**[Lakin & Miles, 1987]**

Lakin,W.L. & Miles,J.A.H., "An AI approach to data fusion and situation assessment", in: Harris,C.J. & White,I. (eds.), *Advances in Command, Control and Communications Systems*, Peter Peregrinus Ltd., 1987.

**[Llinas, 1988]**

Llinas, J., "Toward the utilisation of certain elements of AI technology for multi-sensor fusion", in: Harris, C.J. (ed.), *Application of artificial intelligence to command & control systems*, Peter Peregrinus Ltd., 1988.

**[Mason & Johnson, 1988]**

Mason, C.L. & Johnson, R.R., *DATMS: A framework for distributed assumption-based reasoning*, University of California, Department of Applied Science, 1988.

**[McCarthy, 1980]**

McCarthy, J., "Circumscription - A form of non-monotonic reasoning", *Artificial Intelligence* 13 (1980), pp. 27-39.

**[McDermott & Doyle, 1980]**

McDermott, D. & Doyle, J., "Non-monotonic logic I", *Artificial Intelligence* 13 (1980), pp. 41-72.

**[Merritt, 1989]**

Merritt, D., *Building expert systems in Prolog*, Springer Verlag, 1989.

**[Morgue & Chehire, 1991]**

Morgue, G. & Chehire, T., "Efficiently interfacing an ATMS and an Expert System Shell", *Proceedings Avignon '91: Tenth International Workshop on Expert Systems & their Applications*, EC2, 1991, 15 pp.

**[Naylor et al., 1988]**

Naylor, R.T., Roth, A., Bromley, P.A., Wau, S.N.K., "Battlefield data fusion", in: Harris, C.J. (ed.), *Application of artificial intelligence to command & control systems*, Peter Peregrinus Ltd., 1988.

**[Rauch et al., 1983]**

Rauch, H.E., Firschein, O., Perkins, W.A., Pecora, V.J., "An expert system for tactical data fusion", *Proceedings IEEE Conference on Circuits, Systems and Computers*, 1983.

**[Reiter, 1980]**

Reiter, R., "A logic for default reasoning", *Artificial Intelligence* 13 (1980), pp. 81-132.

**[Snodgrass & Ahn, 1986]**

Snodgrass, R. & Ahn, I., "Temporal databases", *IEEE Computer*, September 1986.

**[Sombé, 1990]**

Sombé, L., *Reasoning under incomplete information in artificial intelligence*, John Wiley, 1990.

**[Stakenborg, 1989]**

Stakenborg, L., *Guide to RMS-literature*, Delft University, Report 89-52, 1989.



**[Stam & Snodgrass, 1989]**

Stam,R.B. & Snodgrass,R., "A bibliography on temporal databases", University of North Carolina, 1989.

**[Stonebraker et al., 1986]**

Stonebraker,M. & Rowe,L.A., "The design of POSTGRES", *Proceedings 1986 ACM-SIGMOD Conference*, Washington, 1986.

**[Stonebraker et al., 1990]**

Stonebraker,M., Rowe,L.A., Hirohama,M., "The implementation of POSTGRES", *IEEE Transactions on Knowledge and Data Engineering* 2(1), 1990.

**[VI-MBB-1, 1984]**

*Military command and control* (Dutch), Royal Netherlands Army, 1984.

**[VS 2-1350, 1988]**

*Handbook for the soldier* (Dutch), Royal Netherlands Army, 1988.

**[VS 2-1351, 1988]**

*Handbook for the officer* (Dutch), Royal Netherlands Army, 1988.

**[VS 30-1, 1987]**

*Organization and operational mode of Soviet armed forces* (Dutch), Royal Netherlands Army, 1987.

**[VS 30-5, 1989]**

*Combat intelligence* (Dutch), Royal Netherlands Army, 1989.

## Appendix A: The temporal database system.

The structure of the code in this appendix is as follows:

1	Basic queries	A.2
2	Temporal distance calculation	A.5
3	Temporal queries	A.7
3.1	Relations between timepoints	A.7
3.2	Relations between timepoints and occasions	A.8
3.3	Relations between occasions	A.8
3.4	Relations between occasions and intervals	A.10

## Tequila: Temporal query language

This appendix contains the Prolog source code for the temporal database system.

The clauses governing the consultation and manipulation of the temporal database are listed. These can be divided into three sections. The first contains predicates for the explicit manipulation of the temporal database, being the direct addition, updating and clipping of timedist entries. The second is the code for the function performing the calculation of the shortest path temporal distances. Finally, a collection of temporal queries is listed that allow to query the relations among timepoints, among occasions, and between occasions and timepoints.

### 1 Basic queries

```
timedist(ref,ref,0,0).
timedist(ref,inf,inf,inf).
timedist(inf,inf,0,inf).

add_timedist(X,Time):-
    add_timedist(begin(X),end(X),Time).

add_timedist(X,Y,Time):-
    X = begin(A),
    Y = end(A),
    assert(timedist(ref,X,Time,Time)),
    assert(timedist(X,Y,0,inf)).

add_timedist(X,Y,Low,High):-
    assert(timedist(X,Y,Low,High)).

update_timedist(X,Y,Low,High):-
    retract(timedist(X,Y,_,_)),
    assert(timedist(X,Y,Low,High)).

clip_node(X,ClipTime):-    % in case of a known begintime
    timedist(ref,begin(X),Begin,Begin),
    retract(timedist(begin(X),end(X),_Low,_High)),
    Timediff is ClipTime - Begin,
    myabs(Timediff,Duration),
    assert(timedist(begin(X),end(X),Duration,Duration)),
    nl,
    write('Node '),
```

```
write(X),
write(' clipped at time '),
write(ClipTime),
nl. /* Begin + Duration = ClipTime */

clip_nodelist([],_):-
!.
clip_nodelist([H|T],ClipTime):-
clip_node(H,ClipTime),
clip_nodelist(T,ClipTime).

myabs(A,A):-
A > 0.
myabs(A,B):-
A < 0,
B is -A.

temporal_boundaries(ReportList,Min,Max):-
timelist(ReportList,TimeList),
sort(TimeList,SortedTimeList),
nth1(1,SortedTimeList,Min),
last(Max,SortedTimeList).

temporal_bounds(X,L1,H1,L2,H2):-
calc_temp_distance(ref,begin(X),L1,H1),
calc_temp_distance(ref,end(X),L2,H2),
!.

calc_temp_distance(A,B,K,L):-
findall([A,B,K,L],temporal_distance(A,B,K,L),List),
write_list(List),
!.

time(X):-
findall([X,Low,High],absolutime(X,Low,High),List),
write_list(List),
!.

absolutime(X,Low,High):-
temporal_distance(ref,begin(X),Low,_),
temporal_distance(ref,end(X),_,High).
```

```
timelist([], []):-  
    !.  
timelist([H|T], [Time|TimeList]):-  
    asktms(report(H,_,_,_,_,Time),_),  
    timelist(T,TimeList).  
  
tell_timedist(X,Y):-  
    timedist(X,Y,Low,High),  
    nl,  
    write_timedist(X,Y,Low,High),  
    fail.  
tell_timedist(_,_):-  
    nl,  
    !.  
  
write_timedist(X,Y,Low,High):-  
    nl,  
    write(X),  
    nl,  
    write(Y),  
    nl,  
    write('Low : '),  
    write(Low),  
    nl,  
    write('High: '),  
    write(High),  
    nl,  
    !.
```

## 2 Temporal distance calculation

```
temporal_distance(B,E,Minimum,Maximum):-
  aggregate([max(Min),min(Max)],distance(B,E,Min,Max),[Min1,Max1]),
  (
    Min1 > Max1
    ->
    (nl,
     write('Temporal error.'),
     nl
    )
  )
  ;
  true
),
limitnum(Min1,Minimum),
limitnum(Max1,Maximum).
```

```
distance(Tb,Te,Min,Max):-
  distance(Tb,Te,[Tb],0,0,MinExp,MaxExp),
  call(Min is MinExp),
  call(Max is MaxExp).
```

```
distance(Te,Te,_Path,Min,Max,Min,Max).
distance(T1,Te,Path,Min1,Max1,Min,Max):-
  timedist(T1,T2,Min12,Max12),
  not(member(T2,Path)),
  add_dist(Min1,Min12,Min2),
  add_dist(Max1,Max12,Max2),
  distance(T2,Te,[T2|Path],Min2,Max2,Min,Max).
distance(T1,Te,Path,Min1,Max1,Min,Max):-
  timedist(T2,T1,Min21,Max21),
  not(member(T2,Path)),
  subtr_dist(Min1,Max21,Min2),
  subtr_dist(Max1,Min21,Max2),
  distance(T2,Te,[T2|Path],Min2,Max2,Min,Max).
```

```
add_dist(ref,ref,0).
add_dist(ref,inf,1000000).
add_dist(inf,ref,1000000).
add_dist(inf,inf,2000000).
add_dist(ref,A,A).
add_dist(A,ref,A).
add_dist(inf,A,R):-
  integer(A),
  R is 1000000 + A.
add_dist(A,inf,R):-
  integer(A),
  R is A + 1000000.
```

```
add_dist(A,B,R):-  
    integer(A),  
    integer(B),  
    R is A + B.
```

```
subtr_dist(ref,ref,0).  
subtr_dist(ref,inf,-1000000).  
subtr_dist(inf,ref,1000000).  
subtr_dist(inf,inf,500000).  
subtr_dist(ref,A,-A).  
subtr_dist(A,ref,A).  
subtr_dist(inf,A,R):-  
    integer(A),  
    R is 1000000 - A.  
subtr_dist(A,inf,R):-  
    integer(A),  
    R is A - 1000000.  
subtr_dist(A,B,R):-  
    integer(A),  
    integer(B),  
    R is A - B.
```

```
limitnum(Num,inf):-          % posinf  
    Num >= 999000,  
    !.  
limitnum(Num,-inf):-        % neginf  
    Num =< -999000,  
    !.  
limitnum(Num,Num).
```

### 3 Temporal queries

#### 3.1 Relations between timepoints

```
equal(T,T).
```

```
less(T,inf):-  
    not(T = inf).
```

```
less(ref,T):-  
    not(T = ref).
```

```
less(T1,T2):-  
    integer(T1),  
    integer(T2),  
    T1 < T2.
```

```
less_equal(_,inf).
```

```
less_equal(ref,_).
```

```
less_equal(T1,T2):-  
    integer(T1),  
    integer(T2),  
    T1 =< T2.
```

```
time_equal(T1,T2):-  
    temporal_distance(T1,T2,0,0).
```

```
time_less(T1,T2):-  
    temporal_distance(T1,T2,Low,High),  
    less(0,Low).
```

```
time_less(T1,T2):-  
    temporal_distance(T1,T2,Low,High),  
    less(Low,0),  
    less(0,High).
```

```
time_less_equal(T1,T2):-  
    temporal_distance(T1,T2,Low,High),  
    less_equal(0,Low).
```

```
time_less_equal(T1,T2):-  
    temporal_distance(T1,T2,Low,High),  
    less_equal(Low,0),  
    less_equal(0,High).
```

```
time_in_interval(T,T1,T2):-  
    time_less_equal(T1,T),  
    time_less_equal(T,T2).
```



### 3.2 Relations between timepoints and occasions

```
time_during_occasion(Time,Occasion):-  
    absolutime(Occasion,Low,High),  
    less_equal(Low,Time),  
    time_less_equal(Time,High).
```

```
time_before_occasion(Time,Occasion):-  
    absolutime(Occasion,Low,_),  
    less(Time,Low).
```

```
time_after_occasion(Time,Occasion):-  
    absolutime(Occasion,_,High),  
    less(High,Time).
```

```
time_is_begin_of_occasion(Time,Occasion):-  
    absolutime(Occasion,Low,_),  
    equal(Time,Low).
```

```
time_is_end_of_occasion(Time,Occasion):-  
    absolutime(Occasion,_,High),  
    equal(Time,High).
```

### 3.3 Relations between occasions

```
earlier(Occasion1,Occasion2,Occasion1,Begin1,Begin2):-  
    starts_earlier(Occasion1,Occasion2,Begin1,Begin2).  
earlier(Occasion1,Occasion2,Occasion2,Begin1,Begin2):-  
    starts_earlier(Occasion2,Occasion1,Begin2,Begin1).
```

```
later(Occasion1,Occasion2,Time):-  
    starts_earlier(Occasion1,Occasion2,_,Time).  
later(Occasion1,Occasion2,Time):-  
    starts_earlier(Occasion2,Occasion1,_,Time).
```

```
later(Occasion1,Occasion2,Occasion2,Begin1,Begin2):-  
    starts_earlier(Occasion1,Occasion2,Begin1,Begin2).  
later(Occasion1,Occasion2,Occasion1,Begin1,Begin2):-  
    starts_earlier(Occasion2,Occasion1,Begin2,Begin1).
```

```
discriminate(Occasion1,Occasion2,Occasion1,Occasion2,Begin1,Begin2):-  
    starts_earlier(Occasion1,Occasion2,Begin1,Begin2).
```

```
discriminate(Occasion1,Occasion2,Occasion2,Occasion1,Begin1,Begin2):-
    starts_earlier(Occasion2,Occasion1,Begin2,Begin1).

starts_earlier(Occasion1,Occasion2,Low1,Low2):-
    absolutime(Occasion1,Low1,_),
    absolutime(Occasion2,Low2,_),
    not(Occasion1 = Occasion2),
    less(Low1,Low2).

before(Occasion1,Occasion2):-
    absolutime(Occasion1,_,High1),
    absolutime(Occasion2,Low2,_),
    not(Occasion1 = Occasion2),
    less(High1,Low2).

after(Occasion1,Occasion2):-
    absolutime(Occasion1,Low1,_),
    absolutime(Occasion2,_,High2),
    not(Occasion1 = Occasion2),
    less(High2,Low1).

during(Occasion1,Occasion2):-
    absolutime(Occasion1,Low1,High1),
    absolutime(Occasion2,Low2,High2),
    not(Occasion1 = Occasion2),
    less_equal(Low2,Low1),
    less_equal(High1,High2).

begins_during(Occasion1,Occasion2):-
    absolutime(Occasion1,Low1,_),
    absolutime(Occasion2,Low2,High2),
    not(Occasion1 = Occasion2),
    less_equal(Low2,Low1),
    less(Low1,High2).

ends_during(Occasion1,Occasion2):-
    absolutime(Occasion1,_,High1),
    absolutime(Occasion2,Low2,High2),
    not(Occasion1 = Occasion2),
    less(Low2,High1),
    less_equal(High1,High2).

overlaps(Occasion1,Occasion2):-
    begins_during(Occasion1,Occasion2)
    ;
    ends_during(Occasion1,Occasion2).
```

```
coincides(Occasion1,Occasion2):-  
    absolutime(Occasion1,Low1,High),  
    absolutime(Occasion2,Low,High),  
    not(Occasion1 = Occasion2).
```

```
disjoint_occasions(Occasion1,Occasion2):-  
    not(overlaps(Occasion1,Occasion2)).
```

### 3.4 Relations between occasions and intervals

```
comes_before(Occasion,BeginTime,_):-  
    absolutime(Occasion,_,High),  
    less_equal(High,BeginTime).
```

```
comes_after(Occasion,_,EndTime):-  
    absolutime(Occasion,Low,_),  
    less_equal(EndTime,Low).
```

```
true_throughout(Occasion,BeginTime,EndTime):-  
    absolutime(Occasion,Low,High),  
    less_equal(Low,BeginTime),  
    less_equal(EndTime,High).
```

```
lies_in(Occasion,BeginTime,EndTime):-  
    absolutime(Occasion,Low,High),  
    less_equal(BeginTime,Low),  
    less_equal(High,EndTime).
```

```
past_overlaps(Occasion,BeginTime,EndTime):-  
    absolutime(Occasion,Low,High),  
    not(Low = BeginTime, High = EndTime),  
    less_equal(Low,BeginTime),  
    less_equal(High,EndTime).
```

```
future_overlaps(Occasion,BeginTime,EndTime):-  
    absolutime(Occasion,Low,High),  
    not((Low = BeginTime, High = EndTime)),  
    less_equal(BeginTime,Low),  
    less_equal(EndTime,High).
```

## Appendix B: Inference and interaction TDB-ATMS

The structure of the code in this appendix is as follows:

1	Inference engine	B.2
2	Match and process clauses	B.4
2.1	Match and meet	B.4
2.2	Process and take	B.5
3	Interaction with TDB and ATMS	B.9

## TDB-ATMS: Inference and interaction

This appendix contains the Prolog source code for the rule firing mechanism and the predicates that handle the interaction between the ATMS and the temporal database.

The interaction with the ATMS and the temporal database is triggered by the application of the rules. Simultaneous calls to the ATMS and the temporal database will be contained in the rules as actions. The forward chaining rule application is a "match and process" algorithm that sequentially steps through the rules.

The code in this appendix is divided into three sections. The rule firing mechanism is listed first. Secondly, the match and process clauses are listed, showing how - among others - the "assume", "derive", and "derive\_and\_assume" calls to the ATMS and the temporal database have been implemented. This is subdivided into a "match-and-meet" and a "process-and-take" section, respectively dealing with the conditions and the actions in the rules. The "take(Action)" clauses of the rule application use the predicates that deal with the interaction between the problem solver and the ATMS, the temporal database, as well as the interaction amongst the latter two. These predicates that are used in the match and process clauses are contained in the third section.

```
1      Inference engine

fire(Type):-
    fire_all(Type,_,_,_)
;
!.

fire(Type,Nr):-
    fire_all(Type,Nr,_,_)
;
!.

fire_all(Type,Nr,Description,LHS,RHS):-
    rule(Type,Nr,Description)::LHS==>RHS,
    fire_rule(Type,Nr,Description,LHS,RHS),
    !.

```

```
fire_rule(Type,Nr,Description,LHS,RHS):-
    match(LHS),
    process(RHS,LHS,Type),
    writerule(Type,Nr,Description),
    fail.
fire_rule(_,_,_,_):-
    nl,
    write('Fire rule? '),
    read(Answer),
    carry_out(Answer),
    !.

carry_out(y):-
    !,
    fail.
carry_out(n):-
    !.
carry_out(_):-
    nl,
    write('Not a valid command, enter (y/n): '),
    read(Answer),
    carry_out(Answer).

writerule(Type,Nr,Description):-
    nl,
    write('Rule fired: '),
    write(Type),
    write(', '),
    write(Nr),
    write(', '),
    write(Description),
    nl,nl,
    !.

write_list({}):-
    !.
write_list([H|T]):-
    write(H),
    nl,
    write_list(T).
```

## 2 Match and process clauses

### 2.1 Match and meet

```
match({}):-
!,
match([Condition|Rest]):-
!,
meet(Condition),
match(Rest).

meet(tms(neg(Prem))):-          % meet tms condition
!,
tms_node(_,neg(Prem),in,_,_,_,_,_).
meet(tms(not(Prem))):-
!,
not(node_exists(Prem,in)).
meet(tms(out(Prem))):-
!,
tms_node(_,Prem,out,_,_,_,_,_).
meet(tms(Prem)):-
!,
tms_node(_,Prem,in,_,_,_,_,_).

meet(neg(Prem)):-              % meet prolog condition
!,
neg(Prem).
meet(not(Prem)):-
!,
not(Prem).
meet(Prem):-
!,
Prem.
```

## 2.2 Process and take

```
process([],_,_):-
!.
process({Action|Rest},LHS,Type):-
take(Action,LHS,Type),
!,
process(Rest,LHS,Type).

take(asktms(X,Status),_,_):-
asktms(X,Status).
take(replace_node(X,Y),_,_):-
tms_replace(X,Y).
take(add_premise(X,Time),_,_):-
add_premise(X,Time).

take(assume(X,Time),_,_):-
(check_node_exists(X,_))
;
build_tms_node(X)
),
assume_node(X),
nl,
write('Adding assumption: '),
write(X),
add_timedist(X,Time),
nl,
write('Starting at time : '),
write(Time),
nl.

take(derive(neg(Y),_,Time),LHS,Type):-
X = neg(Y),
take(derive(Y,_,Time),LHS,Type).
take(derive(neg(X),NogoodNodeList,Time),LHS,Type):-
(
check_node_exists(neg(X),_)
;
build_tms_node(neg(X))
),
(
(check_assumption(X),
pass_nogood(X,NogoodNodeList))
)
;
NogoodNodeList = []
),
make_antecedent_list(LHS,Ante_List),
new_justification(Type,neg(X),Ante_List),
nl,
write('Adding justification for: '),
```



```
write(neg(X)),
add_timedist(X,Time),
nl,
write('Starting at time: '),
write(Time),nl.
take(derive(X,NogoodNodeList,Time),LHS,Type):-
(   check_node_exists(X,_)
;
   build_tms_node(X)
),
(   (check_assumption(neg(X)),
     pass_nogood(neg(X),NogoodNodeList)
   )
;
   NogoodNodeList = []
),
make_antecedent_list(LHS,Ante_List),
new_justification(Type,X,Ante_List),
nl,
write('Adding justification for: '),
write(X),
add_timedist(X,Time),
nl,
write('Starting at time: '),
write(Time),
nl.
take(derive_and_assume(neg(X),_,Time),LHS,Type):-
X = neg(Y),
take(derive_and_assume(Y,_,Time),LHS,Type).
take(derive_and_assume(neg(X),NogoodNodeList,Time),LHS,Type):-
(   check_node_exists(neg(X),_)
;
   build_tms_node(neg(X))
),
(   (check_assumption(X),
     pass_nogood(X,NogoodNodeList)
   )
;
   NogoodNodeList = []
),
make_antecedent_list(LHS,Ante_List),
new_justification(Type,neg(X),Ante_List),
assume_node(neg(X)),
nl,
write('Adding justification and setting assumption: '),
write(neg(X)),
add_timedist(X,Time),
nl,
write('Starting at time: '),
write(Time),
nl.
```

```
take(derive_and_assume(X,NogoodNodeList,Time),LHS,Type):-
    (
        check_node_exists(X,_);
        build_tms_node(X)
    ),
    (
        (check_assumption(neg(X)),
        pass_nogood(neg(X),NogoodNodeList)
        )
    );
    NogoodNodeList = []
),
make_antecedent_list(LHS,Ante_List),
new_justification(Type,X,Ante_List),
assume_node(X),
nl,
write('Adding justification and setting assumption: '),
write(X),
add_timedist(X,Time),
nl,
write('Starting at time: '),
write(Time),
nl.

take(set_node(X,out),_,_):-
    check_node_exists(X,out).
take(set_node(X,out),_,_):-
    check_node_exists(X,in),
    set_node(X,cut).
take(set_node(X,out),_,_):-
    build_tms_node(X).
take(pass_nogood(X,List),_,_):-
    pass_nogood(X,List).
take(set_nogood(X,Time),_,_):-
    set_nogood(X,Time).

take(timedist(X,Y,Low,High),_,_):-
    timedist(X,Y,Low,High).
take(temporal_distance(X,Y,Min,Max),_,_):-
    temporal_distance(X,Y,Min,Max).
take(add_timedist(X,Y,Low,High),_,_):-
    add_timedist(X,Y,Low,High).
take(assert(X),_,_):-
    assert(X).
take(retract(X),_,_):-
    retract(X).
take(call(X),_,_):-
    call(X).
take((X:Y),_,_):-
    take(X,_,_)
;
    take(Y,_,_).
```

```
take((X = Y),_,_):-
    X = Y.
take((X $ Y),_,_):-
    X is Y.
take(write(X),_,_):-
    write(X).
take(write_list(X),_,_):-
    write_list(X).
take(nl,_,_):-
    nl.
take(read(X),_,_):-
    read(X).
take(prompt(X,Y),_,_):-
    nl,
    write(X),
    read(Y).
take(X,_,_):-          % if all else fails: a prolog call
    call(X)
;
fail.
```

```
make_antecedent_list([],[]):-
    !.
make_antecedent_list([H|T],[X|List]):-
    H = tms(X),
    make_antecedent_list(T,List).
make_antecedent_list([H|T],List):-
    \+ H = tms(_),
    make_antecedent_list(T,List),
    !.
```

### 3 Interaction with TDB and ATMS

```
asktms(X, Status):-
    tms_node(_, X, Status, _, _, _, _).

fetchtms(X):-
    asktms(X, in).
fetchtms(_):-
    !.

check_node_exists(X, Status):-
    tms_node(_, X, Status, _, _, _, _).

tell_node(X, Status):-
    tms_node(_, X, Status, _, _, _, _),
    nl,
    write(X),
    fail.
tell_node(_,-):-
    nl,
    !.

check_assumption(X):-
    tms_node(_, X, in, _, _, _, ['Assumption']).

tell_assumption(A):-
    assumption(_, A, _),
    nl,
    write(A),
    fail.
tell_assumption(_):-
    nl,
    !.

tms_replace(X, Y):-
    retract(tms_node(N, X, S, L, J, C, R, P)),
    assert(tms_node(N, Y, S, L, J, C, R, P)).

set_node(X, out):-
    retract(tms_node(_, X, _, _, _, _, _)),
    assert(tms_node(_, X, out, _, _, _, _)).
set_node(X, in):-
    retract(tms_node(_, X, _, _, _, _, _)),
    assert(tms_node(_, X, in, _, _, _, _)).
```

```
add_premise(X,T):-  
    build_tms_node(X),  
    set_premise_node(X),  
    add_timedist(X,T).
```

```
add_assumption(X,T):-  
    build_tms_node(X),  
    assume_node(X),  
    add_timedist(X,T).
```

```
add_justification(X,A,T):-  
    build_tms_node(X),  
    new_justification(just,X,A),  
    add_timedist(X,T).
```

```
pass_nogood(X,NogoodNodeList):-  
    set_nogood_nodes([X],NogoodNodeList),  
    nl,  
    write('Setting nogood: '),  
    write(X),  
    nl,nl,  
    write('Nodes set to out by ATMS are: '),  
    nl,  
    write_list(NogoodNodeList),  
    nl,  
    !.
```

```
pass_nogood_list([],_,_).  
pass_nogood_list([X|Rest],Time,TotalList):-  
    pass_nogood(X,NogoodList),  
    append(NogoodList,List,TotalList),  
    pass_nogood_list(Rest,Time,List).
```

```
set_nogood(X,T):-  
    pass_nogood(X,List),  
    clip_node(X,T),  
    clip_nodelist(List,T).
```

## Appendix C:

### The Assumption-based Truth Maintenance System

The structure of the code in this appendix is as follows:

1	Build nodes, assumptions, environments	C.3
2	Build justification	C.5
3	Update node	C.6
4	Update label	C.7
4.1	Compute justification environment	C.8
4.1.1	Make environment from assumption	C.8
4.1.2	Generate environment cross product	C.9
4.2	Remove subsumed environments	C.12
4.2.1	Environment subsumed	C.16
4.2.2	Check contradictory environments	C.17
5	Update nogood	C.19
5.1	Process nogood tables	C.20
5.2	Process environment tables	C.21

## ATMS Assumption-based Truth Maintenance System

The implementation of the ATMS is based on the implementation in Prolog, listed in [Guoxing, 1989]. We have adapted the code to acquire an interface with our application, to facilitate interaction with the temporal database and to remove some errors from the original code. However, we stress that the basic algorithms governing the ATMS are unaltered and attributed to [Guoxing, 1989]. The size of the ATMS implementation is over 26 KB, we will not list the complete code here. Below, we list selections from the ATMS source code, indicating the adaptations that we have made that do not only concern the interface.

The ATMS implementation has four main functions that govern the construction and maintenance of the four main data structures of the ATMS, discussed in paragraph 3.3.2. The functions are the following:

```
build_node.  
build_assumption.  
build_environment.  
build_justification.
```

The three functions corresponding to the building of nodes, assumptions and justifications are initiated from outside the ATMS, as clarified in this report. The "build\_environment" function is initiated by the ATMS itself, by the "build\_justification" function, which forms the heart of the ATMS. It is this latter function that initiates the truth maintenance. When a new justification is passed to the ATMS, three main functions are triggered to recalculate the consistency of the database. These functions are the updating of nodes, the updating of labels and the updating of nogoods.

The updating of nogoods uses environment tables and nogood tables as main data structures. The environment tables contain the environments in the ATMS, they are ordered by the number of assumptions the environments consists of. Thus, there is an environment table containing the environments with one assumption, with two assumptions, etcetera. Nogood tables are used to keep track of the inconsistent environments in the ATMS, and thus the inconsistent assumptions.

The main adaptation necessary in the ATMS itself to allow a combination of truth maintenance and temporal reasoning techniques is to incorporate the returning of a list of the nodes that were set to out as a result of a recomputation of the labels, see paragraph 3.4.2 and 3.4.3. The 'NogoodNodeList' records the nodes set to out in the course of the label update mechanism. It was added to the code in several places, because it "snakes" through the program. It is explicitly constructed by the ATMS in the "do\_delete\_env" function, see page C.18.

There were several errors in the code that needed repair. The main error was that on passing nogoods to the ATMS the calculation of the subsumption did not travel "far enough" through the ATMS. Beside this, all environments were set to "contradictory" due to this incorrect calculation of the subsumption. The result was that nodes remained valid incorrectly because their labels were not correctly adjusted. The errors and adaptations made are stated in the code below.

## 1 Build nodes, assumptions, environments

```
%% build_tms_node

build_tms_node(Datum):-
    (tms_node(_,Datum,_,_,_,_,_)
    ->
    (write('Node already existed'),
    nl
    )
    ),
    (node_count(C),
    assertz(tms_node(C,Datum,out,0,[],[],[]))
    ),
    system_init1.

%% build_assumption

build_assumption(Datum):-
    tms_node(Index,Datum,Status,Label,Justi,Consq,Rules,Plist),
    add_element('Assumption',Plist,P1),
    retract(tms_node(Index,Datum,Status,Label,Justi,Consq,Rules,Plist)
    ),
    assumption_count(C),
    assertz(assumption(C,Datum,[])),
    assertz(tms_node(Index,Datum,Status,Label,Justi,Consq,Rules,P1)).
```



```
%% build_environment

build_environment(Assum):-
    environment_count(C),
    length(Assum,Len),
    assertz(environment(C,Len,Assum,[],[])),
    fill_assum_env(Assum,C),
    insert_env_in_table(Len,C).

fill_assum_env([],_):-
    !.
fill_assum_env([H|T],Env):-
    retract(assumption(H,Assum,Oenv)),
    ord_add_element(Oenv,Env,Nenv),
    assertz(assumption(H,Assum,Nenv)),
    fill_assum_env(T,Env).

insert_env_in_table(Len,Env_index):-
    retract(env_table(Len,Eset)),
    ord_add_element(Eset,Env_index,E1),
    assertz(env_table(Len,E1)).
```

## 2 Build justification

```
build_justification(Type,Conseq,Antes,NogoodNodeList):-  
% NogoodNodeList added, snakes through rest of the ATMS  
justification_count(C),  
push_j_tms_node_just(Conseq,C),  
push_j_tms_node_cons(Antes,C),  
assertz(justification(C,Type,Conseq,Antes)),  
update_something(Conseq,C,NogoodNodeList).
```

```
push_j_tms_node_just(Conseq,Just_index):-  
retract(tms_node(Conseq,D,S,L,J,Cons,R,Plist)),  
ord_add_element(J,Just_index,Njust),  
assertz(tms_node(Conseq,D,S,L,Njust,Cons,R,Plist)).
```

```
push_j_tms_node_cons([],_):-
```

```
push_j_tms_node_cons([H|T],Just_index):-  
process_one_by_one(H,Just_index),  
push_j_tms_node_cons(T,Just_index).
```

```
process_one_by_one(Ante,Just_index):-  
retract(tms_node(Ante,D,S,L,J,Cons,R,Plist)),  
(Plist=='Assumption')  
->  
assertz(tms_node(Ante,D,S,L,J,Cons,R,Plist))  
;  
(ord_add_element(Cons,Just_index,Ncons),  
assertz(tms_node(Ante,D,S,L,J,Ncons,R,Plist))  
).  
).
```

```
update_something(Conseq,Just_index,NogoodNodeList):-  
tms_node(Conseq,Datum,_,_,_,_),  
(Datum==contra_node  
->  
update_nogood(Just_index,NogoodNodeList)  
;  
(retract(node_queue(X)),  
ord_add_element(X,Conseq,X1),  
assertz(node_queue(X1)),  
update_node(X1)  
).  
).
```

### 3 Update node

```
update_node([]):-
!.

update_node([H|_T]):-
    update_label(H),
    modi_node_queue(H),
    do_loop_test(H),
    node_queue(Qt),
    (Qt==[]
     ->
      !
     ;
     update_node(Qt)
    ).

modi_node_queue(Nd):-
    retract(node_queue(Nq)),
    del_element(Nd,Nq,Nq1),
    assertz(node_queue(Nq1)).

do_loop_test(H):-
    new_env(New_env),
    (New_env==[]
     ->
      !
     ;
     (tms_node(H,_Datum,_,_,Node_conseq,_,_)
      do_list_conseq(Node_conseq)
     )
    ).

do_list_conseq([]):-
!.

do_list_conseq([H|T]):-
    justification(H,_Type,Justi_conseq,_),
    do_justi_conseq(H,Justi_conseq),
    do_list_conseq(T).

do_justi_conseq(H,Conseq):-
    Conseq==contra_node
    ->
    update_nogood(H,_)
    ;
    add_node_queue(Conseq).

add_node_queue(Node):-
    retract(node_queue(Queue)),
    ord_add_element(Node,Queue,Q1),
    assertz(node_queue(Q1)).
```

## 4 Update label

```
update_label(Node):-
    tms_node(Node, Datum, Label, Justi, _, _),
    (Label==[]
     ->
      (retract(env_product(_)),
       assertz(env_product([]))
      )
     ;
     do_update_label(Justi, Node)
    ).

do_update_label([], _) :-
    !.
do_update_label([H|T], Node):-
    compute_justification_env(H),
    env_product(Penv),
    process_penv(Penv, Node),
    do_update_label(T, Node).

%% compute justification environment

compute_justification_env(H):-
    justification(H, Type, _, Antes),
    process_antes(Antes),
    comm_process_for_process_antes.

process_antes([]):-
    !.
process_antes([H|T]):-
    tms_node(H, Datum, Label, _, _, Plist),
    (Plist=='Assumption'
     ->
      (assumption(Ind, Datum, _),
       push_input_assumption(Ind)
      )
     ;
     push_input_envs(Label)
    ),
    process_antes(T).

push_input_assumption(H):-
    retract(input_assumption(Iassum)),
    ord_add_element(Iassum, H, I1),
    assertz(input_assumption(I1)).

push_input_envs(Label):-
    retract(input_envs(Env)),
    ord_union(Label, Env, E1),
```

```
list_to_ord_set(E1,E2),
assertz(input_envs(E2)).
```

```
comm_process_for_process_antes:-
input_assumption(Assums),
make_env_from_assumption(Assums),
base_env(Base_env),
input_envs(Env_choices),
((Assums==[]
;
Base_env\==[]
)
->
generate_env_cross_product(Base_env,Env_choices)
;
generate_env_cross_product([],Env_choices)
).
```

#### 4.1 Compute justification environment

##### 4.1.1 Make environment from assumption

```
make_env_from_assumption([]):-
!.
make_env_from_assumption([H|T]):-
base_env(Base_env),
cons_env(H,Base_env),
base_env(Nenv),
(Nenv==[]
->
(retract(base_env(_)),
assertz(base_env([]))
)
;
make_env_from_assumption(T)
).
```

```
cons_env(Assum,Env):-
ord_add_element(Env,Assum,Nenv),
(environment(_E_ind,_,Nenv,_,_)
->
cons_env_return2(Nenv)
;
cons_env_return1(Nenv)
).
```

```
cons_env_return1(Env):-
    build_environment(Env),
    environment(E_ind,_,Env,_,_),
    (check_contradictory(E_ind)
     ->
        (retract(base_env(Benv)),
         assertz(base_env({})))
        )
    ;
    (retract(base_env(Benv)),
     assertz(base_env(Env)))
    ).
```

```
cons_env_return2(Env):-
    environment(_E_ind,_,Env,_,Contr),
    (Contr\=[]
     ->
        (retract(base_env(Benv)),
         assertz(base_env({})))
        )
    ;
    (retract(base_env(Benv)),
     assertz(base_env(Env)))
    ).
```

#### 4.1.2 Generate environment cross product

```
generate_env_cross_product(Base_env, []):-
    env_product(Env_p),
    environment(Ind,_,Base_env,_,_),
    (Env_p=0
     ->
        push_in_env_product(Base_env)
    ).
```

```
;
((new_env_subsumed(Ind, Env_p)
;
check_contradictory(Ind)
)
->
push_in_env_product(Base_env)
;
(do_check_subsumed(Ind, Env_p),
push_in_env_product(Base_env)
)
)
).

generate_env_cross_product(Base_env, [H|T]) :-
retract(base_env(_)),
assertz(base_env(Base_env)),
append_envs(H, Base_env),
base_env(Nenv),
(Nenv==[]
->
!
;
generate_env_cross_product(Nenv, T)
).

push_in_env_product(Base_env) :-
retract(env_product(Env_product)),
environment(Ind, _, Base_env, _, _),
(Env_product==0
->
ord_add_element([], Ind, Ep)
;
ord_add_element(Env_product, Ind, Ep)
),
```

```
list_to_ord_set(Ep,Epl),
assertz(env_product(Epl)).

append_envs(One_of_input_envs,[]):-
retract(base_env(_)),
environment(One_of_input_envs,_,Assums,_,_),
assertz(base_env(Assums)).
append_envs(One_of_input_envs,Base_env):-
environment(One_of_input_envs,N1,Assum1,_,_),
environment(_Ind,N2,Base_env,_,_),
(N1 > N2
->
(retract(base_env(_)),
assertz(base_env(Assum1)),
make_env_from_assumption(Base_env)
)
;
make_env_from_assumption(Assum1)
).

do_check_subsumed(_,[]):-
!.
do_check_subsumed(Base_env,[H|T]):-
env_subsumed(H,Base_env)
->
(retract(env_product(Env_p)),
ord_del_element(Env_p,H,E3),
assertz(env_product(E3)),
do_check_subsumed(Base_env,T)
)
;
do_check_subsumed(Base_env,T).
```



## 4.2 Remove subsumed environments

```
process_penv(Penv,Node):-
    new_env(New_env),
    case_one(Penv,New_env),
    case_two(Node).

case_one([],_):-
    !.
case_one([HIT],New_env):-
    (New_env==[]
     ->
      order_push_penv(H)
     ;
      sub_case_one(H,New_env)
    ),
    new_env(Nnenv),
    case_one(T.Nnenv).

sub_case_one(Penv,New_env):-
    new_env_subsumed(Penv,New_env)
    ->
    !
    ;
    (check_contradictory(Penv)
     ->
      !
     ;
      do_modify_new_env(Penv,New_env)
    ).

new_env_subsumed(_Penv,[]):-
    !,
    fail.
```

```
new_env_subsumed(Penv, [H|T]) :-
    env_subsumed(Penv, H)
    !
    ;
    new_env_subsumed(Penv, T).

do_modify_new_env(_, []) :-
    !.
do_modify_new_env(Penv, [H|T]) :-
    (env_subsumed(H, Penv)
    ->
        (retract(new_env(New_env)),
        del_element(H, New_env, N1),
        assertz(new_env(N1)),
        order_push_penv(Penv)
        )
    ;
    order_push_penv(Penv)
    ),
    do_modify_new_env(Penv, T).

order_push_penv(Penv) :-
    retract(new_env(Env)),
    ord_add_element(Env, Penv, New_env),
    assertz(new_env(New_env)).

case_two(Node) :-
    tms_node(Node, _, Label, _, _, _),
    new_env(New_env),
    (seteq(Label, New_env)
    ->
        (retract(new_env(_)),
        assertz(new_env([]))
        )
    ).
```

```
    )
;
    (del_env_nodes(Label,Node,New_env),
     add_env_nodes(New_env,Node,Label),
     modify_node_label(New_env,Node),
     modify_node_other_fields(New_env,Node)
    )
).

del_env_nodes(0,_,_):-
!.
del_env_nodes([],_,_):-
!.
del_env_nodes([H|T],Node,New_env):-
    (subset(H,New_env)
     ->
     !
     ,
     (retract(environment(H,N,A,Enode,Contr)),
      del_element(Node,Enode,E1),
      assertz(environment(H,N,A,E1,Contr))
     )
    ),
    del_env_nodes(T,Node,New_env).

add_env_nodes([],_,_):-
!.
add_env_nodes([H|T],Node,Label):-
    (subset(H,Label)
     ->
     !
     ,
     (retract(environment(H,N,A,Enode,Contr)),
      ord_add_element(Enode,Node,E1),

```

```
        assertz(environment(H,N,A,EI,Contr))
      )
      ;
      !
    ),
    add_env_nodes(T,Node,Label).

modify_node_label(New_env,Node):-
  retract(tms_node(Node,D,S,_L,J,C,R,P)),
  assertz(tms_node(Node,D,S,New_env,J,C,R,P)).

%code added for the case New_env=[] and L1=[]
modify_node_other_fields(New_env,Node):-
  retract(tms_node(Node,D1,_S1,L1,J1,C1,R1,P1)),
  (New_env==[]
  ->
  (L1==[]
  ->
  assertz(tms_node(Node,D1,in,L1,J1,C1,R1,P1))
  ;
  assertz(tms_node(Node,D1,out,L1,J1,C1,R1,P1))
  )
  ;
  assertz(tms_node(Node,D1,in,L1,J1,C1,R1,P1))
  ).
```

## 4.2.1 Environment subsumed

```
env_subsumed([], []):-
    !.
env_subsumed(_, []):-
    !.
env_subsumed([], _):-
    !,
    fail.
env_subsumed(E1, E2):-
    E1==E2,
    !.
env_subsumed(E1, E2):-
    environment(E1, N1, Assums1, _, _),
    environment(E2, N2, Assums2, _, _),
    (N1 < N2
    ->
    fail
    ;
    check_assumption_from_env(Assums1, Assums2)
    ).

check_assumption_from_env([], []):-
    !.
check_assumption_from_env(_, []):-
    !.
check_assumption_from_env([], _):-
    !,
    fail.
check_assumption_from_env([H1|T1], [H2|T2]):-
    (H1==H2
    ->
    check_assumption_from_env(T1, T2)
    ;
    ).
```

```
(H1 < H2
->
check_assumption_from_env(T1, [H2|T2])
;
fail
)
).
```

#### 4.2.2 Check contradictory environments

```
check_contradictory(Env_ind):-
environment(Env_ind,N,_Assums_,_Contr),
(Contr\==[]
->
!
;
lookup_nogood_table(N,Env_ind)
).
```

```
lookup_nogood_table(N,Env_ind):-
countk(I),
nogood_table(I,Cenv),
I =< N
->
(subsumed_nogood(Cenv,Env_ind,I)
->
end_lookup_nogood_table
;
lookup_nogood_table(N,Env_ind)
)
;
(end_lookup_nogood_table,
fail
).
```

```
end_lookup_nogood_table:-
    retract(countk_aux(_)),
    assertz(countk_aux(0)),
    !.

subsumed_nogood([],_,_):-
    !,
    fail..
subsumed_nogood([H|T],Env_ind,I):-
    env_subsumed(Env_ind,H)
    ->
    (retract(environment(Env_ind,Num,Assums,Nodes,_Contr)),
     assertz(environment(Env_ind,Num,Assums,Nodes,H))
    )
    ;
    subsumed_nogood(T,Env_ind,I).
```

## 5 Update nogood

```
update_nogood(Just, NogoodNodeList):-
    compute_justification_env(Just),
    env_product(Env_product),
    clear_nogood(Env_product, Just, NogoodNodeList).

% code adapted, before: when Env_product = 0 "clear_nogood" failed
clear_nogood([],_,_):-
    !.
clear_nogood([H|T], Just, NogoodNodeList):-
    retract(environment(H, N, Assum, Nodes, _Contr)),
    assertz(environment(H, N, Assum, Nodes, Just)),
    remove_env_from_labels(H, NogoodNodeList),
    insert_nogood_in_table(N, H),
    process_nogood_table(N, H),
    process_env_table(N, H),
    clear_nogood(T, Just, NogoodNodeList).

remove_env_from_labels(Env, NogoodNodeList):-
    environment(Env, N, Assum, Nodes, _Contr),
    do_delete_env(Nodes, Env, NogoodNodeList).

% code adapted, NogoodNodeList is constructed here
do_delete_env([],_,[]):-
    !.
do_delete_env([H|T], Env, [NogoodNode|List]):-
    retract(tms_node(H, D, S, L, J, C, R, P)),
    del_element(Env, L, L1),
    (L1==[]
    ->
    (assertz(tms_node(H, D, out, 0, J, C, R, P)),
    NogoodNode = D
    )
    )
```



```

      (assertz(tms_node(H,D,S,L1,J,C,R;P)),
        NogoodNode = [])
    ),
  ),
  retract(node_queue(Node_q)),
  ord_add_element(Node_q,H,Node1),
  assertz(node_queue(Node1)),
  do_delete_env(T,Env,List).

insert_nogood_in_table(N_assums,Env):-
  N_assums==0
  ->
  !
  (retract(nogood_table(N_assums,Nogood)),
  ord_add_element(Nogood,Env,Nnogood),
  assertz(nogood_table(N_assums,Nnogood))
  ).
```

### 5.1 Process nogood tables

```
process_nogood_table(0,_):-
  !.
process_nogood_table(129,_):-
  !.
process_nogood_table(N,Cenv):-
  nogood_table(N,Nogood),
  do_test_nogood_subsumed(Nogood,Cenv,N),
  Y is N+1,
  process_nogood_table(Y,Cenv).
```

```
do_test_nogood_subsumed([],_,_):-
!.
do_test_nogood_subsumed([H|T],Cenv,N):-
(H=Cenv,
 T=[]
)
->
!.
/
(env_subsumed(H,Cenv)
->
(dele_h_from_nogood_table(N,H),
 do_test_nogood_subsumed(T,Cenv,N)
)
)
do_test_nogood_subsumed(T,Cenv,N)
).

dele_h_from_nogood_table(N,H):-
retract(nogood_table(N,Nogood)),
del_element(H,Nogood,Nnogood),
assert(nogood_table(N,Nnogood)).
```

## 5.2 Process environment tables

```
process_env_table(0,_):-
!.
process_env_table(129,_):-
!.
process_env_table(N,Cenv):-
env_table(N,E_table),
do_test_env_subsumed(E_table,N,Cenv),
Y is N+1,
process_env_table(Y,Cenv).
```

```
% error, code added, before: every node contradictory!
do_test_env_subsumed([],_,_):-
!.
do_test_env_subsumed([H|T],N,Cenv):-
environment(H,Na,Ass,No,Contr), ((Contr == [],
env_subsumed(H,Cenv) % H subsumed by Cenv (H>=Cenv)
)
->
(retract(environment(N,Na,Ass,No,Contr)),
assertz(environment(H,Na,Ass,No,Cenv)),
remove_env_from_labels(H,_),
do_test_env_subsumed(T,N,Cenv)
)
;
((Contr == []
->
do_test_env_subsumed(T,N,Cenv)
;
(retract(environment(H,Na,Ass,No,Contr)),
assertz(environment(H,Na,Ass,No,Cenv)),
do_test_env_subsumed(T,N,Cenv)
)
)
)
).
```

UNCLASSIFIED

REPORT DOCUMENTATION PAGE

(MOD-NL)

1. DEFENSE REPORT NUMBER (MOD-NL) 2. RECIPIENT'S ACCESSION NUMBER 3. PERFORMING ORGANIZATION REPORT NUMBER  
TD91-3885 - FEL-91-B308

4. PROJECT/TASK/WORK UNIT NO. 5. CONTRACT NUMBER 6. REPORT DATE  
22441 - NOVEMBER 1991

7. NUMBER OF PAGES 8. NUMBER OF REFERENCES 9. TYPE OF REPORT AND DATES COVERED  
100 (INCL. APPEND. & RDP, EXCL. DIST. LIST) 43 FINAL REPORT

10. TITLE AND SUBTITLE  
DATA FUSION: TEMPORAL REASONING AND TRUTH MAINTENANCE

11. AUTHOR(S)  
A.P. KEENE, M. PERRE

12. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)  
TNO PHYSICS AND ELECTRONICS LABORATORY, P.O. BOX 96864, 2509 JG THE HAGUE  
OUDE WAALSDORPERWEG 63, THE HAGUE, THE NETHERLANDS

13. SPONSORING/MONITORING AGENCY NAME(S)  
NETHERLANDS MINISTRY OF DEFENCE

14. SUPPLEMENTARY NOTES  
THE PHYSICS AND ELECTRONICS LABORATORY IS PART OF THE NETHERLANDS ORGANIZATION FOR APPLIED SCIENTIFIC RESEARCH

15. ABSTRACT (MAXIMUM 200 WORDS, 1044 POSITIONS)  
THIS REPORT CONTAINS A SURVEY OF TWO TECHNIQUES THAT CAN BE USED IN THE FIELD OF DATA FUSION: TEMPORAL REASONING AND TRUTH MAINTENANCE. THE AUTOMATIC FUSION OF INTELLIGENCE REPORTS NECESSITATES TAKING INTO ACCOUNT THE FACTOR TIME. INCOMING MESSAGES CAN LEAD TO NEW INTERPRETATIONS OF THE CURRENT BATTLEFIELD SITUATION, CHANGING PREVIOUSLY MADE HYPOTHESES. A DATA FUSION SYSTEM MUST ALSO BE ABLE TO MAKE A PREDICTION OF WHAT SIGHTINGS ARE TO BE EXPECTED, E.G. IN THE CASE OF COLUMNS OF VEHICLES MOVING PAST DIFFERENT SENSORS. THIS REPORT DESCRIBES A TEMPORAL DATABASE SYSTEM THAT CAN CAPTURE (SOME PART) OF THE VOLATILITY OF THE INTELLIGENCE PROCESSING DOMAIN. WHILE PROCESSING INTELLIGENCE REPORTS THERE IS ALWAYS AN AMOUNT OF UNCERTAINTY AND INCOMPLETENESS THAT HAS TO BE DEALT WITH. SO THERE IS A NEED FOR MAINTAINING DIFFERENT LINES OF REASONING OR HYPOTHESES PERTAINING TO THE BATTLEFIELD SITUATION CONCURRENTLY, AND INCORPORATING NEW INFORMATION AS IT BECOMES AVAILABLE. IN THIS REPORT AN ASSUMPTION-BASED TRUTH MAINTENANCE SYSTEM PROVIDES A FRAMEWORK IN WHICH THIS PROBLEM CAN BE SOLVED. A PROTOTYPE HAS BEEN DEVELOPED TO DEMONSTRATE THE APPLICABILITY OF THE AFOREMENTIONED TECHNIQUES. THIS PROTOTYPE, CALLED MEFISTO (MODULAR ENVIRONMENT FOR FUSION AND INTERPRETATION OF SENSOR DATA IN TRACKING OPPOSING FORCES), IS A SIMPLE KNOWLEDGE-BASED SYSTEM INTEGRATED WITH A TEMPORAL TRUTH MAINTENANCE FACILITY.

16. DESCRIPTORS IDENTIFIERS  
ARTIFICIAL INTELLIGENCE MULTI-SENSOR DATA FUSION  
EXPERT SYSTEMS KNOWLEDGE BASED SYSTEMS  
COMMAND, CONTROL, COMMUNICATIONS & INTELLIGENCE

17a. SECURITY CLASSIFICATION (OF REPORT) UNCLASSIFIED 17b. SECURITY CLASSIFICATION (OF PAGE) UNCLASSIFIED 17c. SECURITY CLASSIFICATION (OF ABSTRACT) UNCLASSIFIED

18. DISTRIBUTION/AVAILABILITY STATEMENT UNLIMITED AVAILABILITY 17d. SECURITY CLASSIFICATION (OF TITLES) UNCLASSIFIED

UNCLASSIFIED

**END  
FILMED**

DATE: 2-92

**DTIC**