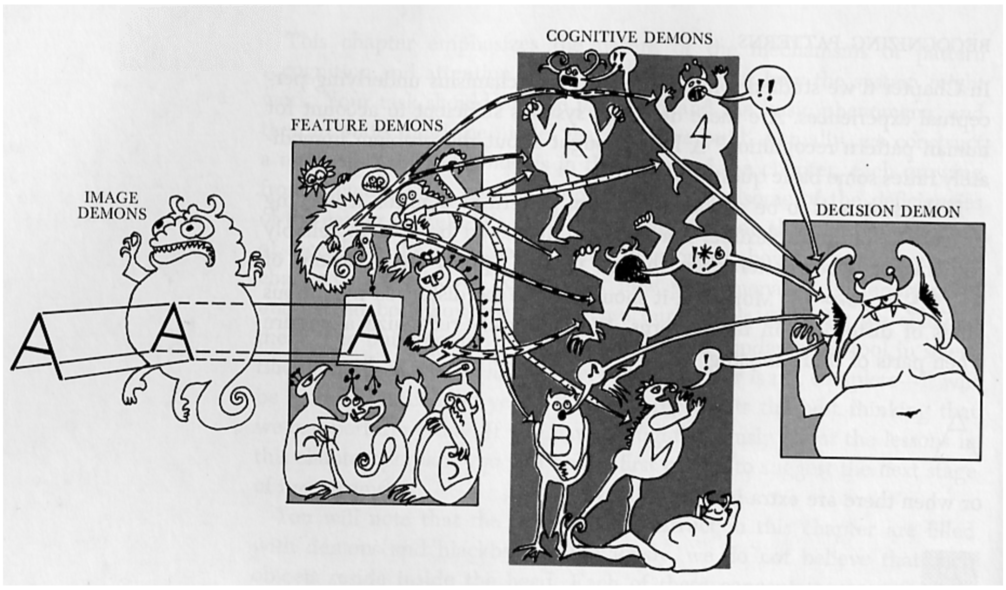
Coordination Approaches for Complex Software Systems



Tibor Bosse, Mark Hoogendoorn, and Jan Treur

vrije Universiteit amsterdam

Preface

This document presents the results of a collaboration between the Vrije Universiteit Amsterdam, Department of Artificial Intelligence and Force Vision to investigate coordination approaches for complex software systems. The project was funded by Force Vision. More in particular, the following people participated:

Force Vision:
Rob Duell, Andy van der Mee, and Bas Vermeulen.

Vrije Universiteit:
Tibor Bosse, Egon van den Broek, Mark Hoogendoorn, and Jan Treur.

Contents

1 Introduction

How to control or coordinate the processes in a complex software system is a nontrivial issue. By a component-based approach to software systems a divide and conquer strategy can be used to address the various aspects involved. This may lead to a possibly large number of components, which each can be analysed and designed independently. However, a designer may still be left with a *connectivity problem*: how can all these fragments be combined into a coherent system. One aspect of connectivity is how specific information generated by one component can become available to another component that needs that information. This is sometimes called the *data flow problem*. Such a problem is often addressed by drawing connections between components that indicate where possibly which information can transfer from one component to another one (a *data flow diagram*). Another possibility to solve this is by creating a (*shared data*) storage where information from all components is posted and which can be accessed by all components to obtain needed information. Such solutions may provide satisfactory solutions for the connectivity problem in the static sense of what the possibilities of transfer between components are. However, this does not address the dynamics in the sense of when actually such information transfer occurs, or when a component is *active* in processing its input information to generate new output information. This problem of the connectivity between components in terms of dynamics is the harder problem, sometimes called the *control problem* or the *coordination problem*. This report addresses this problem. Whilst in the literature both the terms *coordination* and *control* are used, in the remainder of this document only the former term is used.

First, in Section 2 a more detailed analysis of the problem is provided. Next, in Section 3 the methodology is described to explore what specific coordination approaches can contribute. Section 4 briefly introduces the modelling techniques and supporting software tools used (with a reference to two appendices where more details are given). In Section 5 a number of coordination approaches obtained from the literature are briefly introduced. Section 6 describes a set of test examples that can be used as input for the coordination approaches. In Section 7, 8, and 9 the simulations are shown that were undertaken to evaluate the usefulness of the coordination approaches for the test examples. Section 10 presents the results, and Section 11 is a final discussion, positioning the results achieved this far and providing further perspectives.

2 Coordination in Software Systems: The Problem

Software systems that consist of a large number of components are not easy to coordinate. One approach is to prescribe in a centralised manner when exactly a component should become active. This can be obtained, for example, by a control specification (or coordination specification) that indicates, based on results of components that already have been active, which component has to be the next active component. An example of a component-based modelling approach incorporating such a form of coordination specification (by means of socalled supervisor rules) is the first version of DESIRE (Design and Specification of Interacting Reasoning components), developed the end of the 80s and the beginning of the 90s of the previous century; cf. Langevelde, Philipsen, and Treur (1992); Gavrila and Treur (1994). However, for more complex component-based systems such a coordination specification can have a number of disadvantages:

- it can become *large* and *intransparent*.
- it may suffer from *overspecification*, i.e., the dynamics of component activation may have to be prescribed in much more detail than actually needed
- it may require quite an *effort* to acquire the coordination knowledge, i.e., find out how the coordination choices should be in all possibly occurring system states
- its *flexibility* and *adaptivity* with respect to circumstances at runtime often is limited

To remedy the first disadvantage, it is possible to add hierarchical structure to the component-based system, by clustering components to higher-level components, thus introducing aggregation levels, in such a way that at each aggregation level the number of lower-level components from which a higher-level component is composed is low. By itself this does not address the other three disadvantages. A second way to address the disadvantages above, especially the second one, is to allow components to be active in parallel and where possible leave it to a component when it becomes active, for example, in response to new input information. These two ways to address the disadvantages, by hierarchical structure and by giving more autonomy to components allowing them to be active in parallel, were worked out in the second, agent-oriented version of DESIRE developed in mid and second half of the 90s; cf. Brazier, Dunin-Keplicz, Jennings, and Treur (1995, 1997); Brazier, Treur, Wijngaards, and Willems (1998); Brazier, Jonker, and Treur (1998/2004, 2002a, 2002b).

However, while addressing the first two disadvantages in a satisfactory manner, this approach does not contribute much to address the third and fourth disadvantage. In fact, it can even make the third disadvantage worse, because opening up the possibility of more autonomy and parallel processing of components may seriously increase the number of possible states of the system: e.g., for n components, for the sequential case at any point in time there can be 1 component out of n possible components active, whereas in the parallel case there can be 1 subset of components out of 2^n possible subsets active. To overcome such problems, this report aims to contribute to solutions addressing the third and fourth disadvantage as well.

3 Methodology

To explore possibilities to address the coordination problem as described above, a general methodology has been followed that is based on the following elements (see also Figure 1):

- a number of promising *coordination approaches* from the literature are selected
- a number of *test examples* representing software component configurations are chosen
- simulations are performed where selected coordination approaches are applied to the chosen test examples, resulting in a number of *simulation traces*
- the simulation traces are evaluated (automatically) on a number of relevant *dynamic properties*

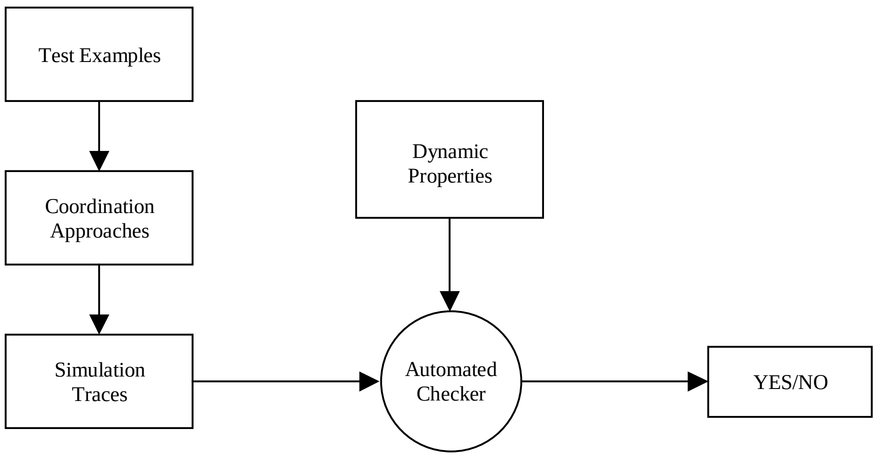


Fig. 1. General research methodology

Each of the elements of Figure 1 will be briefly discussed below.

3.1 Coordination Approaches

The problem of coordination of component-based software systems has crucial aspects in common with the problem of coordination in natural (biological), cognitive (human and animal mind) or societal systems (organisational structures). Evolution processes over long time periods have generated solutions for the coordination problem in these areas. Therefore it may make sense to analyse in more detail how these solutions work. Some literature is available that describes theories for coordination in these areas. This literature can be used as a source of inspiration to obtain more innovative approaches to coordination of complex component-based software systems. As a first step a number of such approaches will be evaluated to see whether they can overcome the problems identified in Section 2.

3.2 Test Examples

To evaluate a given coordination approach, adequate test examples of component-based software configurations are needed. One may be tempted to use a real-life component-based software system as a test example, for example, consisting of hundreds of components. However, such type of testing for one case would take a lot of effort, and to get a reasonable idea it should be repeated for a representative number of software systems at least. For this stage of the exploration this would not be appropriate. Instead, a number of smaller test examples have been identified. As a source the library of workflow patterns described in (van der Aalst et al., 2002) has been used. The examples given there have been extended with input and output data and information flow channels.

3.3 Simulation

To test the selected coordination approaches on the chosen examples, implementations have to be made. One way to do this would be to create specific implementations for each of the (abstract) test examples, by explicitly defining the internal functioning of the components involved. Next, one would add to these implementations one by one implementations of the coordination approaches, and then run each of these implementations. The resulting log data, which should include a registration of the processing time, for example, in terms of processor time or number of computation steps, can then be evaluated. Such an evaluation at an implementation level, however, has some drawbacks: the specific implementations chosen may affect the results, and the specific underlying software/hardware combination may affect the processing times measured; e.g., think of aspects of concurrency that within a software/hardware environment may have to be mapped onto a form of interleaving of processes. Therefore a different approach is chosen. All the testing is done within one given simulation environment. Within this environment, one by one the processing of a software system based on one example and one coordination mechanism is simulated. In that case, the examples are defined at an abstract level (i.e., only in terms of input-output relations, ignoring the internal functioning). The measured time then is simulated time, not processing time. In simulated time, processes can easily be active in parallel. The simulation environment chosen is logic-based, so that the simulation models are logically specified and both these models and the resulting simulation traces can be logically analysed, supported by another software environment.

3.4 Evaluation

To evaluate the resulting simulation traces, in the first place it is needed to identify the relevant properties on which such an evaluation should be based. A number of aspects can be covered in such properties. A first aspect is effectiveness or successfulness to provide the desired output for the example system. When a coordination approach does not involve the right components at the right times, and therefore is not able to generate the desired output that is possible, then it is not effective. A second aspect to evaluate is efficiency: to what extent time is wasted in the process to come up with output. A third aspect is to what extent the coordination approach is able to generate the possible activation traces one has in mind for the given example. Such properties can be formally specified and automatically checked for the simulation traces.

4 Software Environments

To support the methodology described in Section 3 two software environments are used:

- a simulation environment to specify simulation models and to execute these models in order to get simulation traces
- a checking environment to specify relevant dynamic properties of traces and to check such properties against traces

For the first, the LEADSTO environment is used (cf., Bosse, Jonker, v.d. Meij, and Treur 2005), for the second the TTL environment. In the Appendices A and B a more extensive description of these environments is shown.

4.1 The LEADSTO Simulation Environment

An important problem during system design is the validation of the design specification: can it be proven that the specification shows the expected behaviour (e.g., as described by requirements) before it is actually implemented? Especially when the specification is given in terms of abstract high-level concepts this is a non-trivial task. Simulation can be a useful method to analyse a design. In order to simulate a system to be designed, it is important to model its dynamics. LEADSTO can be used to model the dynamics of systems to be designed, on the basis of highly abstract process descriptions. If those dynamics are modelled correctly, the LEADSTO software environment can use them for simulation of the behaviour of the system. Although such simulations are no formal proof, they can contribute to an informal validation of the specification: by performing a number of simulations, it can be tested whether the behaviour of the specification is satisfactory. Depending on the domain of application, specifications of a simulation model need to be formulated quantitatively or qualitatively. Usually, within a given application explicit boundaries can be given in which the mechanisms take effect. For example, “from the time of planting an avocado pit, it takes 4 to 6 weeks for a shoot to appear”. When considering current approaches to modelling dynamics, the following two classes can be identified: logic-oriented modelling approaches, and mathematical modelling approaches, usually based on difference or differential equations. Logic-oriented approaches are good for expressing qualitative relations, but less suitable for working with quantitative relationships. Mathematical modelling approaches (e.g., Dynamical Systems Theory), are good for the quantitative relations, but expressing conceptual, qualitative relationships is very difficult. The LEADSTO language (and software environment) is a language combining the specification of qualitative and quantitative relations.

The LEADSTO format is an executable format that can be used to obtain a specification of a simulation model in terms of executable dynamic properties. The format is defined as follows. Let α and β be state properties of the form ‘conjunction of literals’ (where a literal is an atom or the negation of an atom), and e, f, g, h non-negative real numbers. In the LEADSTO language $\alpha \twoheadrightarrow_{e, f, g, h} \beta$, means:

if	state property α holds for a certain time interval with duration g,
then	after some delay (between e and f) state property β will hold for a certain time interval of length h.

A specification of dynamic properties in LEADSTO format has as advantages that it is executable and that it can often easily be depicted graphically. For more details, see Appendix A.

4.2 The TTL Analysis Environment

For the purpose of specification and execution of a simulation model, the limited format of the LEADSTO language is adequate. However, to analyse the more complex dynamic properties that emerge from such direct, executable temporal relationships, for example, in simulations, a more expressive language is needed. The Temporal Trace Language (TTL) is such a more expressive language for the analysis of dynamic properties. This predicate logical language supports formal specification and analysis of dynamic properties covering both qualitative and quantitative aspects. A special software environment has been developed for TTL, featuring both a Property Editor for building and editing TTL properties and a Checking Tool that enables the formal verification of such properties against a set of (simulated or empirical) traces.

For the TTL properties informal, semi-formal or formal representations can be given, briefly defined as follows. A state ontology is a specification (in order-sorted logic) of a vocabulary. A state for ontology Ont is an assignment of truth-values {true, false} to the set At(Ont) of ground atoms expressed in terms of Ont. The *set of all possible states* for state ontology Ont is denoted by STATES(Ont). The set of *state properties* STATPROP(Ont) for state ontology Ont is the set of all propositions over ground atoms from At(Ont). A fixed *time frame* T is assumed which is linearly ordered. A *trace* or *trajectory* γ over a state ontology Ont and time frame T is a mapping $\gamma : T \rightarrow$ STATES(Ont), i.e., a sequence of states γ_t ($t \in T$) in STATES(Ont). The set of all traces over state ontology Ont is denoted by TRACES(Ont). Depending on the application, the time frame T may be dense (e.g., the real numbers), or discrete (e.g., the set of integers or natural numbers or a finite initial segment of the natural numbers), or any other form, as long as it has a linear ordering. The set of *dynamic properties* DYNPROP(Ont) is the set of temporal statements that can be formulated with respect to traces based on the state ontology Ont in the following manner.
Given a trace γ over state ontology Ont, the input state of some component c at time point t is denoted by state(γ, t, input(c)); analogously state(γ, t, output(c)) and state(γ, t, internal(c)) denote the output state and internal state.

These states can be related to state properties via the formally defined satisfaction relation |=; i.e., state(γ, t, output(c)) |= p denotes that state property p holds in trace γ at time t in the output state of component c. Based on these statements, dynamic properties can be formulated in a formal manner in a sorted first-order predicate logic with sorts TIME or T for time points, Traces for traces and F for state formulae, using quantifiers over time and the usual first-order logical connectives such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$.

5 Coordination Approaches

As mentioned earlier, the coordination problem in software systems has crucial aspects in common with the problem of coordination in natural (biological), cognitive (human and animal mind) or societal systems (organisational structures). Therefore, a large body of literature is available that describes coordination approaches in these areas. In this section, some of the most well-known approaches are discussed. Section 5.1 focusses on the Behavior Networks approach by Patty Maes. Section 5.2 describes Ornstein (1986)'s structure of the Multi-Mind. Section 5.3 describes Minsky (1985)'s Society of the Mind theory. Section 5.4 describes Selfridge (1958)'s Pandemonium model, and Section 5.5 addresses the classifier combination techniques known as *voting methods*. All sections, with the exception of Section 5.5, are summaries on the basis of Franklin (1997).

5.1 Behavior Networks

Behavior networks have been introduced by Pattie Maes in 1989. Behavior networks contain *competence modules* which can be seen as components or agent. They are interactive, mindless agents, each with a specific competence. The question is how the behavior of these modules can be coordinated. One option is to hardwire or hand-code the coordination. This strategy works well only for systems with simple, non-conflicting goals in a relatively static and not too complex environment like a thermostat, and assembly line robot or a toy AI system playing tic-tac-toe. Some military agencies often rely on a hierarchical coordination structure, and symbolic AI systems employ this strategy as well. Some expert system shells implement algorithms that, given a set of input-output pairs as examples, produce an appropriate decision tree. A hierarchical strategy often suffers rigidity, working well until an unusual circumstance is encountered, then crashing. In Jackson's (1987) work, a distributed system of coordination is present, where strength decide who gets to act. Maes takes the same approach. A competence module looks very much like a production rule, each having some preconditions. Each module also consists of lists of additions and of deletions, that is, statements the module wants to add to the global database or statements it wants to delete. The difference between a competence modules and a production rule is the presence of an activation, a number indicating some kind of strength level.

5.1.1 Algorithm

To illustrate the algorithm, one can think of each competence module as occupying a node of a digraph. The links between the nodes are completely determined by the competence modules. If a competence module X will add a proposition b which is on competence Y's precondition list, then put a successor link between X and Y. There might be several such propositions resulting in several links between the same nodes. Next, whenever you put a successor going one way, put a predecessor going the other. Finally, suppose you have a proposition m on competence Y's delete list that is also a precondition for competence X, draw a conflictor link from X to . So how can we use this digraph? First of all, the underlying digraph spreads the activation. The activation comes from the activation stores by the competence modules themselves, from the environment, and from goals. The system has only built in global goals. So let's describe the sources of spreading activation over the system. The environment awards activation to a competence module for each of its true preconditions. The more true preconditions a competence has, that is, the more relevant it is to the current situation, the more activity it's going to receive from the environment. This source of activation allows the system to be opportunistic. Next, each goals awards activation to each component that, by being active, will satisfy the goal. In other words, if the competence includes a proposition on its add list that satisfies a goal, then this goal will send activation to that competence. Protected goals are also possible: A completed goal inhibits any competence that will undo it. Finally, activation is spread from competence to competence along links. Along successor links, one competence strengthens those competences whose preconditions it can help fulfill, it does so by sending them activation along the link. Along predecessor links, one components strengthens any other competence whose add list fulfills one of its preconditions. A competence sends inhibition along a conflictor link to any other competence that can delete one of its true precondition, thereby weakening it. Every conflictor link is inhibitory. A competence is called executable when all of its preconditions can be satisfied. The algorithm in pseudo code:

```
Loop forever
   1. Add activation from environment and goals
   2. Spread activation forward and backward among competence
      modules
   3. Decay – total activation remains constant
   4. Competence module fires if
         a. it’s executable
         b. it’s over threshold
         c. it’s the maximum such
   5. If one competence module fires, its activation goes to 0,
      and all thresholds return to their normal value
   6. If none fires, reduce all thresholds by 10%
```

5.1.2 Mathematical model

Maes uses a mathematical model to calculate the activations and how it spreads. This text originates from Maes 1989. Let a competence module i be defined by a tuple (c_i, a_i, d_i, α_i) where c_i is the list of preconditions that need to be fulfilled before the competence module can become active, a_i and d_i represent the expected effects of the activation of the module in the form of an add and delete list, and each competence module has a level of activation α_i. Furthermore, the following elements are assumed to be given:

- A set of competence modules 1....n,
- A set of propositions P,
- A function S(t) returning the propositions that are observed to be true at time t; S is implemented by an independent process (or the real world),
- A function G(t) returning the propositions that are the goals of the agent at time t; G is again implemented by an independent process,
- A function R(t) returning the propositions that are a goal of the agent that have already been achieved at time t, R is again implemented by an independent process,
- A function executable(i, t) which returns 1 if competence module i is executable at time t (i.e. if all of the preconditions of competence module i are members of S(t)), and 0 otherwise.
- A function M(j), which returns the set of modules that match proposition j, i.e., the modules x for which $j \in c_x$
- A function A(j) which returns the set of modules that achieve propositions j, i.e. modules x for which $j \in a_x$
- A function U(j) which returns the set of modules that undo proposition j, i.e. modules x for which $j \in d_x$
- π, the mean level of activation,
- θ, the threshold of activation, where is lowered 10% every time no module could be selected, and is reset to its initial value whenever a module becomes active,
- ϕ, the amount of activation energy injected by the state per true proposition,
- γ, the amount of activation energy injected by the goals per goal,
- δ, the amount of activation energy taken away by the protected goals per protected goal.

Given competence module x = (c_x, a_x, d_x, α_x), the input of activation to module x from the state at time t is:

$$input\_from\_state(x,t) = \sum_j \phi \frac{1}{\#M(j)} \frac{1}{\#c_x}$$

where $j \in S(t) \cap c_x$ and where # stands for the cardinality of a set.

The input of activation to competence module x from the goals at time t is:

$$input\_from\_goals(x,t) = \sum_j \gamma \frac{1}{\#A(j)} \frac{1}{\#a_x}$$

where $j \in G(t) \cap a_x$.

The removal of activation from competence module x by the goals that are protected at time t is:

$$taken\_away\_by\_protected\_goals(x,t) = \sum_j \delta \frac{1}{\#U(j)} \frac{1}{\#d_x}$$

where $j \in R(t) \cap d_x$.

The following equation specifies what a competence module x = (c_x, a_x, d_x, α_x) spreads backward to a competence module y = (c_y, a_y, d_y, α_y):

$$spreads\_bw(x,y,t) = \begin{cases} \sum_j \alpha_x(t-1) \frac{1}{\#A(j)} \frac{1}{\#a_y} & \text{if } executable(x,t) = 0 \\ 0 & \text{if } executable(x,t) = 1 \end{cases}$$

where $j \notin S(t) \wedge j \in c_x \cap a_y$.

For forward spreading from module x to y the following definition is used:

$$spreads\_fw(x,y,t) = \begin{cases} \sum_j \alpha_x(t-1) \frac{\phi}{\gamma} \frac{1}{\#M(j)} \frac{1}{\#c_y} & \text{if } executable(x,t) = 1 \\ 0 & \text{if } executable(x,t) = 0 \end{cases}$$

where $j \notin S(t) \wedge j \in a_x \cap c_y$.

The following equation specifies what module x takes away from module y:

$$takes\_away(x,y,t) =$$

$$\begin{cases} 0 & \text{if } (\alpha_x(t-1) < \alpha_y(t-1)) \wedge (\exists i \in S(t) \cap c_y \cap d_x) \\ max(\sum_j \alpha_x(t-1) \frac{\delta}{\gamma} \frac{1}{\#U(j)} \frac{1}{\#d_y}, \alpha_y(t-1)) & \text{otherwise} \end{cases}$$

where $j \in c_x \cap d_y \cap S(t)$.

The activation level of a competence module y at time t is defined as:

$$\begin{aligned} \alpha(y,0) &= 0 \\ \alpha(y,t) &= decay(\alpha(y,t-1)(1 - active(y,t-1)) \\ &\quad + input\_from\_state(y,t) + input\_from\_goals(y,t) \\ &\quad - taken\_away\_by\_protected\_goals(y,t) \\ &\quad + \sum_{x,z}(spreads\_bw(x,y,t) + spreads\_fw(x,y,t) - takes\_away(z,y,t))) \end{aligned}$$

where x ranges over the modules in the network, z ranges over the modules in the network minus the module y, t > 0, and the decay function is such that the global activation remains constant:

$$\sum_y \alpha_y(t) = n\pi$$

The competence module that becomes active at time t is module I such that:

$$active(t,i) = 1\ if \begin{cases} \alpha(i,t) >= \theta & (1) \\ executable(i,t) = 1 & (2) \\ \forall j\ fulfilling(1) \wedge (2) : \alpha(i,t) >= \alpha(j,t) & (3) \end{cases}$$

$$active(t,i) = 0\ otherwise$$

5.2 The Multi-Mind

Ornstein (1986) claims that we are not a single person but we are many. Our strong subjective sense of a single, unified, conscious agent controlling life's events with a singular integrated purpose is only an illusion. It is illusory to think that a person has a single mind. Rather, there are many. Ornstein sees the mind as being composed of different kinds of small minds. We have lots of these minds that are specialised to handle different chores. These different entities are wheeled into consciousness, then usually returned to their place after use, and put back on the shelf. The memories from Ornstein's perspective are more like data structures that are retrieved. The conscious mind wheels in various small minds, which accomplish quite limited and specific purposes. This wheeling in and out of small minds allows for diverse centers of control. Ornstein speaks of centers of control at lower levels having developed over millions of years to regulate the body, to guard against danger, to organise and plan efforts, and so on. These various centers have different priorities; some are more important than others.

Ornstein identifies four strong tendencies or patterns of behavior:

- *What have you done for me lately?* More sensitivity for more recent information.
- *Don't call me unless anything new and exiting happens.* Unexpected or extraordinary events seem to enjoy a fast track in our consciousness.
- *Compared to what?* We constantly judge by comparisons and rarely make absolute judgments of any kind.
- *Get to the point.* The meaning of any event, its relevance to the person (or the autonomous agent), is the point.

Ornstein claims that we throw out almost all the information that reaches us. The theory of Ornstein can be seen as a high-level theory that will give a framework within which to view the work on mechanisms.

Ornstein states that our world appears to us the way it does because we are built the way we are. The world we create is also affected by internal transformations, an example is the difference in perception of increase in electric shocks (exponent > 1) and brightness (exponent < 1).

The structure of the multi-mind according to Ornstein is as follows. At the lowest level of organisation are the basic neural transformations. These can be pictured as groups of neurons acting cooperatively to perform a set function. Then come domain-specific data-processing modules, the quick and stupid analytical systems of the mind, one of which might produce the consistent perception of red under bright sunlight and dusk. Slower, but smarter, more general, and more flexible, are the talents. Combinations of talents, useful in particular situations, comprise small minds. And finally, at the top of the heap, rests consciousness, into which small minds are wheeled as our goals and environment demand. One question that's still left is who is in control of the wheeling. Ornstein postulates a governing self that controls the wheeling of small minds in and out of consciousness. In most of us, which small mind gets wheeled in is decided automatically on the basis of blind habit. However, a person can become conscious of the multi-mind and begin to run *them* instead of hopelessly watch anger wheel in once again.

5.3 Society of the Mind

In Minsky (1985), Marvin Minsky motivates his theory of mind from an evolutional perspective: each human cranium contains hundreds of kinds of computer, developed over hundreds of millions of years of evolution each with a somewhat different architecture. Contrary to Ornstein, Minsky takes the bottom up approach. He wants to show how you can build a human mind from many little parts, each mindless by itself. He refers to these little parts as agents. Each mental agent by itself can only do some simple things that needs no mind or thought at all. Yet when these agents are joined in societies this leads to true intelligence. Minsky's agents care all processes, even when they empower memory. Basically, Minsky is trying to sell the idea of intelligence implemented by a society of relatively unintelligent agents. Agents can call on other agents as procedures can in a programming language. The chance of picking two agents randomly out of the human mind and their having anything whatever to say to one another is vanishingly small. Each agent only uses a small number of others with whom it can communicate. If you look at the brain, the lack of communication between agents seem plausible.

5.3.1 Minsky’s mechanisms

Minsky proposes many possible mechanisms of mind, most of them at a relatively high level of abstraction with many possible implementations.

The first involves the use of collections of agents for the representation of concepts. Agents can either be active or not, or may have some activation level other than on or off. Active agents can represent properties or features, for example representing a certain shape, substance, color, and size. When for example another shape becomes active, different agents in the shape division will become active. Question that remains is how these representations are retrieved. Minsky introduces the notion of a K-line as the basic mechanism for memory. A K-line is a mental data structure and also an agent. It connects other agents and awakens them when appropriate. You can think of a K-line as a wire-like structure that attaches itself to agents that are active when the K-line is formed. A K-line representing the sentence “Jack flies a kite” is shown in Figure 2.

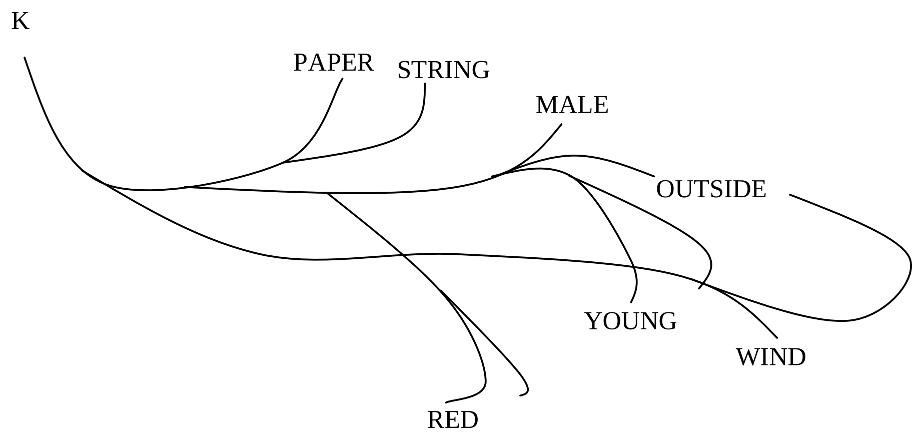


Fig. 2. Example K-line

K-lines can be directly hooked to agents as shown above, but also to other, preexisting K-lines as shown in Figure 3.

Another important question is how these mechanisms are controlled. Minsky proposes a B-brain influencing an A-brain, that in turn, interacts with the world. Picture the A-brain as being comprise of agents that sense the outside world and of other, motor agents that act upon it. The B-brain is sitting atop only in contact with A’s agents and is composed out of executives who direct, or at least, influence A’s activity. Some examples: if A seems to be repeating itself, caught in an endless loop, B makes it try something new. If A does something B likes, B makes A remember it. If A is too much involved with detail, B makes it take a higher-level view, and conversely. The notion of a B-brain provides a high-level, abstract coordination mechanism. Both brains are composed hierarchically. At the bottom are the agents with their own hierarchy. At the next level up we find societies, organisations of agents. Up another level you have layers of societies. Minds according to Minsky develop as sequences of layers of societies. Each new layer begins as a set of K-lines and learns to exploit whatever skills have been acquired by the previous layer. When a layer acquires some useful and substantial skill, it tends to stop learning and changing.

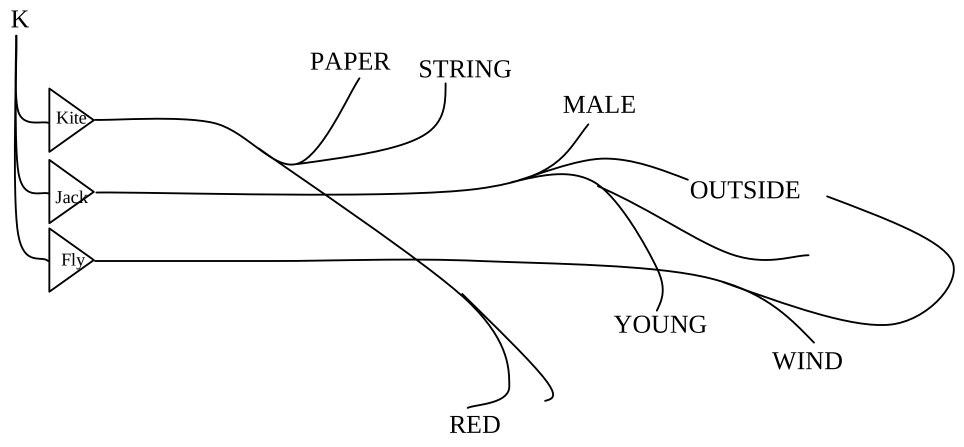


Fig. 3. Example K-line, hooked to other K-lines

Minsky refers to another possible mechanism of mind, at least for some high-level agents, as a difference engine. A comparison of the current situation with a description of the goal it wants to reach provides a set of differences. Agents acting upon the world so as to minimise these differences are then activated, thus moving the situation towards the goal. The strategy is referred to, in symbolic AI, as means-end analysis. Means-end analysis is a "weak" method in that it requires little domain knowledge to accomplish its objective. Difference engines require goal descriptions. Goals must persist over some time, and require some image or description of a desired state.

Now suppose a procedure has failed in a certain situation. Fixing it might introduce errors in other situations where it now works perfectly well. Minsky suggests inserting a censor that remembers some abstraction of the situation in which the procedure doesn't work for this purpose. When that situation arises again, the censor suppresses the misbehaving procedure and calls on some other, special purpose, procedure to do the job.

Another element is learning new behavior, the old behavior must remain while the learning process is ongoing. Therefore, the old system is kept intact and operational while building the new as a detour around the old. The system can be tested without letting it assume control. When satisfied, cut or suppress some of the connections of the older system.

Conflicting goals can also be present when looking at the agents within the mind. Minsky claims that such conflicts among our most insistent goals produce strong emotional reactions. These emotions are needed to defend against competing goals. He concludes that the question is not whether systems can have any emotions, but whether machines can be intelligent without any emotions.

Finally, Minsky introduces the idea of accumulation. Each agency will accumulate under a wide variety of agents to do its bidding, so as to have several different ways of getting its job done. Some of these might be more efficient than others, but if one is lost in a particular circumstance, chances are there will be another way.

5.4 Pandemonium

Selfridge (1958) proposed a pandemonium theory of perception, built on primitive constructs called *demons*. A demon is a rule, procedure or agent in Minsky's sense. In computer science, demons are processes sitting around for something specific to happen. Selfridge uses demons to identify objects, the one from a crowd of demons shouting loudest is taken to identify this particular object. Jackson extends this idea to a theory of mind. He identifies demons involved in perception but also demons that cause external actions and demons that act internally on other demons. These classes need not be disjoint; a single demon may, for example, affect an action while influencing some other demon as a side effect. These demons can be seen as an abstraction of Minsky's agents. Now picture these demons living in a stadium. Almost all of them are up in the stands; they're the crowd cheering on the performers. Six are down on the playing field, exciting the crowd in the stands. Demons in the stands respond selectively to these attempts to excite them. Some are more excited than others; some shout louder. The loudest demon in the stands gets to go down, and join those on the field, displacing one of those currently performing back to the stands. The loudness of the shouting of a demon is dependant upon being linked with the demon that must excite. Stronger links produce louder responses. These links are created in the following way. Initially, the system starts off with a certain number of initial demons and initial, built-in links between them. New links are made between demons and existing links are strengthened in the proportion to the time they have been together on the field, plus the gain of the system (when all is going well, the gain is higher). In addition, a sub arena is present that performs a number of tasks. First of all, it measures the system's well-being so that "improved conditions" can be discerned, and turns the gain up or down. The sub arena also performs sensory input by sending demons representing low-level input to the playing field, providing an interface between the actual sensory input and whatever the system does with it. Demons also represent low-level actions that are carried out by the sub arena at the command of action demons on the playing field.

A visualisation of the pandemonium theory, instantiated for the domain of pattern recognition, is shown in Figure 4. Here, the goal of the pandemonium is to recognise a certain letter (the letter R in this case). Different feature demons are responsible for identifying certain characteristic features of the letter. Next, cognitive demons use this information to determine how loud they will shout. Each cognitive demon corresponds to a single letter. Finally, a decision demon decides which cognitive demon shouts loudest, and the corresponding letter is selected.

5.4.1 Concepts via Pandemonium

Jackson also allows for the creation of concepts in the system. Demons that have appeared together frequently can be merged into a single concept demon. When concept demons are created, their component demons survive and continue to do their things. Concept demons help overcome the bottleneck of the limited playing field. Concept demons can be grouped into compound concept demons. To model dreaming, simply turn off the sub arena interference, especially the external sensory channels.

Decay is also built into the system. Unused links decay, or lose strength at some background rate. Negative links may decay at a different rate. High-level demons enjoy a slower decay rate.

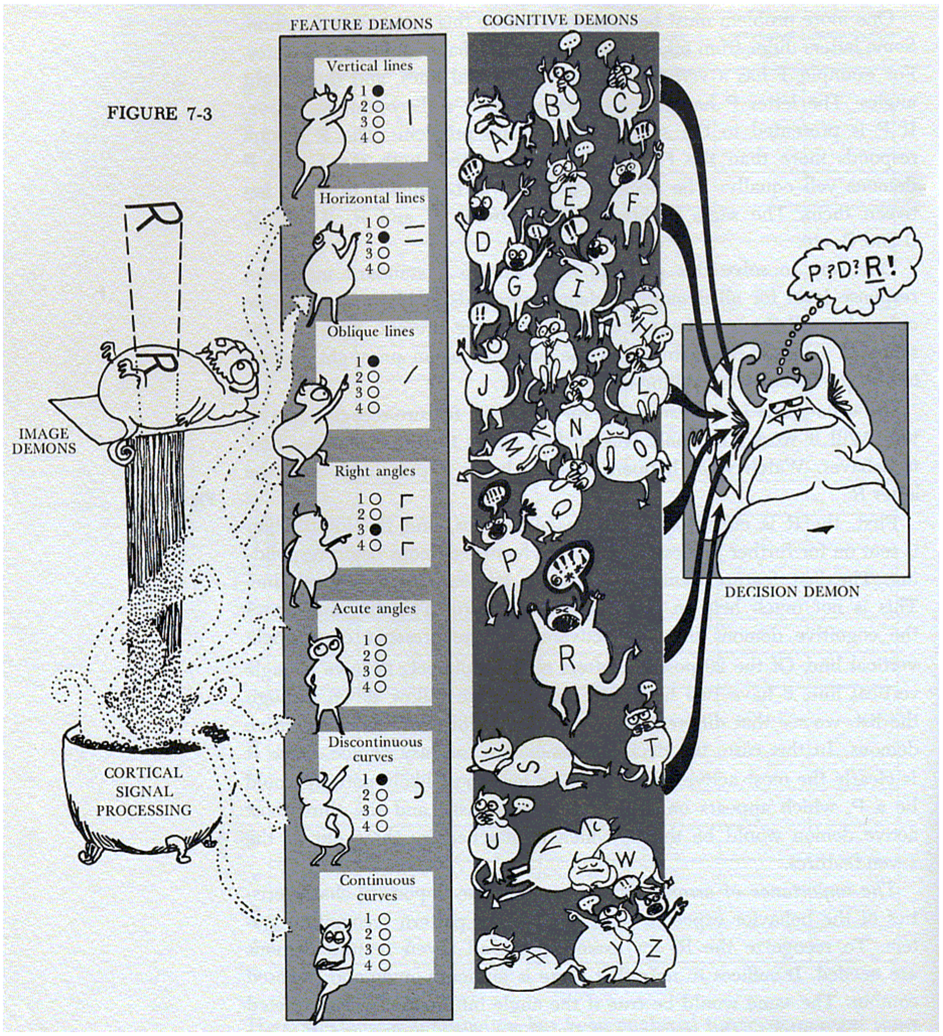


Fig. 4. Visualisation of the Pandemonium theory
(from Lindsay and Norman, 1977, p. 266)

5.4.2 A Computing Perspective on Pandemonium

Jackson (1987) states that this pandemonium system avoids the major pitfalls of parallel and serial computing by combining their better features. Serial machines are often too slow, and at any given time are actively using only a fraction of their available hardware. Parallel machines can be faster and make more efficient use of their hardware. *But they often spend much of their time communicating between one processor and another*. This system combines the best of the two worlds according to Jackson. First, it scans the demons in the crowd in parallel to determines the loudest. After that the demons are executed into a single threat on the playing field.

5.5 Voting

In this section, the classifier combination techniques known as *voting methods* will be discussed. Voting methods are simple algorithms that can be used to combine classifier outputs. However, as will be illustrated in this section, voting methods can be used to combine the output of agents. They are called voting methods as most are direct derivatives of techniques used in elections. By replacing the electorate by components (or agents) and the candidates by the possible activations of components, it is very simple to apply the voting methods to coordination problems in component-based software systems (or multi-agent systems). The major advantage of the voting methods lies in their elegant simplicity. Furthermore, the voting methods tend to have a very acceptable performance rate.

Voting methods can be explained as follows. Consider an agent *A* as a function that assigns a value *v* to each possible class *c* depending on the data sample *d* of which the correct class is sought, i.e., *A(d,c) = v*. The value *v* that the agent returns can represent, for example, the chance the agent assigns to *c* of being the correct class for *d* (also called a confidence value) or a simple boolean (represented by 0 and 1) to indicate whether *c* is the top-favorite of *A* or not. A voting algorithm *V* consist of one or two elementary arithmetic functions that are applied per class *c* to all the values v_i assigned by all the agents A_i to *c*. For example, if *V* is the voting method known as the product rule then V(d,c) = (i Ai(d,c). The agents are then ordered by the value assigned to them by V and the top class (or bottom class, depending on V) is returned as the matching class for d. Note that a voting method also defines the type of values v it uses as input. For instance, when comparing the voting methods of the Borda count and the sum rule, the major difference does not lie in their applied algorithm (both are more or less characterised by V(d,c) = (i A_i(d,c)), but on the input they use. The sum rule uses the afore-mentioned confidence value, while the Borda count uses the rank assigned to all the classes by the agents. This also implies that some voting methods cannot use the output of some agents. For example, the sum rule cannot use the output of an agent that assigns only ranks such as the Borda count would use.

In pattern recognition research, voting methods are often used when the late-fusion part of a classifier combination is of little importance. This includes, among others, research into ensemble creation. In turn, the wide use of voting methods has generated interest into these methods themselves. The relationship between several of the voting methods has been researched in order to find to one with the best performance (e.g., see Kittler and Alkoot (2001) and Kuncheva (2002)). Other research has focused on finding alternative versions of some of the voting methods. These efforts aim to increase the flexibility of the methods. Examples include Ho, Hull and Srihari's adaptation of the Borda Count (1994) and the weighted voting variants (Lam and Suen (1995) and Günter and Bunke (2004)).

As is apparent from the last line of research, voting methods can be improved upon. Voting methods are simple and, as a result, are not well equipped for variance. This variance can stem from the amount, format and importance of the input sources. Sometimes classifiers, but especially agents, which are the input sources of voting methods, are able to provide more information than that can be used in voting methods. This may result a loss of information, which in turn implies a potential loss of performance. Yet, how serious is this loss of information and what is its impact on the performance? After all, the striking simplicity of voting methods makes them so easy to use that it may warrant a small dip in performance. The diversity of voting methods and the degree of information they use offer a perfect means to explore this trade-off between needed knowledge and performance on the one side and ease of use and simplicity on the other.

This section will delve into this issue and describes what problems frequently arise if the simple voting methods are used on complex multi-agent systems.

5.5.1 Voting Methods for Pattern Recognition

In agent research, there is a growing use of multi-agent systems with the goal to increase recognition performance. In many cases, plurality voting is a part of the combination process. In this section, we discuss several well known voting methods from politics and economics on agent combination in order to see if an alternative to the simple plurality vote exists. We found that better methods are available, that are comparatively simple and fast.

Introduction

The area of multi-agent systems has rapidly established itself as a major topic in the agent community (Dietterich (2000), Kittler and Alkoot (2001), and Xu et al. (1992)). In this section, we will explore and evaluate the application of several well-known voting methods on combining multiple-classifier hypotheses.

In pattern recognition, two forms of classifier combination exist: the multi-stage, hierarchical (Alpaydin et al. (2000) and Vuurpijl and Schomaker (2000)) methods and the ensemble (or late fusion) (Dietterich (2000) and Kuncheva (2002)) methods. In the first approach, the classifiers are placed in a multi-layered architecture where the output of one layer limits the possible classes, or chooses the most applicable classifier, in the next layer. The second approach explores ensembles of classifiers, trained on different or similar data and using different or similar features. The classifiers are run simultaneously and their outputs are merged into one compound classification. In most cases, this combination of output hypotheses is done by using the simplest of voting methods (plurality vote; often erroneously called "majority vote"), though more elaborate combination schemes have been proposed (e.g., Dempster-Schafer, BKS and DCS (Giacinto and Roli, 2000). Plurality voting is mostly used in classifier combination, as it is simple and yields acceptable results. However, we will show that there exist alternative, and sometimes better voting methods, par excellence suitable for multi-agent systems. The concept of voting is well-known from politics and economics, where multiple opinions shared by people must be merged into one final decision. Many different voting methods stem from these areas, which are all relatively simple to perform but use different amounts of information. In this section, we will present and discuss the best known of these voting methods that are suitable for application in classifier combination. The performance of these voting methods will be assessed by combining various ensembles of classifiers.

In the next section, an overview and discussion is presented of the voting methods. Followed by an introduction of bagging, a method to test the effect of different voting methods.

Voting methods

In this section, the voting methods will be presented. We will start in next Section with a general overview of voting methods and their application to combining agents. Next, the actual voting methods will be discussed conform three distinguishable classes: unweighed voting methods, confidence voting methods, and ranked voting methods.

General overview

In human society (as 'natural' multi-agent systems), voting is a formal way of expressing opinions. A well-known example is the *election* of a president. In this example, *voters* are the people that express their opinion by means of a *vote*. When voting, a voter chooses one of the *candidates* or indicates some kind of rank-order which indicates his preference. The *voting method* is the mechanism of integrating all votes into one final decision. The *winner* is the candidate that is chosen as result of the voting method.

Here, we would like to show how voting is translated for use in multi-agent systems. Now, the agents are the voters, the possible classes are the candidates and an election is the of one sample. This produces a winner, which is the resulting decision made for the sample by the ensemble of agents. The actual voting depends on the voting method used, but an agent expresses its opinion simply by classifying a sample. The result of this classification, be it a single class, a ranked list of all classes, or even a scored list of the classes, can be interpreted as a vote. The voting methods are simple, formal, step-by-step methods (see the next section). So, implementing them, given the translation above, is straightforward.

Actually, the process of voting by agents is simpler than the process of voting by humans. Agents are programmed to classify a sample independent of the results of other agents.

Therefore, current agents will not alter their results in order to use the voting method to the benefit of a preferred class, as a human might do. In other words, an agent does not cheat (yet).

Unweighed voting methods

The unweighed voting methods consist of methods in which each vote carries equal weight. The only differentiation between the candidates is the number of votes they have received. As a consequence, voters cannot express the degree of preference of one candidate over another. Although this removes relevant information, it also results in less complex methods because no elaborate measures need to be taken to limit the power of the voter when expressing degrees of preference. Another drawback is the larger chance on a tied result. With the lack of extra information this can only be solved by
a random draw. Three of the voting methods presented here (*amendment*, *run-off***,** and *Condorcet*) are multi-step procedures. These methods require that the agents are able to give a preference choice between any two given classes. This makes these three voting methods more difficult to apply than other unweighed voting methods. It might be argued that the multi-step methods should be placed under the ranking methods, but the separate steps are inherently unweighed voting, so they are discussed here.

In terms of multi-agent systems, the (single step) unweighed voting methods demand no prerequisites from the agents, but also do not use any extra information the classifiers may provide. The multi-step methods expect the agents to be able to handle two-class subdomains of a larger population of classes.

Plurality: Also known as 'first past the post', plurality is the simplest form of voting. Every voter has one vote, which it can cast for any one candidate. The candidate with the highest number of votes wins. The benefits of this method are its simplicity and ease of use. The major drawback of plurality voting is the real possibility of a win on a small number of votes and thus of a minority (and very probably an erroneous) winner.

Majority: In majority voting every voter has one vote that can be cast for any one candidate. The candidate that received the majority (i.e., more than half) of the votes, wins the election. Note that majority voting is often confused with plurality voting in which no majority is needed to win. The benefits of this method are its simplicity and its low error count. The method only appoints a winner in case of a majority candidate, so in order to produce an error the majority of the agents has to be wrong. The chances of this happening are low, especially with a large number of agents. However, the downside is that when no majority candidate is present, no result is produced and the sample is rejected by the voting method.

Amendment vote: Amendment voting starts with a majority vote between the first two candidates. The winner of that election is pitted against the next available candidate and so on until the one remaining candidate is declared the winner. This voting method is favourable for the candidates that are added last in the total election. This lack of neutrality should be recognised when using this voting method.

Runoff vote: The runoff vote is a two step voting process. In the first step each voter can vote for any one candidate. The two candidates with the highest number of votes advance to the next round. The second round is a majority vote between these two candidates in which all voters can participate again. The runoff vote solves the biggest problem of the plurality vote and has no rejections like the majority vote at the cost of a slight decrease of transparency. It will always deliver a winner and the chances of electing a minority candidate have decreased considerably.

Condorcet count: In this method, all candidates are compared in pairwise elections. The winner of each election scores a point. The candidate with the highest number of points wins the total election. This method is more complex then the other unweighed voting methods, but also suffers least from the problems of these methods.

Confidence voting methods

In confidence voting methods, voters can express the degree of their preference for a candidate. This is done by assigning a value (called the confidence value, hence the name for these voting methods) to candidates. The higher the confidence value, the more the candidate is preferred by the voter. Examples of confidence scores are probabilities and distances. The prerequisite for using these voting methods in multi-agent systems is not only that agents produce such a confidence value, but also that these confidence values are scaled correctly. So questions like "is there a limit to the confidence value or will any number do?" and "how does one proportionally correctly translate a preference for a candidate in a value?", should be answered.

Pandemonium: Every voter is given one vote, which it can cast for any one candidate. The voter casts the vote by stating its confidence in the candidate. The candidate which received the vote with the highest confidence of all votes cast wins. This method, known as Selfridge's Pandemonium (Selfridge, 1958), is one of the very first examples of using separate experts/agents in computer science. It is very simple, but misses the possibility for a voter to express differences of preference between candidates. Only the voter's top choice and its confidence are known. Furthermore, there is no limit to the amount of confidence nor a scale for voter's to adhere to. While limits are easily added to the method, a correct scale is still difficult to implement. However, with well scaled classifiers, this method could be sufficient.

Sum rule: When the sum rule is used each voter has to give a confidence value for each candidate. Next all confidence values are added for each candidate and the candidate with the highest sum wins the election.

Product rule: Like with the sum rule, each voter gives a confidence value for each candidate. Then all confidence values are multiplied per candidate. The candidate with the highest confidence product wins. The product rule is highly subjective to low confidence values. A very low value can ruin a candidate's chances on winning the election no matter what its other confidence values are.

Ranked voting methods

In ranked voting methods, the voters are asked for a preference ranking of the candidates. This way, more information on the voter's preference is used than in the unweighed voting methods. On the other hand, it does only convey the degree of preference between two classes in fixed amounts (the ranks) instead of the confidence values of the confidence vote methods. This constitutes a loss of information, though it is easier in use (no problems in scaling the voters confidences) and it prevents over-confidence in voters (see also Ho et al. (1994)). Ranked voting methods are useful in classifier combination if the classifiers can give some kind of confidence value that is hard to scale correctly.

Borda count: This method, developed by Jean-Charles de Borda (1781), needs a complete preference ranking from all voters over all candidates. It then computes the mean rank of each candidate over all voters. The classes are reranked by their mean rank and the top ranked class wins the election. Note that the Borda count is the ranked variant of the sum rule.

Single transferable vote (STV): Also known as alternative voting (in case of one winner situations), each voter gives a preference ranking of the candidates. Incomplete ranks are possible, though it may result in a voter losing his vote altogether. A majority vote is held based on the highest ranked candidate of each voter's preference ranking. If some candidate gains the majority, it wins the election. Otherwise, the candidate with the least number of votes in the majority election is eliminated from further participation. This candidate is removed from all preference rankings. Now, the process repeats itself, starting with the majority vote, until one candidate gains the majority. One low rank in an STV election is less disruptive for a candidate's chances of winning then in the Borda count. However, due to the elimination procedure, complex and illogical side effects may occur (e.g. voting for a candidate may result in its loss of the election).

Bagging

To test the effect of the different voting methods when combining outputs of agents, the technique of *bagging* can be used (Breiman, 1994). Bagging is a simple method known from pattern recognition to increase the recognition performance of a classification technique that depends on training the classifier. Bagging consists of the following steps:

1. New training sets were created by randomly sampling with replacement from the original training set. A number of training sets between 10 and 20 is sufficient (Breiman, 1994). The number of samples in each training set is normally equal to the number of samples in the original training set. Note that the number of different samples is probably smaller as doubles are possible (and even very likely).
2. For each new training set, train a classifier using a consistent technique. The bagged classifier is now complete.
3. For classification, each sample is classified by all classifiers.
4. When the classifiers return their results for a sample, these results are then combined using a plurality vote.

5.5.2 Variants of the Borda Count Method

In all fields of pattern recognition, there exist multiple, different techniques to classify instances of patterns, each approach being characterised by its own virtues and shortcomings. The idea of combining the output of multiple classifiers has been studied for several years (Ho (1992), Powalka et al. (1995), Selfridge (1958), and Xu et al. (1992)) but it is still difficult to choose a suitable combination algorithm. The choice of a good combination algorithm is even more pressing for multi-agent systems since it enables the use of all available knowledge and the extra computing time becomes less of a problem with the current developments in computer processing power.

Instead of defining the integration of opinions for agents as a meta-decision problem, we will focus on less cumbersome techniques. This avoids the undesirable consequences of meta-decision (Vuurpijl and Schomaker, 1998): (1) an extra, large amount of training data is needed and (2) for every agent that is added, the complete “meta-agent” needs to be trained again.

The most straightforward form of opinion integration is to let the agent cast a vote by forwarding the outcome they prefer best. The outcome with the most votes wins. This is called plurality voting and while it is simple and quite effective, it lacks depth. With depth we mean that agents often have a ranking of outcomes to indicate which are more likely candidates than others. Plurality voting only uses the absolute top of those rankings. In this section, we will thoroughly discuss a specific method for combining the rankings of different agents: the Borda count.

The Borda count is an easy, intuitively appealing, and powerful method of combining different rankings. Moreover, it has some variants that may perform better on specific decision making problems. However, the theoretical foundation of the approach is less well developed then in the case of plurality voting.

Standard Borda count and two variants

The Borda count is originally a voting method in which each voter gives a complete ranking of all possible alternatives (Ho (1992) and Borda (1781)). The highest ranked alternative (in for example an *n*-way vote) gets *n* votes and each subsequent alternative gets one vote less (so the number two gets *n-1* votes and the number three *n-2* and so on). Then, for each alternative, all the votes are added up and the alternative with the highest number of votes wins the election.

Ties in the accumulated votes are not resolved in the original Borda count. This method was introduced in 1770 by Jean-Charles de Borda (Black, 1968).

Each agent is a voter and the classes are the candidates. The method has depth as it uses the entire ranking information to come to a decision, not just the best guess of each agent. It also returns a complete ranking of the possible classes instead of its best guess, offering more flexibility for further uses. For example, consider decisions with a large number of possible outcomes, where the top-ranked candidate may be wrong. If the application context allows it (i.e., a collection of outcomes can be the answer instead of just one outcome), one could choose to accept a group of the best possible outcomes instead of just the top guess, increasing the probability of including the correct outcome. The ranked result of the Borda count gives suggestions concerning the alternatives just below the top rank.

What the Borda count lacks, is a way to differentiate between several outcomes based on their general performance or expertise. In fact, the assumption is that the top-ranked candidates of all outcomes are of comparable quality, thus all outcomes (voters) are treated equal, while this may not be desirable. A solution for this problem is given in (Ho, 1992). Another way of calculating the Borda count is averaging the rank given by each voter to an outcome, instead of adding up the votes. The new ranking is then calculated by ranking the averaged votes, highest one on top. Note that effectively, this does not change the results of the combination process, however, the concept of an average rank has interesting implications: assuming a probability distribution of rank numbers for a given outcome, there exist other measures than the mean to describe central values of that distribution. An example is the median: the rank value that splits the number of given rank numbers in half. The Borda count using the median instead of the mean, will be less susceptible to extreme voting behavior of a few agents with respect to some outcomes.

The second Borda variant is Nanson's Borda-elimination procedure (Black (1968) and Nanson (1882)). This is a multi-step procedure that repeatedly performs a Borda count and with each iteration deletes the lowest Borda ranked alternative from each agent's ranking. This allows the top-ranked outcomes to recover from extreme low votes.

5.5.3 Drawbacks of voting

The popularity of voting methods is easy to explain. They provide an increased performance for a minimal effort. They are simple, straightforward, and easily understood. This makes voting methods ideal in research on combining classifiers that does not focus on the combination of outputs, but that needs it all the same. As such, voting methods are much used in early fusion and ensemble creation research. Yet, when attention is shifted to the late fusion, it rapidly becomes obvious that voting methods have severe drawbacks.

As the pattern recognition problems grow increasingly complex, and with it the classifier combination system that has to solve it, the available information becomes more voluminous and heterogeneous. This is where the boon of the voting methods turns into their bane. Due to the inherent simplicity of the methods, they cannot properly deal with this multitudinous information. Either the voting methods will not function properly (1) or they will harshly reduce the information density (2).

Case (1) occurs more frequently in the more complex voting methods, such as the Borda Count and the Product rule. These methods have very strict requirements on their input. For example, most voting methods require all classifiers to produce results, in order to give a robust performance over subsequent classifications. The more sophisticated methods even require complete results over all classes. If these requirements are not met, the algorithm will not function or it will behave in an undefined manner. The Product rule, for example, multiplies all confidence values a class received from each classifier. If a class did not receive a confidence value from a classifier, it is unclear and implementation dependent what the result will be. Regarding it as a zero confidence would ruin the total score of the class, while ignoring it would amount to substituting it with a confidence of one (i.e. perfect) which is also unfair. It is possible to adapt or implement the voting methods so that irregularities in the input can be handled without failure or undefined results (mostly by explicitly defining the results in such cases). However, this will lead to a different behavior of the algorithm in different cases. This is a disadvantage, as it is a consequence of the adaptation and not a functionally planned imbalance.

Another situation in which voting methods do not function robustly can be seen in the second part of Section 5.5.2. Here, the Borda count and some variants are tested on two sets of data with a different type of noise in it. The tests show that the Borda count has a very different performance on both types of noise. In other words, if the classifiers on which the voting methods base their conclusions are subject to a larger variance in the type of errors they produce (not an unlikely situation, especially in harder problem areas), the voting methods will start to behave more eccentrically. They possess no mechanisms to detect or to cope with such variance.

Case (2) applies to the inability of voting methods to deal with extra information. This time, it are the simpler methods that are more problematic, instead of the complex voting methods as in case (1), yet all methods suffer this drawback. Voting methods work with the outcome of the classifiers and accept only that each classifier assigns a single similar feature per class. Simple arithmetics are then applied to these features. Furthermore, no prior information on the process whatsoever is used. All classifiers and possible classes are treated equally and all features are used exactly according to their numerical representation. For example, the difference between the first and fourth rank in the Borda Count is identical to the difference between the 11th and 14th rank, even though in most classifier outcomes, the former difference would be much more telling than the latter.

In previous sections, it became clear that the more complex voting methods that use more information have a similar or better performance than the methods that use less information. While this does not constitute a proof, it does support the notion that more information offers more possibilities to find the correct partition between the classes. If this is true on the relatively simple problems of the previous sections, one can only imagine how important the extra information will be with the truly complex applications in pattern recognition.

Voting methods are not designed to cope with extra information. While it is possible to adapt some methods to some of the problems (for example by adding a type of weight measure to the votes of classifiers), this only involves a small bit of information in each adaptation. The more information is put into the system, the more adaptations are needed and the more intricate the solution will be. It is highly doubtful that creating such a system by increased adaptations will outperform a system that is designed from the start to handle the information. Furthermore, it is highly questionable whether all problems can be solved, either in increasingly large combinations or even individually.

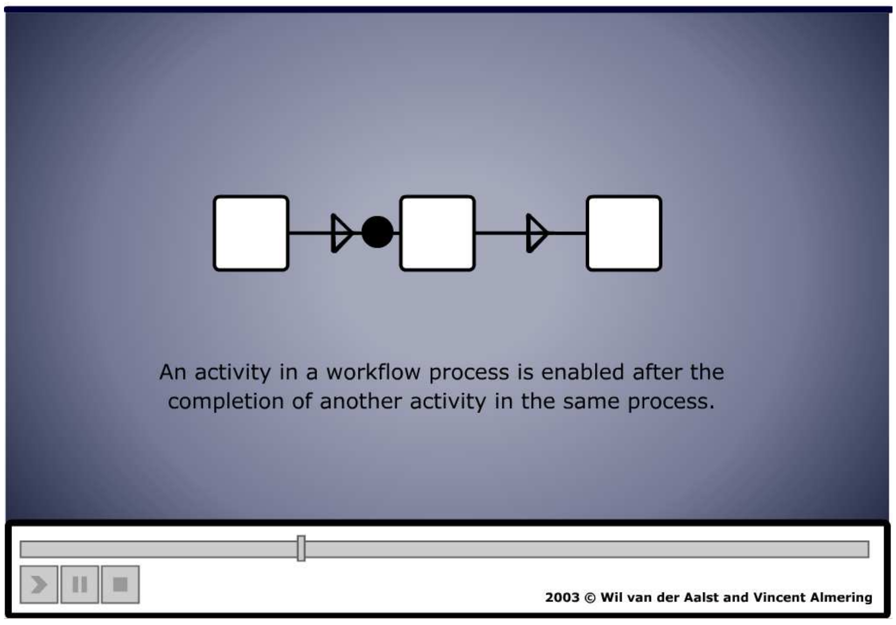
Voting methods are excellent when the classifier combination needs to be quick and simple. They should be used as such and are hard to beat in that area. However, using them when high quality results are expected or when a lot of information is available, is a misapplication of these elegant little algorithms.

6 Test Examples

Test examples have been identified to test the different coordination approaches. The examples were inspired by the workflow patterns defined by van der Aalst *et al.*, 2002, also shown at www.workflowpatterns.com. In total, seven test examples are described, as shown below. A test example consists of a number of components (agents), called {C1, C2, C3, ...}, and several types of data, called {d1, d2, d3, ..}. Different components need different data as input, and create different data as output.

Example 1 - Sequence

A screenshot of a computer animation of pattern 1 (taken from www.workflowpatterns.com) is depicted in Figure 5. The pattern is straightforward: after completion of the first component, the second component is activated, and after completion of the second, the third component is activated.



An activity in a workflow process is enabled after the completion of another activity in the same process.

Fig. 5. Workflow pattern 1 - Sequence

On the basis of this pattern, a next step was to create a corresponding test example. In principle, this means defining an example (in terms of components and data) in such a way that, if provided as input to a coordination approach, pattern 1 will come out. A visualisation of such an example (based on Pattern 1 above) is given in Figure 6. As can be seen in the figure, in this case component C1 needs data d1 as input, and creates data d2 as output. Moreover, as indicated in the box on the right, the *input data* (the data that is initially available to the system) is d1, and the *goal data* (the data that the system needs to create in order to be successful) is d4. Given this situation, the expectation is that if any coordination mechanism is applied to the example, the result will be a trace in which the components are activated in sequence (i.e., first C1, then C2, and then C3). Note that in this case it is assumed that data is shared, i.e., whenever a component generates output data, this data is immediately available to all other components in the system. This could be implemented, for example, by incorporating a “shared repository”, where all components store their output data and read their input data from. As opposed to this assumption, another alternative would be to allow local access to data, for example by incorporating explicit information links that specify which data is transferred from components to each other. Finally, note that another assumption is that data cannot be removed. Thus, once data is written to the shared repository, it will stay there.

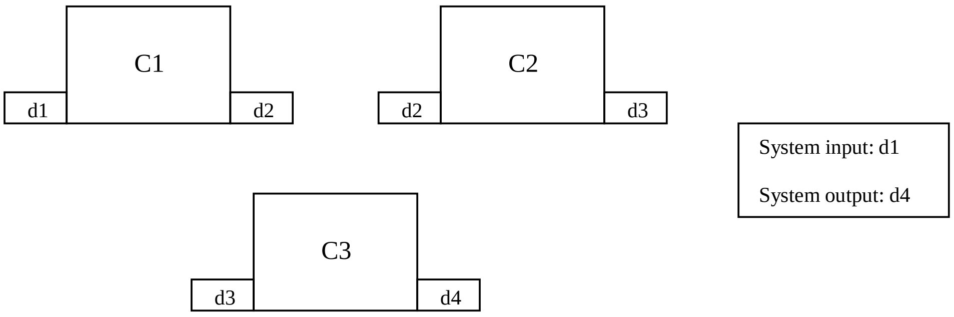


Fig. 6. Test example 1 - Sequence

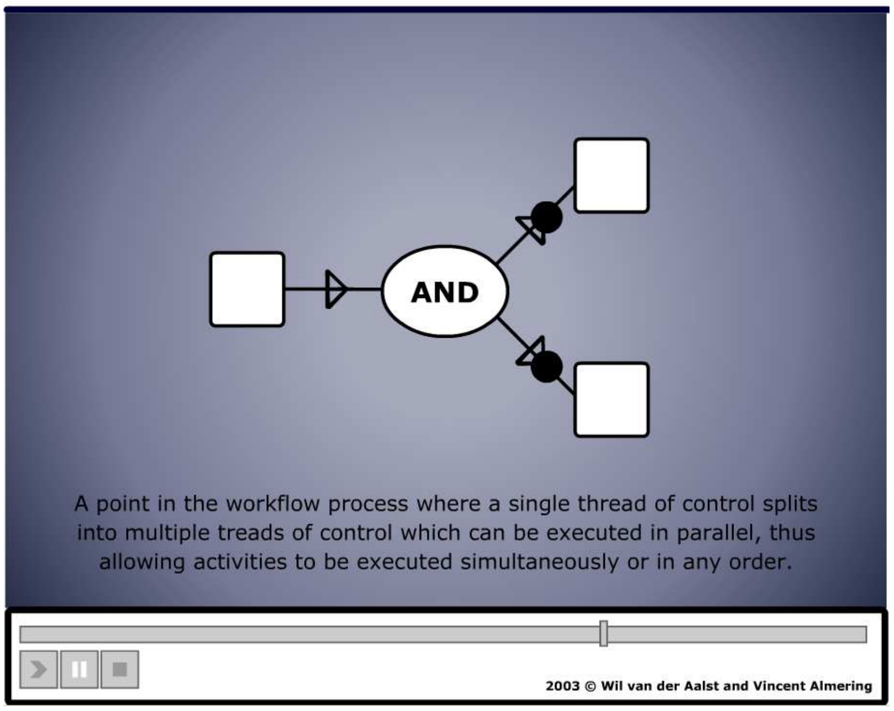
In LEADSTO, this example is formalised as follows:

```
component_input_number((c|1), 1)
component_input ((c|1), (d|1))
component_output_number((c|1), 1)
component_output ((c|1), (d|2))

component_input_number((c|2), 1)
component_input ((c|2), (d|2))
component_output_number((c|2), 1)
component_output ((c|2), (d|3))

component_input_number((c|3), 1)
component_input ((c|3), (d|3))
component_output_number((c|3), 1)
component_output ((c|3), (d|4))

initial_data(d|1)
goal_data(d|4)
```

Here, the first statement indicates that component C1 needs one type of input data. The second statement indicates that this component needs d1 as input data.

Example 2 - Parallel Split

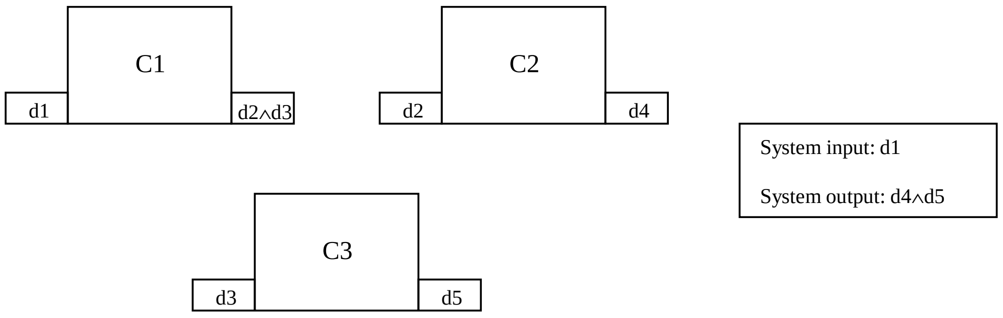
A screenshot of pattern 2, the parallel split, is depicted in Figure 7. Here, the two components on the right can be executed either simultaneously or in any order.



A point in the workflow process where a single thread of control splits into multiple treads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.

Fig. 7. Workflow pattern 2 - Parallel Split

The test example that was created on the basis of this pattern is shown in Figure 8.



System input: d1

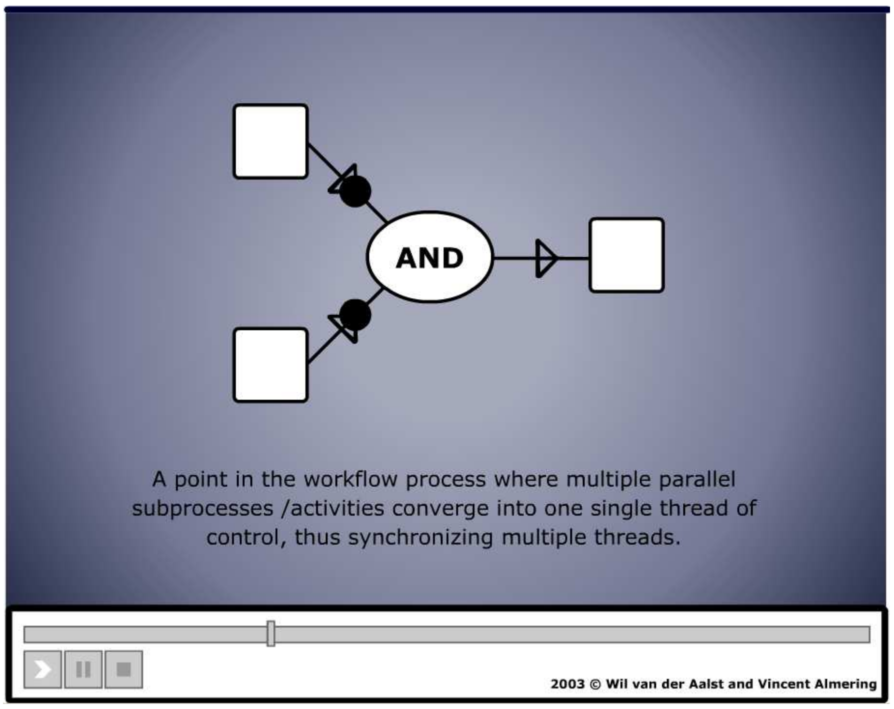
System output: d4∧d5

Fig. 8. Test example 2 - Parallel Split

Note that in this case the ∧ stands for the conjunction of two data types. For example, the output data of component C1 is d2 and d3. Likewise, the goal data is d4 and d5.

Example 3 - Synchronization

A screenshot of pattern 3, the synchronization, is depicted in Figure 9. Here, the two components on the left can be executed either simultaneously or in any order.



A point in the workflow process where multiple parallel subprocesses /activities converge into one single thread of control, thus synchronizing multiple threads.

Fig. 9. Workflow pattern 3 - Synchronization

The test example that was created on the basis of this pattern is shown in Figure 10.

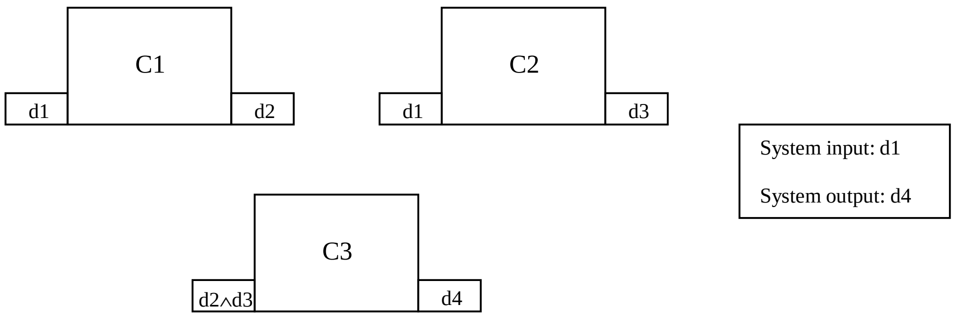


Fig. 10. Test example 3 - Synchronization

Note that in this case it is assumed that a component cannot reason with “partial” data (this would be the case when, e.g., component C3 starts reasoning with d2 only, whilst its input data is d2 and d3).

Example 4 - Exclusive Choice

A screenshot of pattern 4, the exclusive choice, is depicted in Figure 11. Here, either component B or component C may be activated, but not both.

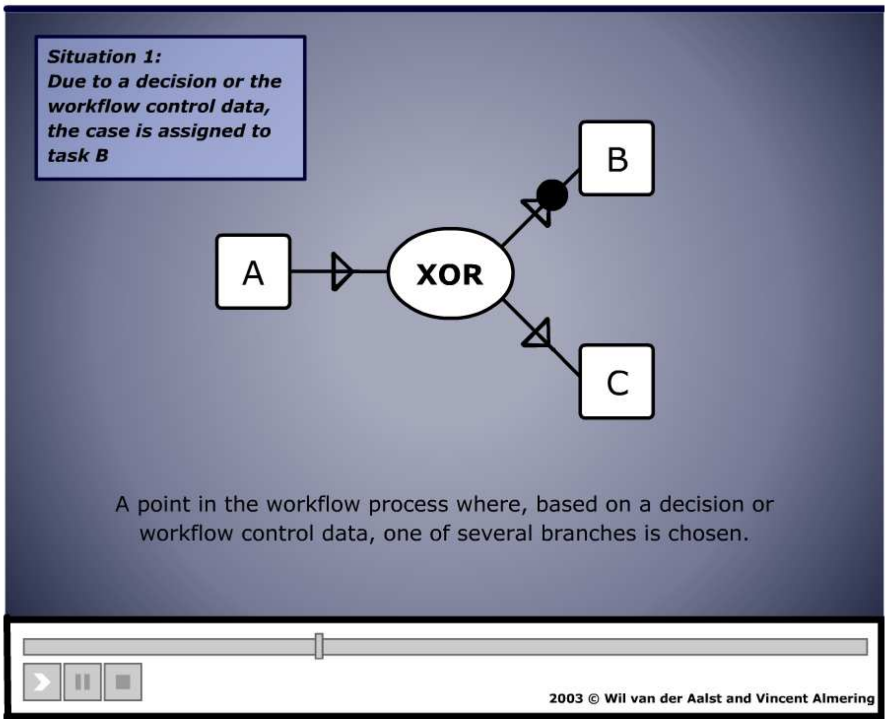

Situation 1:
Due to a decision or the workflow control data, the case is assigned to task B
B
A
XOR
C
A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.
2003 © Wil van der Aalst and Vincent Almering

Fig. 11. Workflow pattern 4 - Exclusive Choice

The test example that was created on the basis of this pattern is shown in Figure 12.

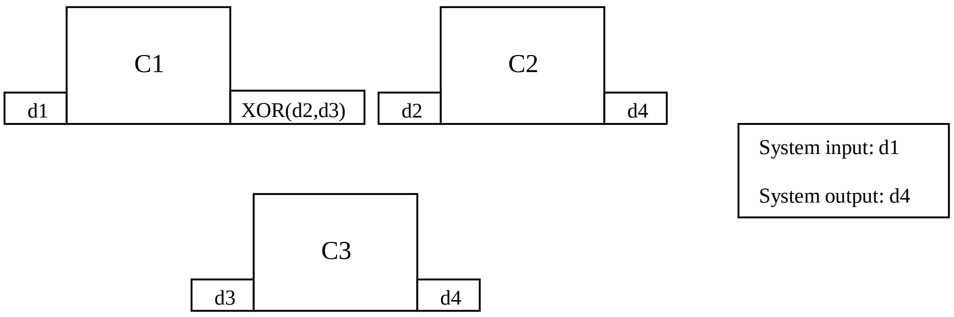
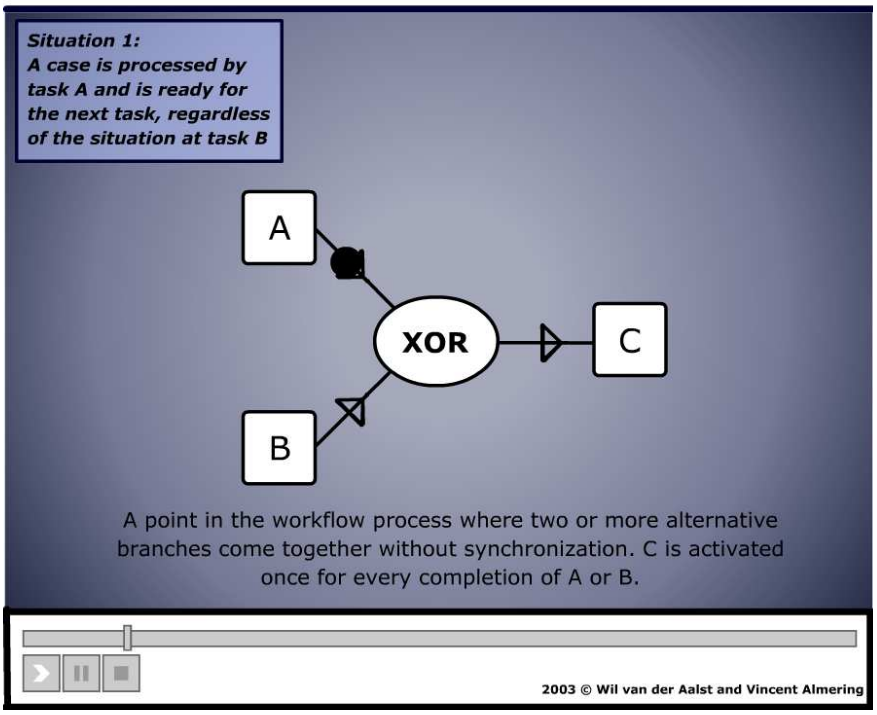


Fig. 12. Test example 4 - Exclusive Choice

Note that in this case the XOR stands for the exclusive disjunction of two data types. For example, the output data of component C1 is either d2 or d3, but not both. The specific output generated by the component may differ in different simulation runs.

Example 5 - Simple Merge

A screenshot of pattern 5, the simple merge, is depicted in Figure 13. Here, either component A or component B may be activated, but not both.

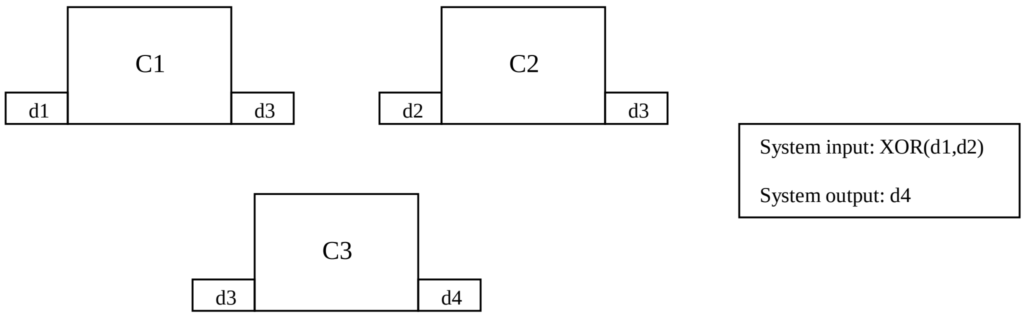

Situation 1:
A case is processed by task A and is ready for the next task, regardless of the situation at task B

A point in the workflow process where two or more alternative branches come together without synchronization. C is activated once for every completion of A or B.

2003 © Wil van der Aalst and Vincent Almering

Fig. 13. Workflow pattern 5 - Simple Merge

The test example that was created on the basis of this pattern is shown in Figure 14.



System input: XOR(d1,d2)

System output: d4

Fig. 14. Test example 5 - Simple Merge

Note that in this case the input data is the exclusive disjunction of d1 and d2, i.e., in some simulation runs it is d1, and in others it is d2.

Example 6 - Multi Choice

A screenshot of pattern 6, the multi choice, is depicted in Figure 15. Here, either component B, or component C, or both components may be activated.

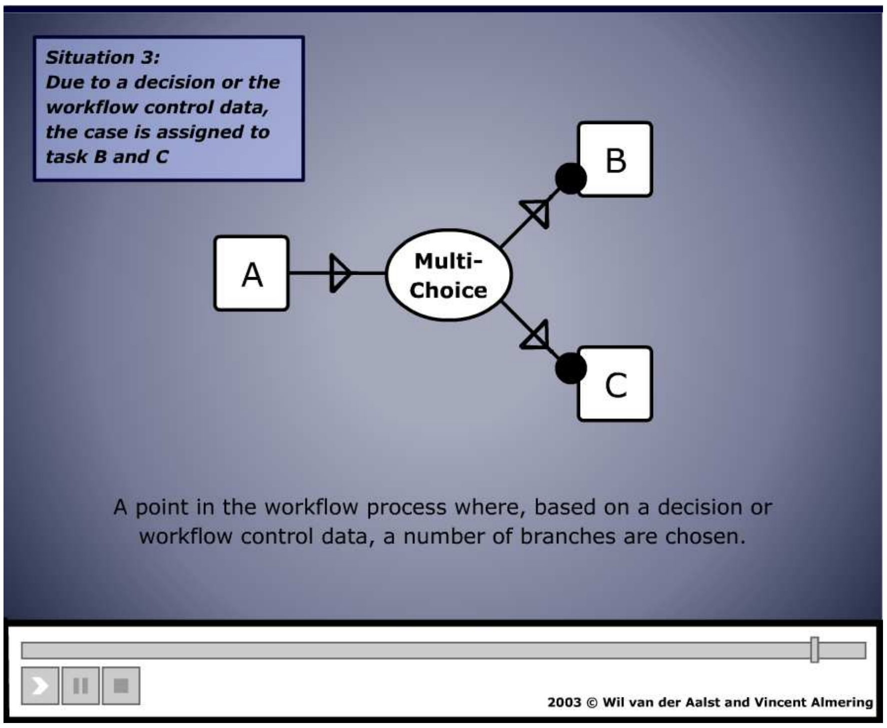

Situation 3:
Due to a decision or the workflow control data, the case is assigned to task B and C
A
Multi-Choice
B
C
A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen.
2003 © Wil van der Aalst and Vincent Almering

Fig. 15. Workflow pattern 6 - Multi Choice

The test example that was created on the basis of this pattern is shown in Figure 16.

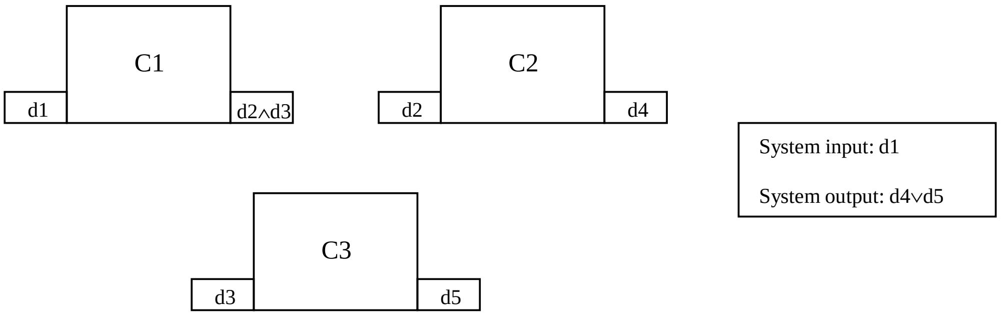
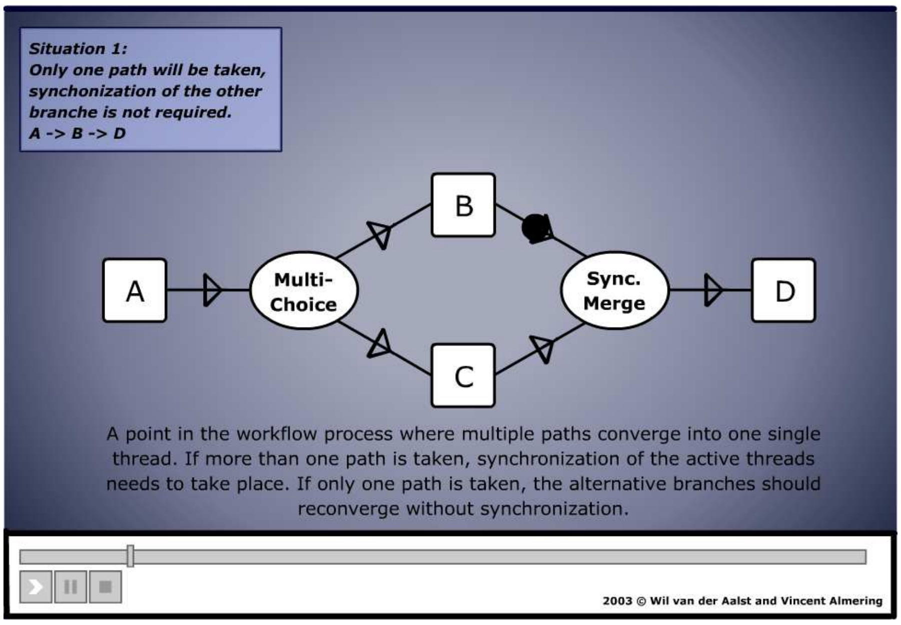


Fig. 16. Test example 6 - Multi Choice

Note that in this case the ∨ stands for the standard disjunction of two data types. Thus, in this case the goal data of the system is d4 or d5 or both.

Example 7 - Synchronizing Merge

A screenshot of pattern 7, the synchronizing merge, is depicted in Figure 17. The beginning of this pattern is similar to pattern 6, but after the “sync. merge” entity has been reached, only component D has to be activated.


Situation 1:
Only one path will be taken, synchronization of the other branche is not required.
A -> B -> D

A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization.

2003 © Wil van der Aalst and Vincent Almering

Fig. 17. Workflow pattern 7 - Synchronizing Merge

The test example that was created on the basis of this pattern is shown in Figure 18.

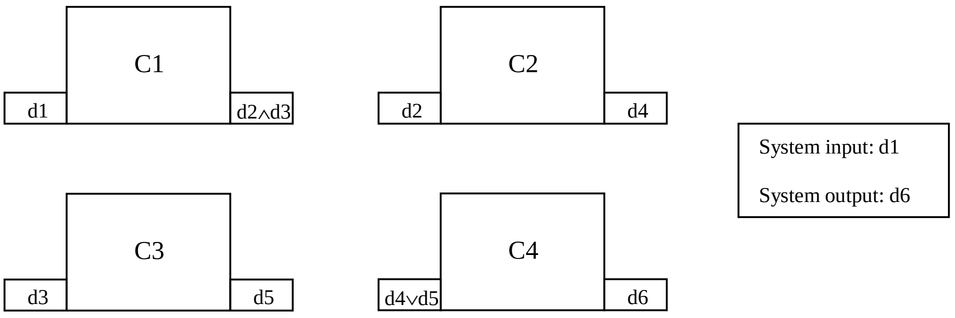


Fig. 18. Test example 7 - Synchronizing Merge

As can be seen in Figure 18, in this last example both a conjunction in a component’s output data and a disjunction in a component’s input data occur. Furthermore, note that, when formalising this example in LEADSTO, the disjunction on the input side of C4 is modelled by defining three separate variants of C4: one with d4 as input, one with d5 as input, and one with d4 and d5 as input.

To compare the coordination approaches described in Section 5 against these patterns, a number of simulation experiments have been performed. In these experiments, the focus was on three approaches in particular: Maes' *Behavior networks*, Selfridge's *Pandemonium*, and the *Voting Mechanism*. These approaches were chosen for two reasons. First, because they are among the most popular and well-known approaches in the literature on coordination. Second, because together they more or less cover the area of different coordination approaches: the Behavior networks use a rather 'global' strategy (i.e., the different agents have information about all other agents in the system), whereas the Voting Mechanism and (especially) the Pandemonium use a 'local' strategy (i.e., the agents involved only have information about themselves or their direct neighbours).

The three selected coordination approaches have been implemented in the LEADSTO simulation language. Finally, the implemented simulation models have been applied to the test examples. The simulation models for the Behavior networks, the Pandemonium, and the Voting Mechanism, are addressed, respectively, in Section 7, 8, and 9.

7 Behavior Networks Simulation

7.1 Simulation Model

The simulation model for Maes' behavior networks is created based on the mathematical calculations as presented in Section 5.1. One difference is present which is the lowering of the threshold which is not performed within the simulation model, the highest executable component is simply selected. It can however easily be incorporated in the simulation model. The LEADSTO specification for the algorithm can be found in Appendix C1 and roughly corresponds to the order in which the formulas are presented in 5.1.2. The setting that have been used for ϕ, γ, and δ are 0.1, 0.3 and 0.5 respectively. The ontology used within the simulation model is shown in Table 1. Note that this does not specify the complete ontology within the LEADSTO specification, but does explain all the terms that will be shown in the traces in the next section.

Table 1. Relations used within the behavior networks simulation model

Relation	Description
input_from_state: TIME x COMPONENT x VALUE	At the time point the component gets the value for activation through the state at that time point.
input_from_goals: TIME x COMPONENT x VALUE	At the time point the component gets the value for activation through the goals that have been set.
spreads_fw: COMPONENT x COMPONENT x TIME x VALUE	At the specified time point the specified activation spreads forwards from the first component to the second
spreads_bw: COMPONENT x COMPONENT x TIME x VALUE	At the specified time point the specified activation spreads backwards from the first component to the second
executable: TIME x COMPONENT	This specifies that the component is executable at the particular time point.
decay: TIME x COMPONENT x VALUE	The component has the specified decay value at the particular time point.
alpha: TIME x COMPONENT x VALUE	The component has the specified alpha value at the particular time point.
active: TIME x COMPONENT x VALUE	This relationship specifies whether or not a component was active at a particular time point. In case VALUE is 1 this is the case, in case of a 0 this is not the case.
activated: COMPONENT	The component is activated.

7.2 Simulation Traces

This section presents the simulation traces that have resulted from executing the algorithm on the examples as presented in Section 6. The calculations in the first trace will be explained in detail whereas for the remainder of the traces only the overall result of the calculations will be shown.

Trace Example 1 – Sequence

Figure 19 shows the simulation trace of the behavior network algorithm for the first example. The left side of the figure shows the states during the simulation whereas the right side shows a time line where a dark indicates the state being true and a light box the state being false.

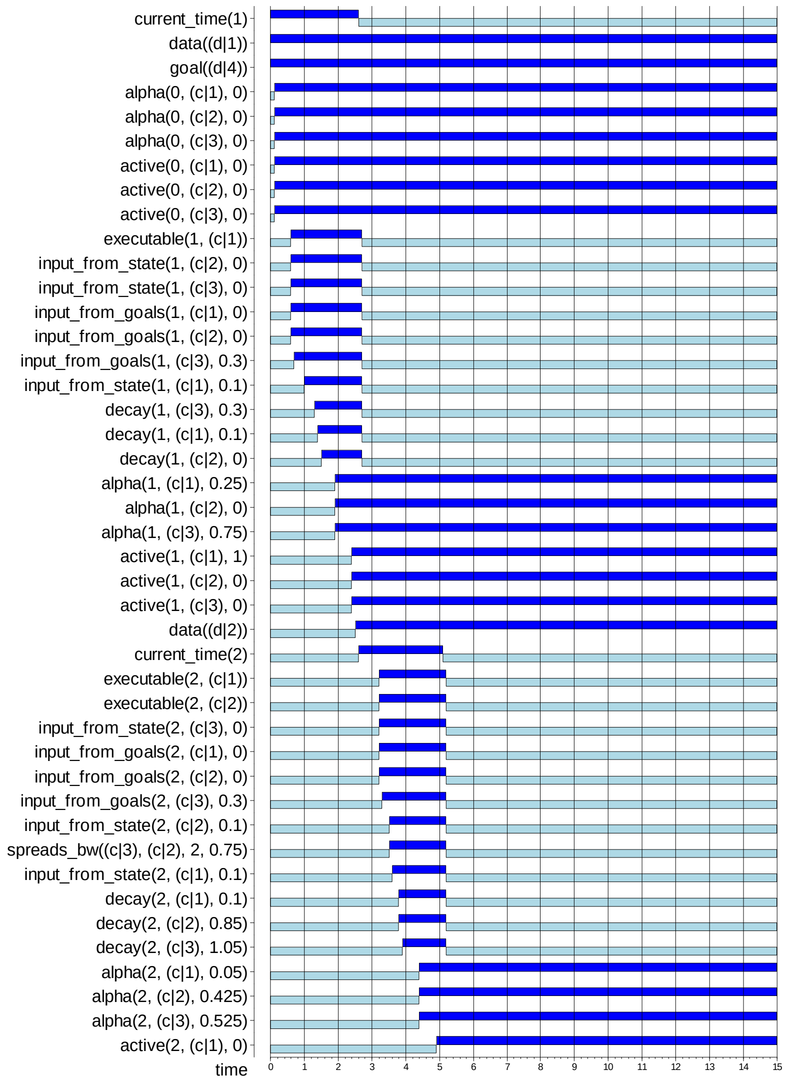


Fig. 19. Trace resulting from running the behavior networks algorithm on example 1 (continued on next page)

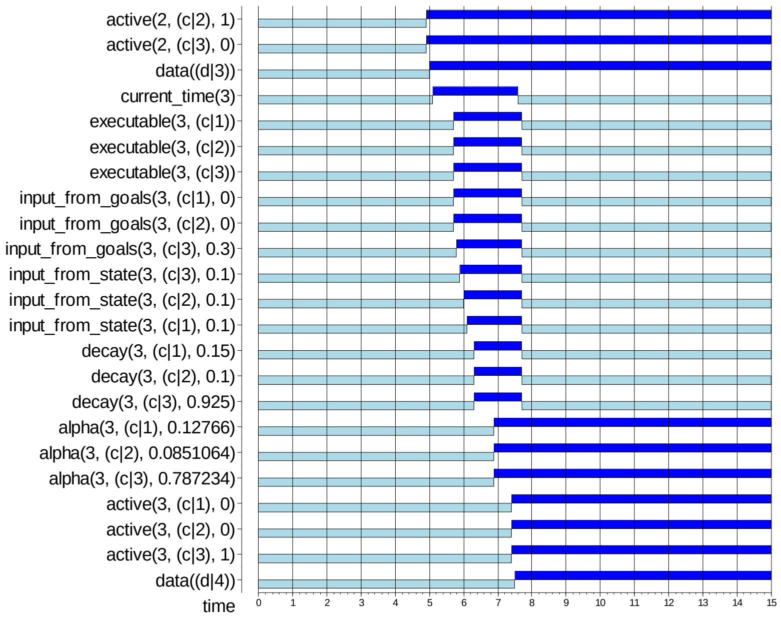


Fig. 19 (contd). Trace resulting from running the behavior networks algorithm on example 1

Initially, the data present is set to d1:

    data(d|1)

Furthermore, the goal is set to d4 for this particular scenario:

    goal(d|4)

Before executing the algorithm several initial values are set to enable a proper functioning of the algorithm. First of all, the alpha value of the component currently present in the system are set to 0 for the time point before the current time point (i.e. time point 0):

    alpha(0, c|1, 0)
    alpha(0, c|2, 0)
    alpha(0, c|3, 0)

Furthermore, the components' activity at time point 0 is set to 0 as well:

    active(0, c|1, 0)
    active(0, c|2, 0)
    active(0, c|3, 0)

Now the algorithm is executed. First of all, it is determined that only component C1 is executable give the current data available:

    executable(1, c|1)

Calculations are performed to determine the activity within the different component. To enable this calculations several intermediate steps are taken. First of all, the input from the current state is calculated (i.e. given the current data available what is the activation caused for the different components). Since component C1 is the only component that has its preconditions fulfilled, it is the only component to have activation from this source:

    input_from_state(1, c|1, 0.1)

Another intermediate step is the input from the goals. Since only C3 has a goal as an output, this component it the only one to receive activation through this source:

    input_from_goals(1, c|3, 0.3)

Due to the fact that the previous alpha value is 0, no activation is spread around the network, so the decay can be calculated for the three components present in the system by simply summing up the input from the goals and state per component:

    decay(1, c|1, 0.1)
    decay(1, c|2, 0)
    decay(1, c|3, 0.3)

Calculating the alpha value entails normalizing these numbers. The maximum activation is set to 1 in this example, resulting in the following alpha values:

    alpha(1, c|1, 0.25)

alpha(1, c|2, 0)
alpha(1, c|3, 0.75)

As a result, component C1 is activated as this is the executable component with the highest alpha value:

active(1, c|1, 1)
active(1, c|2, 0)
active(1, c|3, 0)

Due to the activity of component C1 its output data is generated, which shows in the trace by means of the presence of data d2:

data(d|2)

A new round of the algorithm is performed, both components C1 and C2 are now derived to be executable as the data is assumed to remain present permanently. The input from the goals remains the same as these have not changed. The input from state however changes due to the additional data d2 being present, resulting in an input from state for component C2 as well:

input_from_state(2, c|2, 0.1)

Since C3 was not active at the previous time point, its activation spreads back through the network, resulting in a backwards spread from C3 to C2:

spreads_bw(c|3, c|2, 2, 0.75)

Calculation of the decay can now be performed:

decay(2, c|1, 0.1)
decay(2, c|2, 0.85)
decay(2, c|3, 1.05)

Normalisation takes place and eventually C2 is selected, resulting in data d3 being present. In the last cycle, C3 is selected with by far the highest alpha value, resulting in the overall goal being reached:

data(d|4)

Trace Example 2 – Parallel Split

Similar to the trace described in the section above, Figure 20 shows the trace of the behavior networks algorithm in the parallel split case. There are however more possible outcomes for the trace which can all be generated by the algorithm.

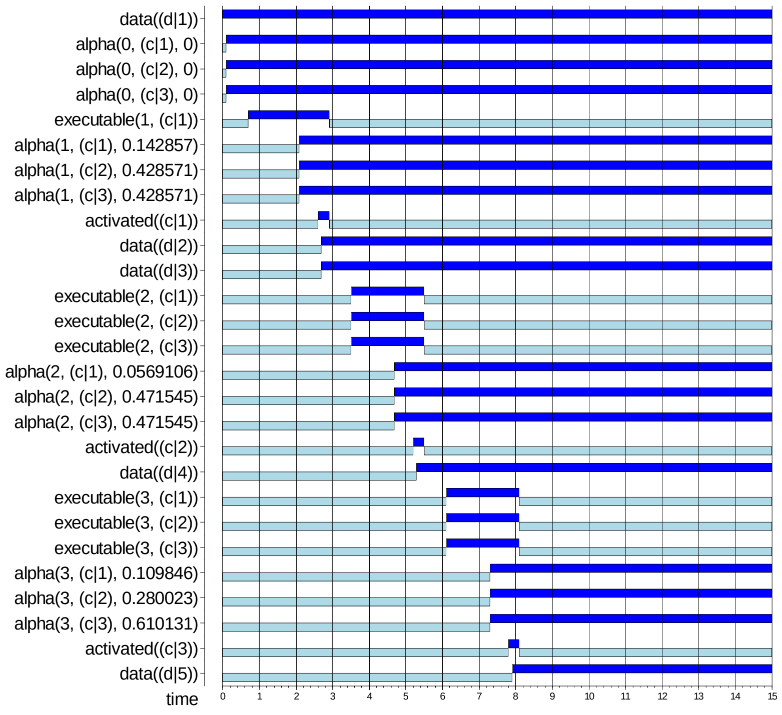


Fig. 20. Trace resulting from running the behavior networks algorithm on example 2

Figure 21 shows another possible outcome of the algorithm; this variation in outcome is established through a random selector in case the evaluation value of the different components is the same. Figure 21 shows that first of all, component C1 is activated as this is the only executable component at the start. Data d2 and d3 are both generated and now a conflict arises: Both C2 and C3 have the same evaluation value, which is logical: They both have their input data available and contribute evenly to the overall goal. A random choice is made, in the case of Figure 20 component C2 is selected first after which component C3 is selected. Figure 21 shows however that the different order can be generated as well (C2 after C3). At this point, the algorithm does no support parallel execution, but this can easily be incorporated if wanted.

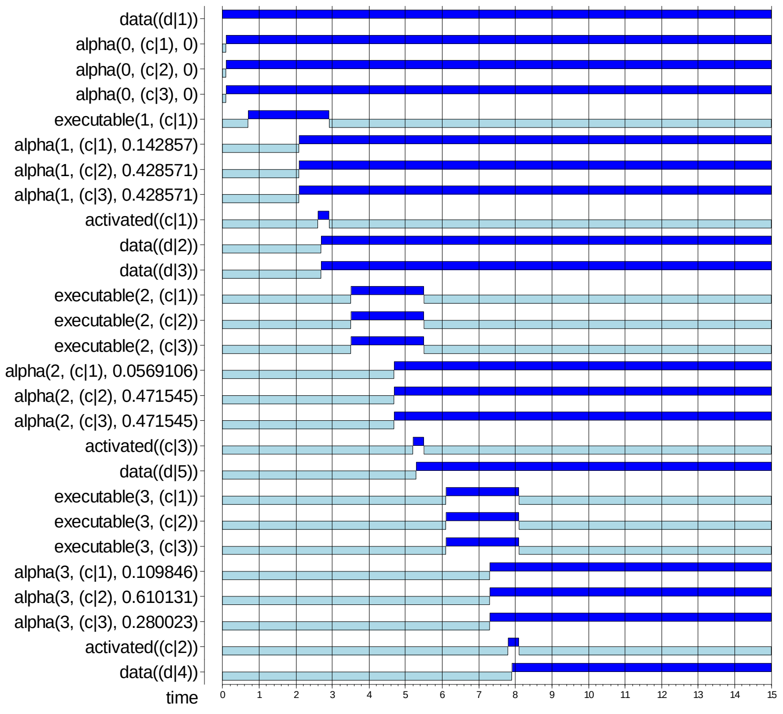


Fig. 21. Another trace resulting from running the behavior networks algorithm on example 2

Trace Example 3 – Synchronization

Figure 22 shows and example of a trace for patterns 3, in this particular trace the activation sequence is C1-C2-C3. The trace is specified in the same fashion as before, and will therefore not be explained in detail.

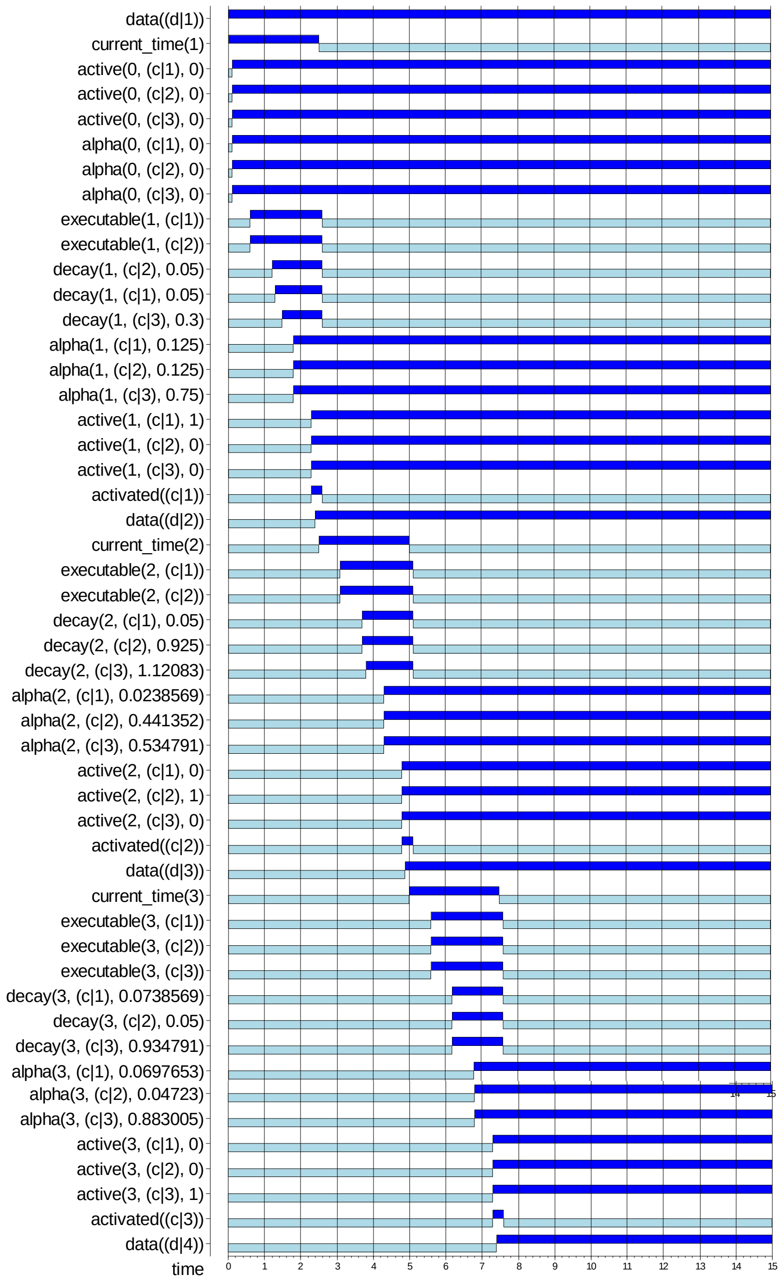


Fig. 22. Trace resulting from running the behavior networks algorithm on example 3

Trace Example 4 – Exclusive Choice

Figure 23 shows an example trace for the exclusive choice example. As can be seen, C1 is activated first, after which it generates output d3. Since only C3 can use this data, this component is selected and generates the goal data.

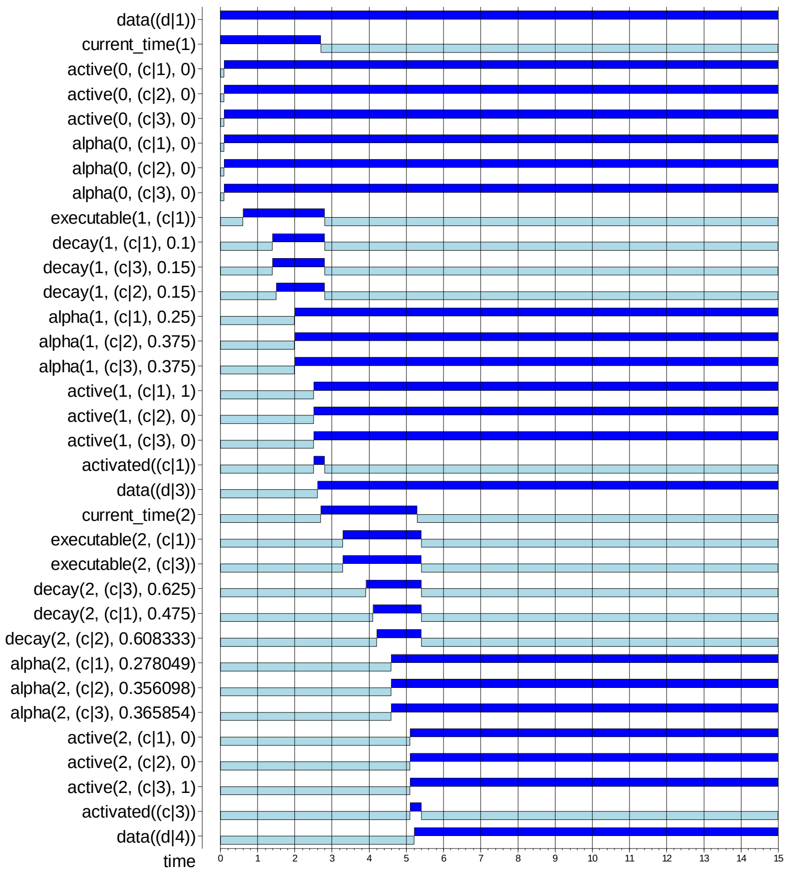


Fig. 23. Trace resulting from running the behavior networks algorithm on example 4

Trace Example 5 – Simple Merge

An example result trace for example 5 using behavior networks is shown in Figure 24. Initially, date d1 is available in this trace, resulting in C1 being activated after which C3 is activated and the goal data is formed.

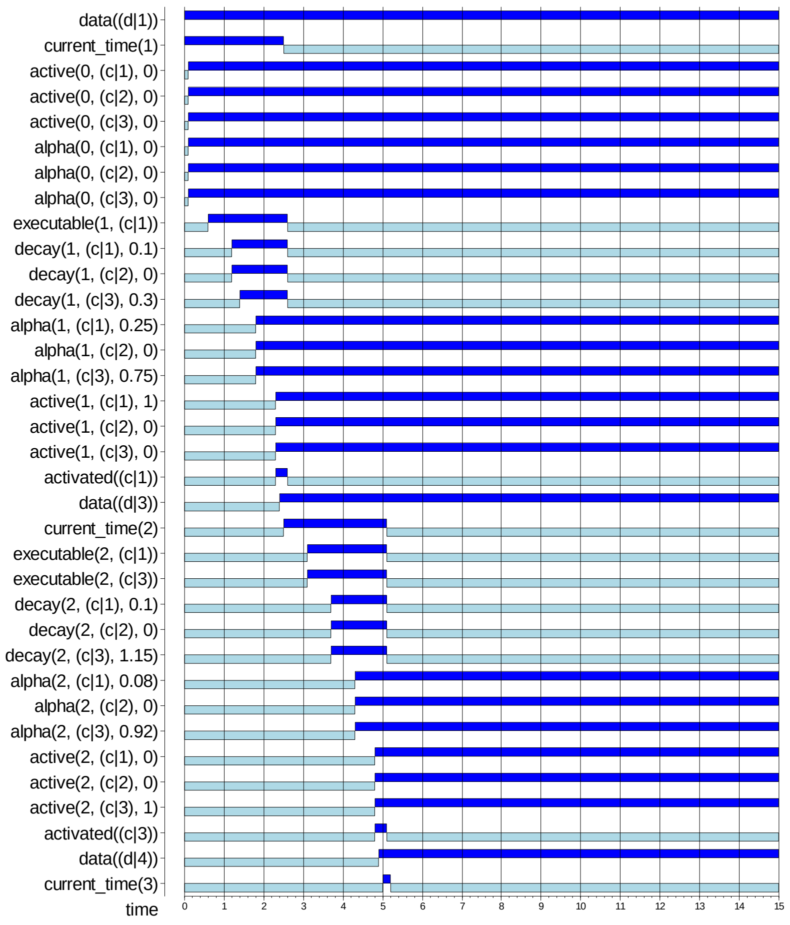


Fig. 24. Trace resulting from running the behavior networks algorithm on example 5

Trace Example 6 – Multi Choice

In the multi choice scenario there are three components. In the trace shown in Figure 25 the algorithm first activates C1 after which C2 is activated. Finally, C3 is activated as well, as this is allowed by the pattern. This is not an efficient trace as the shortest path would either be C1 – C2 or C1 – C3, some fine-tuning of the parameters can however improve these situations.

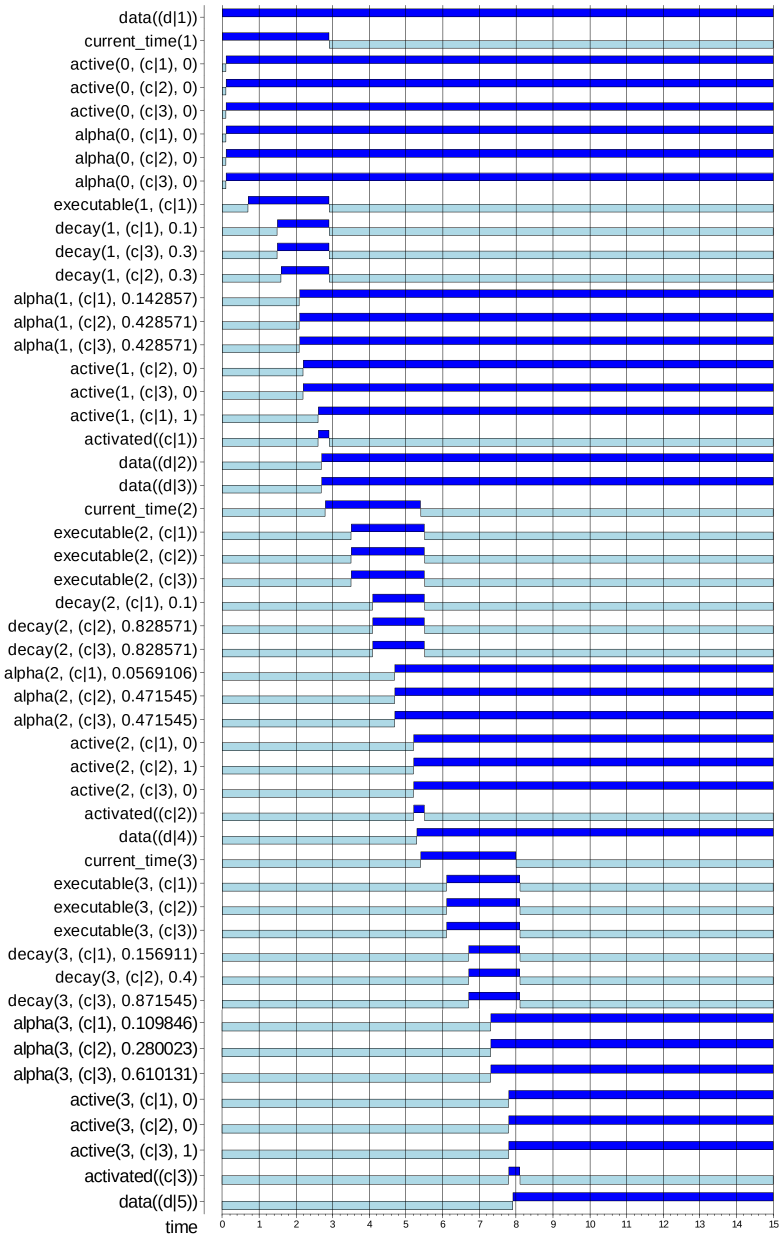


Fig. 25. Trace resulting from running the behavior networks algorithm on example 6

Trace Example 7 – Synchronizing Merge

An example trace for the synchronizing merge is shown in Figure 26. Note that the component C4 has been split up into three components: C4, C5 and C6: C4 takes as input d4 and generates d6; C5 uses d5 to generate d6, and finally, C6 takes d4 and d5 as input and generates d6. This has been done to minimise adaptation of the algorithm. The trace shows that the following sequence is taken: C1 – C2 – C3 – C4 whereas you would expect the algorithm to go for the optimal solution. Fine-tuning of the parameters could avoid these kind of inefficient traces.

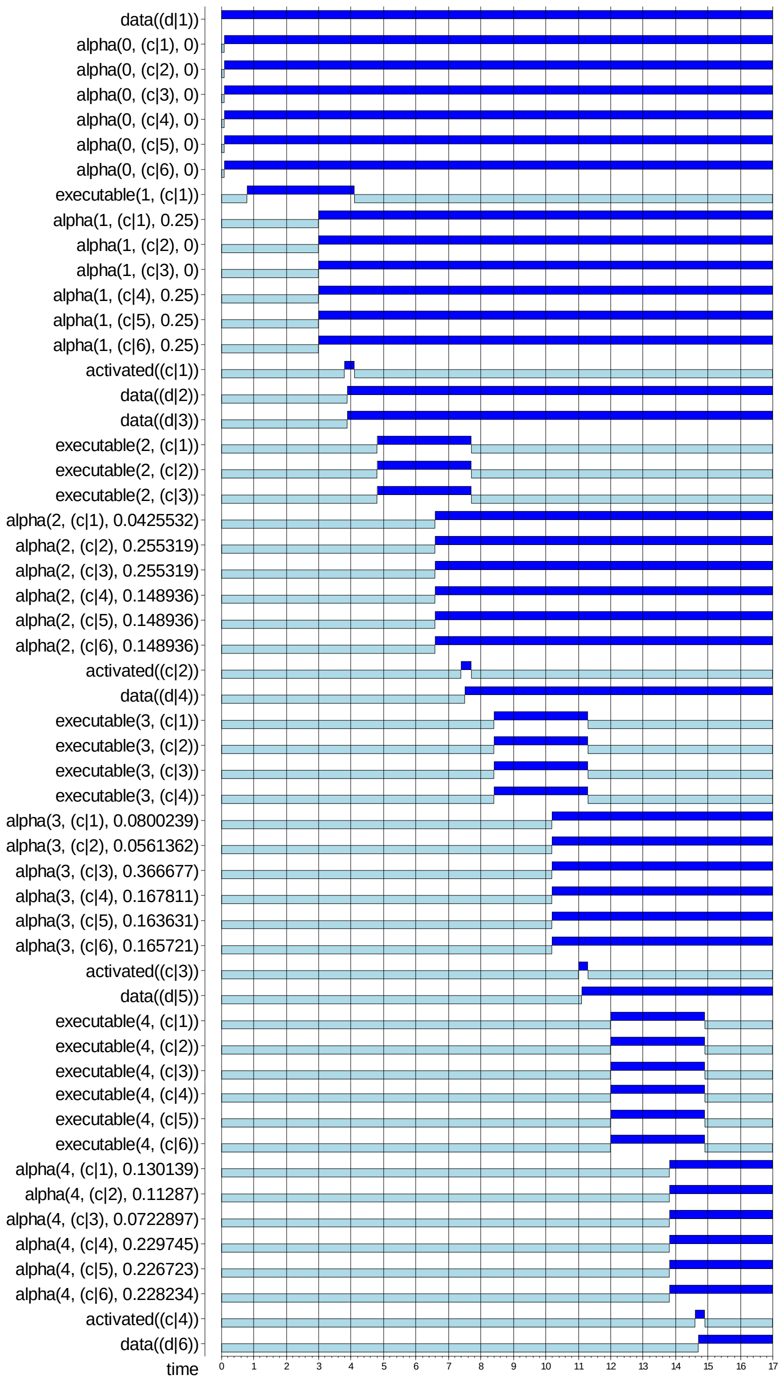


Fig. 26. Trace resulting from running the behavior networks algorithm on example 7

8 Pandemonium Simulation

8.1 Simulation Model

Generic Part

The algorithm used for the Pandemonium is similar to the description in Section 5, but modified with some simplifying assumptions. In particular, the following procedure is assumed: at the beginning of the process, only the initial data is placed at the "shared repository" (see Section 6). Whenever new data has been added to the repository, a new round starts in which all agents can *shout*. The idea is that, the more urgent an agent thinks it is for him to be activated, the louder it will shout. The agent that shouts loudest will be allowed to start reasoning. In case two agents shout with the exact same strength, then either the first agent, or the second agent, or both are activated (this decision is made randomly, with equal probabilities). When an agent is activated, this results in the agent adding its output data to the repository, and the start of a new round.

The LEADSTO specification for the algorithm can be found in Appendix C2. To model this algorithm in LEADSTO, the following ontology is used (again, only the elements that are shown in the traces are mentioned):

Table 2. Relations used within the pandemonium simulation model

Relation	Explanation
data: DATA	This specifies that a certain type of data is present in the repository.
shout: COMPONENT x VALUE	An agent shouts with a certain (real) value.
active_component: COMPONENT	An agent is activated.

Specific Part

To determine how loud they will shout, the agents make use of a *shout function*. For different variants of the Pandemonium model, different shout functions may be used. In the current model, each agent uses the following types of information in its shout function at time point t:

- the amount of data it needs as input (represented by i1)
- the amount of its input data that is available at t (represented by i2)
- the amount of data it produces as output (represented by o1)
- the amount of its output data that is already present at t (represented by o2)
- the maximum amount of input data any agent may need (represented by max_i)
- the maximum amount of output data any agent may produce (represented by max_o)

Given these elements, the shout value (i.e., the strength with which an agent shouts, represented by sv) is modelled as follows:

$$sv = (i2/i1)^{\beta 1} * (1 - o2/o1)^{\beta 2} * (i1/max\_i)^{\beta 3} * (o1/max\_o)^{\beta 4}$$

Here, $\beta 1$, $\beta 2$, $\beta 3$, and $\beta 4$ are real numbers between 1 and 1.5, indicating the importance of the corresponding factor. For example, $\beta 1=1.4$, $\beta 2=1.3$, $\beta 3=1.1$, and $\beta 4=1.2$. Thus, sv will be a value between 0 and 1.

To illustrate the idea, consider example 3 above (the synchronization). Suppose that at a certain point in time the only data that is present at the repository is d1 and d2. Then, the following information would be available to agent C3:

- i1 = 2 (because C3 needs d2 and d3 as input)
- i2 = 1 (because only d2 is available as input)
- o1 = 1 (because C3 produces d4 as output)
- o2 = 0 (because d4 is not present as output)

- max_i = 2 (because no component needs more than 2 different data types)
- max_o = 1 (because no component produces more than 1 different data types)

As a result, the shout value sv of C3 will be: $(1/2)^{\beta 1} * (1 - 0/1)^{\beta 2} * (2/2)^{\beta 3} * (1/1)^{\beta 4}$.

8.2 Simulation Traces

This section presents the simulation traces that have resulted from applying the pandemonium algorithm to the examples presented in Section 6.

Trace Example 1 – Sequence

Figure 27 shows the simulation trace of the pandemonium algorithm for example 1.

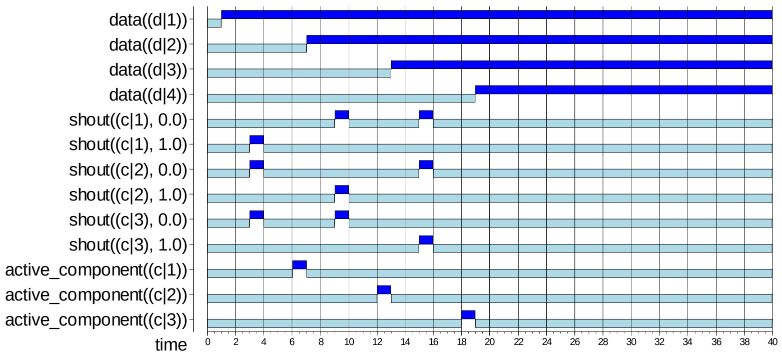


Fig. 27. Pandemonium trace 1.

As can be seen in Figure 27, initially the only data that is present is d1:
data((d|1))
Based on this data, every agent starts shouting. Agent C1 shouts loudest (with strength 1.0, whilst the others shout with strength 0.0):
shout((c|1), 1.0)
shout((c|2), 0.0)
shout((c|3), 0.0)
Thus, agent C1 is selected to become active:
active_component((c|1))
As a result, agent C1 creates data d2, which is stored at the repository as well:
data((d|2))
Again, every agent starts shouting. Agent C2 shouts loudest (with strength 1.0, whilst the others shout with strength 0.0):
shout((c|1), 0.0)
shout((c|2), 1.0)
shout((c|3), 0.0)
Next, agent C2 is selected to become active:
active_component((c|2))
Next, agent C2 creates data d3, which is stored at the repository as well:
data((d|3))
Again, every agent starts shouting. Agent C3 shouts loudest (with strength 1.0, whilst the others shout with strength 0.0):
shout((c|1), 0.0)
shout((c|2), 0.0)
shout((c|3), 1.0)
Next, agent C3 is selected to become active:
active_component((c|3))
Eventually, agent C3 creates data d4, which is stored at the repository as well:
data((d|4))
Since d4 is the goal data, at this point the process terminates.

Trace Example 2 – Parallel Split

As already shown earlier, test example 2 contains a non-deterministic element. Therefore, applying the pandemonium algorithm to this pattern may result in different traces. In total, three different traces may be generated. These traces are shown in Figure 28, 29, and 30.

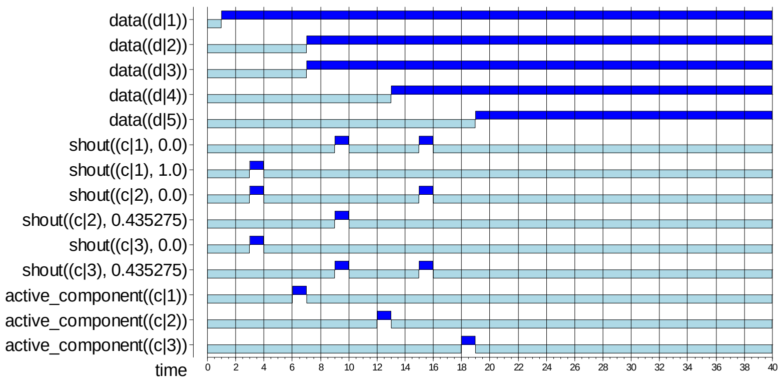


Fig. 28. Pandemonium trace 2a.

This trace shows a situation where the example components are activated in a sequence: first C1 is activated, then C2 (based on a random choice), and then C3.

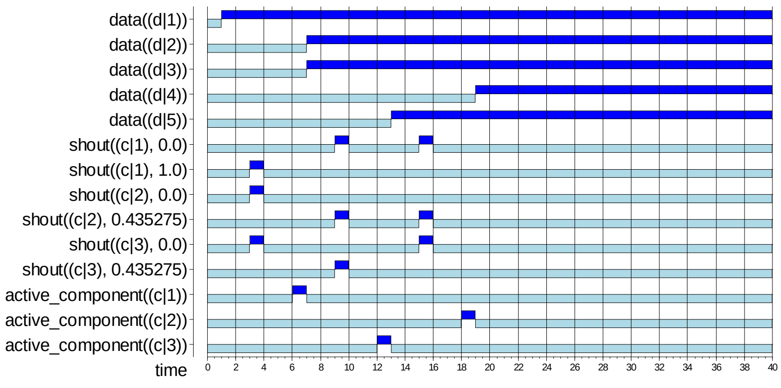


Fig. 29. Pandemonium trace 2b.

This trace shows another situation where the example components are activated in a sequence: here, first C1 is activated, then C3 (based on a random choice), and then C2.

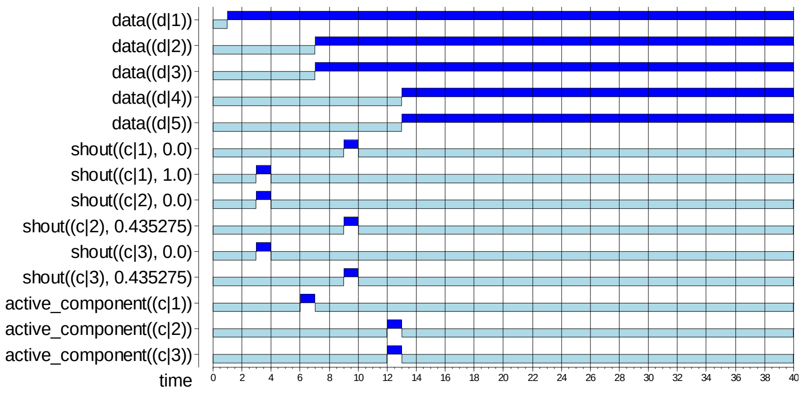


Fig. 30. Pandemonium trace 2c.

This trace shows the situation where two of the example components are activated in parallel: here, first C1 is activated, but then both C2 and C3 are activated simultaneously (again, based on a random choice).

Like test example 2, most examples contain a non-deterministic element, thus may result in multiple different traces. To limit complexity, for the remaining patterns only one trace is shown.

Trace Example 3 – Synchronization

Figure 31 shows a simulation trace of the pandemonium algorithm for example 3. Here, first both C1 and C2 are activated, followed by C3.

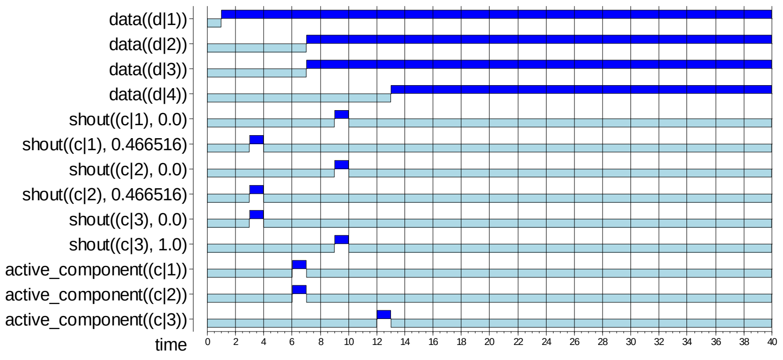


Fig. 31. Pandemonium trace 3.

Trace Example 4 – Exclusive Choice

Figure 32 shows a simulation trace of the pandemonium algorithm for example 4. Here, first C1 is activated, followed by C2.

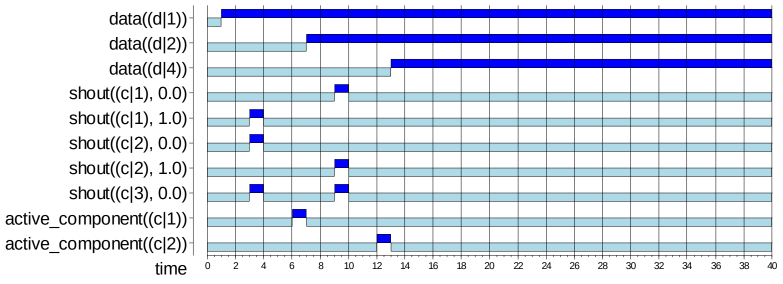


Fig. 32. Pandemonium trace 4.

Trace Example 5 – Simple Merge

Figure 33 shows a simulation trace of the pandemonium algorithm for example 5. Here, first C1 is activated, followed by C3.

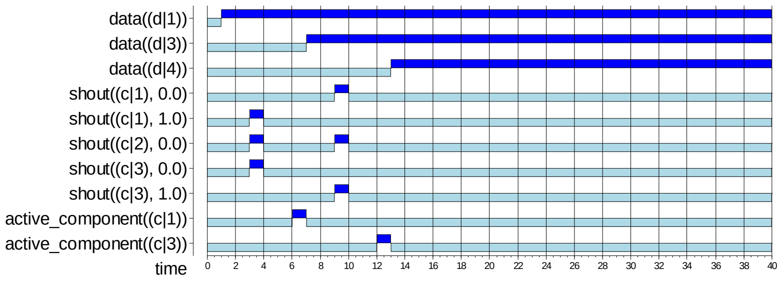


Fig. 33. Pandemonium trace 5.

Trace Example 6 – Multi Choice

Figure 34 shows a simulation trace of the pandemonium algorithm for example 6. Here, first C1 is activated, followed by both C2 and C3.

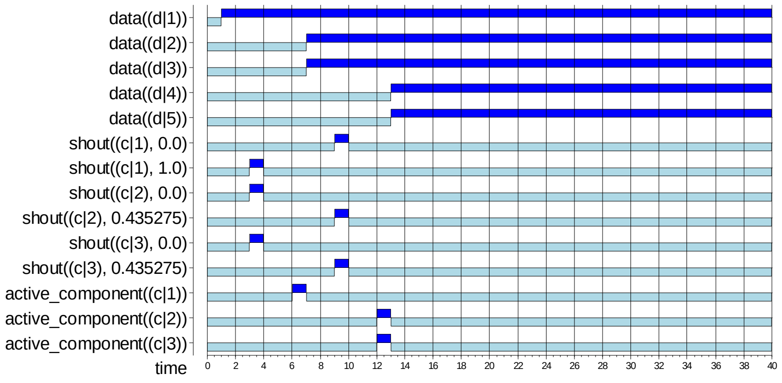


Fig. 34. Pandemonium trace 6.

Trace Example 7 – Simple Merge

Figure 35 shows a simulation trace of the pandemonium algorithm for example 7. Here, first C1 is activated, then both C2 and C3 are activated, and finally C6 (which is a specific variant of C4, see the description of the example) is activated.

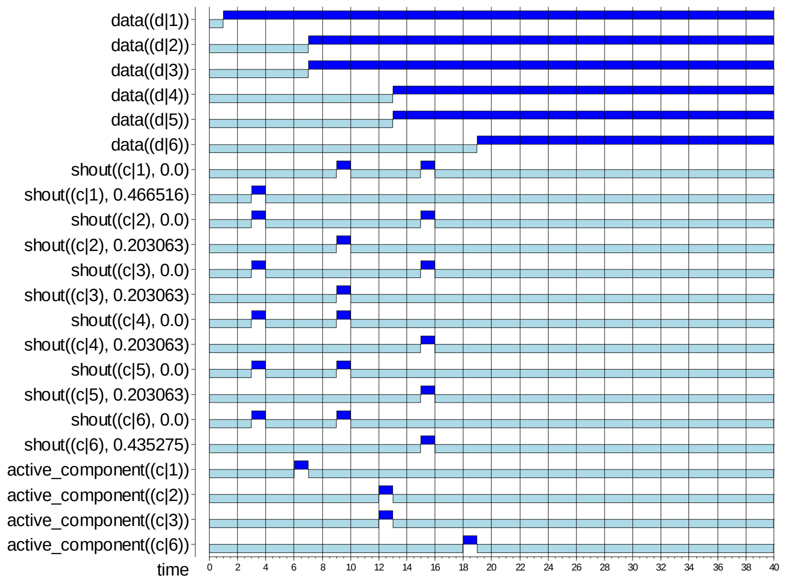


Figure 35. Pandemonium trace 7.

9 Voting Simulation

9.1 Simulation Model

Generic Part

The algorithm used for the Voting Mechanism is similar to the description in Section 5. The following procedure is assumed: at the beginning of the process, only the initial data is placed at the "shared repository" (see Section 6). Whenever new data has been added to the repository, a new round starts in which all agents can *vote*. The idea is that each agent can vote on only one agent (possibly on itself). After all agents have voted, the votes are counted, and the agent with most votes will be allowed to start reasoning. In case two agents have an equal amount of votes, then either the first agent, or the second agent, or both are activated (this decision is made randomly, with equal probabilities). When an agent is activated, this results in the agent adding its output data to the repository, and the start of a new round.

The LEADSTO specification for the algorithm can be found in Appendix C3. To model this algorithm in LEADSTO, the following ontology is used (again, only the elements that are shown in the traces are mentioned):

Table 3. Relations used within the voting simulation model

Relation	Explanation
data: DATA	This specifies that a certain type of data is present in the repository.
vote_for: COMPONENT x COMPONENT	An agent votes for a certain (other) agent.
active_component: COMPONENT	An agent is activated.

Specific Part

To determine on whom they will vote, the agents make use of a *voting algorithm*. For different variants of the Voting model, different voting algorithms may be used. In the current model, each agent follows the following algorithm:

1. if my input is present, and my output is not, then I vote for myself
2. if my input is not present, and this input is generated by one other agent, vote for that agent
3. if my input is not present, and this input is generated by n>1 other agents, vote for one of those agents (at random)
4. if my output is present, and this output is used by one other agent, vote for that agent
5. if my output is present, and this output is used by n>1 other agents, vote for one of those agents (at random)
6. if my output is present, and this output is used by no other agents (i.e., it is part of the goal data), do not vote

Note that this algorithm assumes a *local* perspective of the agents. This means that each agent only has knowledge about itself and its direct neighbours. For example, each agent knows which other agents need the data that it produces as input, but does not know which data the other agents produce as output.

9.2 Simulation Traces

This section presents the simulation traces that have resulted from applying the voting algorithm to the examples presented in Section 6.

Trace Example 1 – Sequence

Figure 36 shows the simulation trace of the voting algorithm for example 1.

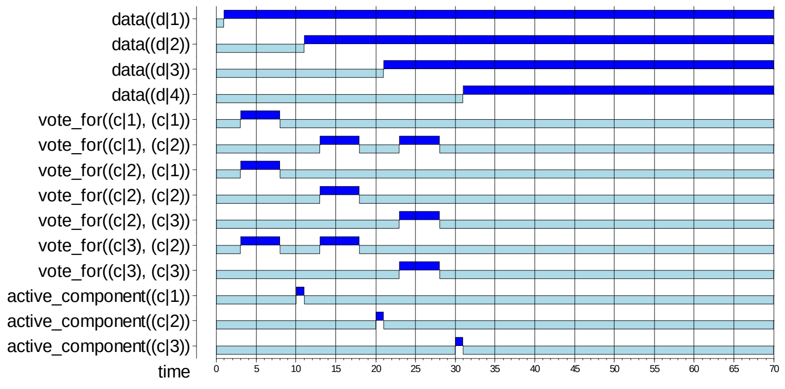


Fig. 36. Voting trace 1.

As can be seen in Figure 36, initially the only data that is present is d1:

data((d|1))

Based on this data, every agent starts voting:

vote_for((c|1), (c|1))
vote_for((c|2), (c|1))
vote_for((c|3), (c|2))

Agent C1 receives 2 votes, agent C2 receives one vote, and agent C3 receive no votes. Thus, agent C1 is selected to become active:

active_component((c|1))

As a result, agent C1 creates data d2, which is stored at the repository as well:

data((d|2))

Again, every agent starts voting:

vote_for((c|1), (c|2))
vote_for((c|2), (c|2))
vote_for((c|3), (c|2))

Agent C2 receives all 3 votes and is thus selected to become active:

active_component((c|2))

Next, agent C2 creates data d3, which is stored at the repository as well:

data((d|3))

Again, every agent starts voting:

vote_for((c|1), (c|2))
vote_for((c|2), (c|3))
vote_for((c|3), (c|3))

Agent C3 receives 2 votes, agent C2 receives one vote, and agent C1 receive no votes. Thus, agent C3 is selected to become active:

active_component((c|3))

Eventually, agent C3 creates data d4, which is stored at the repository as well:

data((d|4))

Since d4 is the goal data, at this point the process terminates.

Trace Example 2 – Parallel Split

Figure 37 shows a simulation trace of the voting algorithm for example 2. Here, first C1 is activated, then C2, and then C3.

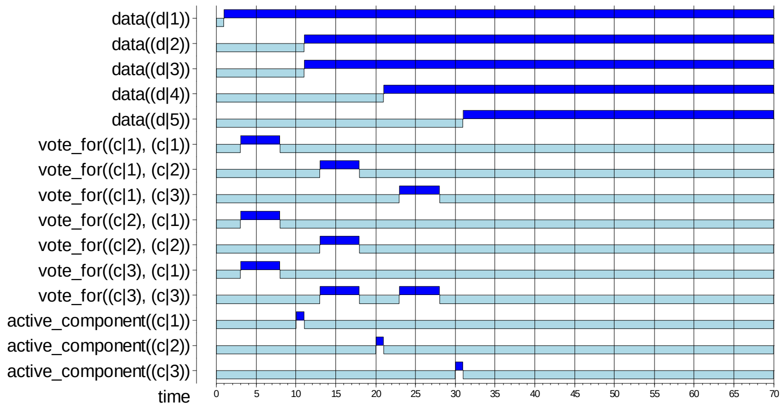


Fig. 37. Voting trace 2.

Trace Example 3 – Synchronization

Figure 38 shows a simulation trace of the voting algorithm for example 3. Here, first C1 is activated, then C2, and then C3.

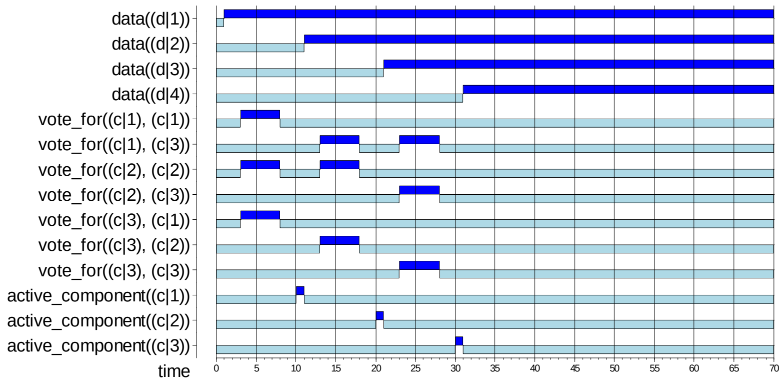


Fig. 38. Voting trace 3.

Trace Example 4 – Exclusive Choice

Figure 39 shows a simulation trace of the voting algorithm for example 4. Here, first C1 is activated, followed by C3.

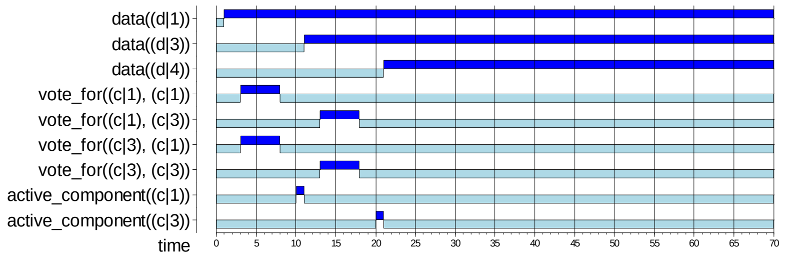


Figure 39. Voting trace 4.

Trace Example 5 – Simple Merge

Figure 40 shows a simulation trace of the voting algorithm for example 5. Here, first C2 is activated, followed by C3.

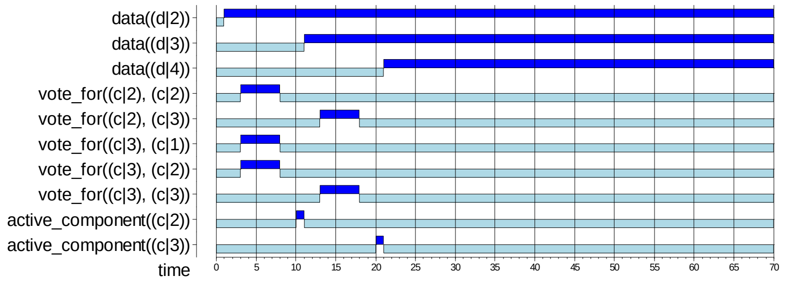


Fig. 40. Voting trace 5.

Trace Example 6 – Multi Choice

Figure 41 shows a simulation trace of the voting algorithm for example 6. Here, first C1 is activated, followed by C3.

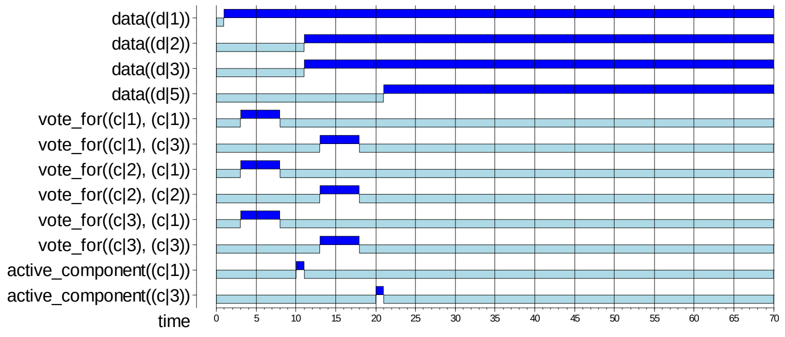


Fig. 41. Voting trace 6.

Trace Example 7 – Simple Merge

Figure 42 shows a simulation trace of the voting algorithm for example 7. Here, first C1 is activated, then C3 is activated, then C2 is activated, and finally C6 (which is a specific variant of C4, see the description of the example) is activated. Note that the current version of the model contains a small bug: in some cases multiple votes are made by one agent. In Figure 42, this can be seen in the third voting round. Here, agent C3 votes for C5 and C6 at the same time. In a next version of the model this bug will be removed.

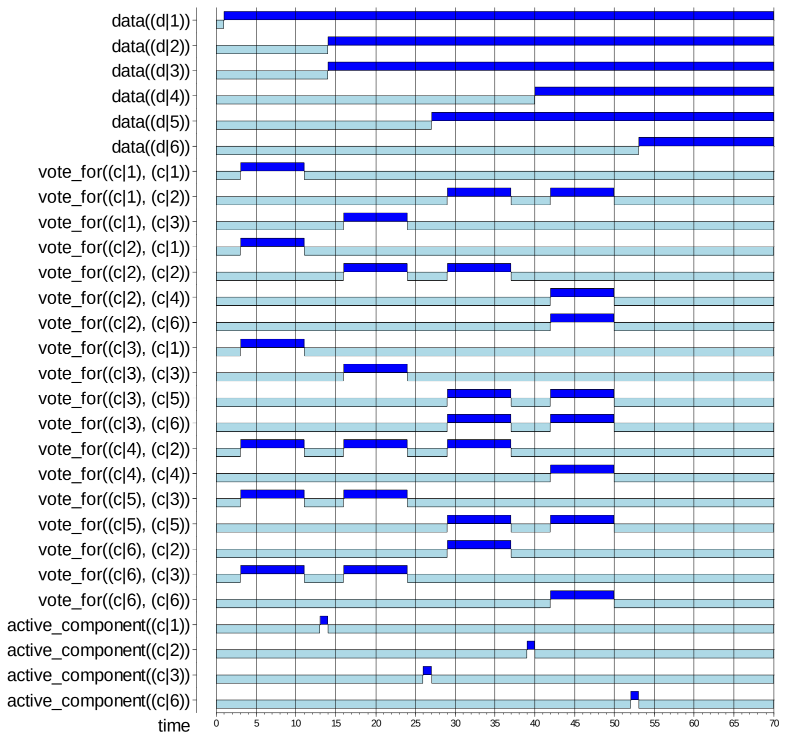


Fig. 42. Voting trace 7.

10 Simulation Evaluation

This section addresses the evaluation of the performance for the different algorithms that have been simulated in Section 7, 8, and 9. This evaluation can be performed from multiple perspectives. First of all, the achievement of the goals that have been set for the system are an important evaluation criterion. Secondly, an element in the evaluation is the efficiency of the algorithm. Finally, patterns can be specified which occur in the component specifications used for the algorithms, and it can be checked whether the coordination mechanism can indeed identify these patterns within the component specifications. To enable automated checking of the results of the algorithms, a formal specification of the different properties is required. For this purpose, the TTL language as described in Section 4.2 is used to formalise the desired properties. After such a formal description has been obtained, the automated TTL checker can be used to see how well the algorithm performs.

10.1 Successfulness

In the trace γ all goals g will eventually be derived:

successful_algorithm(γ:TRACE) ≡
∀t:TIME, d:DATA
[state(γ, t) |= goal(d) ⇒
∃t2:TIME [t2 ≥ t ∧ state(γ, t2) |= data(d)]]

The results of automatically checking the property against the traces that were generated in the simulation are shown in Table 4. A plus in the table indicates that the solution is found in all generated traces, a minus indicates that no solution is found in at least one of the generated traces.

Table 4. Successfulness of the different algorithms on the examples

Example	Behavior Networks	Pandemonium	Voting
Sequence	+	+	+
Parallel Split	+	+	+
Synchronization	+	+	+
Exclusive choice	+	+	+
Simple Merge	+	+	+
Multi Choice	+	+	+
Synchronizing merge	+	+	+

As can be seen, all algorithms eventually find the solution for the examples that have been used. More extensive testing is to be done to investigate under what circumstances the different algorithms fail to find the solution.

10.2 Efficiency

Efficiency can be viewed from multiple perspectives. First, one can look at the efficiency of the solution path found by the algorithm. It is for now assumed that each component takes an equal amount of time, and therefore the most efficient solution is simply the solution in which the least amount of components have been activated. Another way to describe efficiency can also be the efficiency of the algorithm itself, i.e. how much computation time does the algorithm need to generate a solution.

The approach taken in this section is to check whether the shortest path is used to reach the goals that are set. For the formalization of this property, it is for now assumed that the length of the shortest path is known for the particular example being checked:

efficient_algorithm(γ:TRACE, shortest_path:INTEGER) $\equiv$
successful_algorithm(γ) $\wedge$
component_activations(γ, shortest_path)

To enable a definition of the amount of activations of a component, first the activation of one component is defined, including its interval:

has_activation_interval(γ:TRACE, c:COMPONENT, tb:TIME, te:TIME) $\equiv$
tb < te $\wedge$
state(γ,te) |≠ activated(c)
[∀t tb≤t<te ⇒ state(γ,t) |= activated(c)] $\wedge$
∃t1<tb [∀t2 t1≤t2<tb ⇒ state(γ,t2) |≠ activated(c)]

An example of a definition for a trace with one component activation is shown below.

component_activations(γ:TRACE, 1) $\equiv$
∃c:COMPONENT, tb:TIME, te:TIME
has_activation_interval(γ, c:COMPONENT, tb:TIME, te:TIME) $\wedge$
[∀c2:COMPONENT, tb2:TIME, te2:TIME
    [has_activation_interval(γ, c2:COMPONENT, tb2:TIME, te2:TIME) ⇒
    c = c2 $\wedge$ tb = tb2 $\wedge$ te = te2]]

Table 5 shows the outcome of checking the property in the TTL Checker for the generated traces. Again, a plus indicates that in all generated traces the efficient solution was found, whereas a minus was put in the table in the other case.

Table 5. Efficiency of the different algorithms on the examples

Example	Behavior Networks	Pandemonium	Voting
Sequence	+	+	+
Parallel Split	+	+	-
Synchronization	+	+	+
Exclusive choice	+	+	+
Simple Merge	+	+	+
Multi Choice	-	-	+
Synchronizing merge	-	-	-

For the first five examples, both the behavior networks and the pandemonium always find the optimal path to the solution. For voting the optimal solution for the parallel split is not always found: apparently, there are situations when this approach is not efficient. An example of such a situation is the case that C1 and C2 have already been active, but C3 still has to be activated. In that case, C2 will not vote anymore, because its output data is part of the goal data (see the voting algorithm described in Section 9.1). Moreover, C3 will vote for itself, because its output data is not present yet. However, C1 could possibly vote for C2 (it does not know that C2 has already been active, because it has only *local* information). If this is the case, then C2 and C3 will both receive one vote, and C1 will receive no votes. As a result, it is possible that C2 is again selected (by random choice) to be active, although it has already been activated. Clearly, this is a very inefficient move. This problem could be solved by allowing a more global perspective for the agents.

For the multi choice and synchronizing merge, the behavior network fails to find the optimal solution in some cases. For the first, it activates both C2 and C3 whereas only one of the components is required to obtain the goal data. Adapting the parameters of the algorithm could prevent this from occurring. Furthermore, in the synchronizing merge case, both C2 and C3 are activated whereas C4 only needs one input to generate its output.

Also the Pandemonium model is not always efficient for the multi choice and synchronizing merge. For the multi choice, this is the case because the model sometimes generates traces where first C1 is activated, and then C2 and C3 are activated simultaneously. Although this solution is efficient in terms of activation rounds (i.e., only two rounds), it is not efficient in terms of component activations: three components are activated in total, where two activations would have been sufficient (i.e., C1 followed by C2, or C1 followed by C3). For the synchronizing merge, in some cases the same situation occurs as with the behavior network: sometimes both C2 and C3 are activated simultaneously, whilst only one of them is required.

The Voting model however succeeds in always finding the efficient solution for the multi choice. Here, the aforementioned situation that both C2 and C3 are activated never occurs, because there is always one component that receives more votes than the others. However, like the other approaches, the Voting model is sometimes inefficient with respect to the synchronizing merge. Here, again the same situation occurs as with the behavior network and the pandemonium: sometimes both C2 and C3 are activated, where only one of them is necessary.

10.3 Specifying and Checking Patterns

As has been mentioned, patterns can be specified for component examples and it can be checked whether these patterns are indeed found by the different algorithms. For the examples used in this document the component specifications originate from workflow patterns and therefore the patterns to be found within the component examples are precisely the workflow patterns from which these examples have been derived. Specification of patterns can be done from two perspectives: (1) exhaustively summing up all possible outcomes; (2) specifying the constraints between activation intervals of different components. For the second approach the interval relations as identified by Allen (ref) are used and specified in TTL:

```
before(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ e1 < b2
meets(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ e1 = b2
overlaps(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ b1 < b2 < e1 < e2
equals(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ b1 = b2 ∧ e1 = e2
starts(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ b1 = b2 ∧ e1 < e2
finished_by(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ b1 < b2 ∧ e1 = e2
contains(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ b1 < b2 ∧ e1 > e2
```

Below, the workflow patterns that have been used (1-7) are specified using TTL expressions. First, all traces are summed up in an informal fashion and thereafter the TTL expressions specifying the constraints between the activation intervals of the different components are shown.

Pattern 1 - Sequence

Possible traces:
ABC

Activation interval constraints in TTL:

```
∃bA,eA,bB,eB,bC,eC:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bA, eA, bB, eB) ∧
before(bB, eB, bC, eC)

/*
If desired, the following additional condition can be included to ensure that no other components are
activated during the trace:
∀c:COMPONENT ∀t1,t2:TIME
  [has_activation_interval(trace1, c, t1, t2) ⇒
     [c=A ∧ t1=bA ∧ t2=eA] ∨ [c=B ∧ t1=bB ∧ t2=eB] ∨ [c=C ∧ t1=bC ∧ t2=eC]]
*/
```

Pattern 2 - Parallel Split

Possible traces:
A[BC]
Note: [BC] means either simultaneously or in any order (= in theory, any of the possibilities before, meets, overlaps, equals, starts, finished_by, contains. However, in our current specifications (both Maes and Pandemonium) we do not handle parallelism. Thus, in the case of [BC] we will only generate the traces BC and CB).

Activation interval constraints in TTL:
∃bA,eA,bB,eB,bC,eC:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bA, eA, bB, eB) ∧
before(bA, eA, bC, eC)

Pattern 3 – Synchronization

Possible traces:
[AB]C

Activation interval constraints in TTL:
∃bA,eA,bB,eB,bC,eC:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bA, eA, bC, eC) ∧
before(bB, eB, bC, eC)

Pattern 4 - Exclusive Choice

Possible traces:
- AB
- AC

Activation interval constraints in TTL:
[∃bA,eA,bB,eB:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
before(bA, eA, bB, eB)]
∨
[∃bA,eA,bC,eC:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bA, eA, bC, eC)]

Pattern 5 - Simple Merge

Possible traces:
- AC
- BC

Activation interval constraints in TTL:
[∃bA,eA,bC,eC:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bA, eA, bC, eC)]
∨
[∃bB,eB,bC,eC:TIME
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, C, bC, eC) ∧
before(bB, eB, bC, eC)]

Pattern 6 - Multi Choice

Possible traces:

- AB
- AC
- A[BC]

Activation interval constraints in TTL:

parallel_split ∨ exclusive_choice

Pattern 7 - Synchronizing Merge

Possible traces:

- ABD
- ACD
- A[BC]D

Activation interval constraints in TTL:

```
[∃bA,eA,bB,eB,bD,eD:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, D, bD, eD) ∧
before(bA, eA, bB, eB) ∧
before(bB, eB, bD, eD)]
∨
[∃bA,eA,bC,eC,bD,eD:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, C, bC, eC) ∧
has_activation_interval(trace1, D, bD, eD) ∧
before(bA, eA, bC, eC) ∧
before(bC, eC, bD, eD)]
∨
[∃bA,eA,bB,eB,bC,eC,bD,eD:TIME
has_activation_interval(trace1, A, bA, eA) ∧
has_activation_interval(trace1, B, bB, eB) ∧
has_activation_interval(trace1, C, bC, eC) ∧
has_activation_interval(trace1, D, bD, eD) ∧
before(bA, eA, bB, eB) ∧
before(bA, eA, bC, eC) ∧
before(bB, eB, bD, eD) ∧
before(bC, eC, bD, eD)]
```

Table 6 shows whether the algorithms have indeed found the patterns (+) or whether there exists a trace in which the patterns was not found (-).

Table 6. Patterns found by the different algorithms within the examples

Example	Behavior Networks	Pandemonium	Voting
Sequence	+	+	+
Parallel Split	+	+	+/-*
Synchronization	+	+	+
Exclusive choice	+	+	+
Simple Merge	+	+	+
Multi Choice	+	+	+
Synchronizing merge	+	+	+

The behavior network, pandemonium, and voting algorithms always finds the patterns that have been identified. In the parallel split case the success of the voting algorithm however is debatable. The reason for this is that besides the expected patterns (A[BC]) also patterns such as A-B-B-C appear. According to personal communication with van der Aalst this is however not a violation of the pattern. Following this perspective a trace satisfies a pattern when the components as prescribed by the patterns also occur being active in the trace in the specified sequence. It is however allowed for other components (either a different component or activation of the same component at another time point) to be active within the same trace. For checking the more strict version (i.e. exactly the prescribed sequence without other activation) the *closed world assumption* version of the property (see description of pattern 1 in this section) can be used.

10.4 Comparison of Approaches

To conclude, the voting, pandemonium and behavior network algorithms have been thoroughly evaluated with respect to a number of relevant *performance indicators*, namely successfulness, efficiency, and pattern checks. It turned out that all algorithms found the solution in all cases. However, none of the algorithms is always efficient for all patterns. Both the behavior network and pandemonium algorithm perform equally well; they succeed for the “simple” cases and sometimes fail to be efficient for the two complicated cases (i.e. multi choice and synchronizing merge). Surprisingly, the voting algorithm always finds the most efficient solution for one of the complicated cases, namely the multi choice. It does however fail in the rather trivial case of the parallel split. All algorithms also find the patterns specified for each of the component examples. All and all, when comparing the algorithms, the performance based on the criteria specified above is almost similar. The way in which they find the component activation sequences is however completely different. The behavior networks algorithm needs a global overview of the system: it needs to know for each component what data it needs as input and what data it generates as output. Such a global view might not always be available or might be inconvenient. On the other hand, for the pandemonium a completely local view is sufficient, each agent only needs information about its own input and output data. In between is the voting algorithm, which needs information about itself and its direct neighbours. When comparing the algorithms on required computation time, the behavior networks take far more computation time compared to the other approaches. This has two causes: first, due to the fact that all global information is used within the algorithm; it has a lot more information to take into consideration. Second, both for the voting and pandemonium algorithm the calculations per agent can be performed in parallel which can not be done in the behavior networks algorithm.

11 Discussion

The work reported in this document has increased insight in the area of coordination of complex software systems in a number of respects. Moreover, based on the experiences during this project, a number of ideas for further steps to be undertaken have been developed.

What has been found this far

The following was gained by the work as reported. First, the following problems to obtain a coordination specification for a more complex component-based system were identified.

- it can become *large* and *intransparent*.
- it may suffer from *overspecification*, i.e., the dynamics of component activation may have to be prescribed in much more detail than actually needed
- it may require quite an *effort* to acquire the control knowledge, i.e., find out how the control choices should be in all possibly occurring cases
- its *flexibility* and *adaptivity* with respect to circumstances at runtime often is limited

To address these problems, specific coordination approaches borrowed from other disciplines have been explored and found to be useful for the area of coordination of complex software systems. It turns out that, in the form as used (kept close to the original description in the literature), each of these approaches to coordination may have its value. At least the first three of the identified problems are addressed by such coordination approaches.

Concerning differences between the considered coordination approaches, it can be concluded that an advantage of the pandemonium approach is its locality: for this algorithm, each component only needs information about its own input and output data, not about other data. The behavior networks algorithm and (to a certain extent) the voting algorithm do not have this advantage. In these algorithms, the components involved need to have more global information. This is a potential drawback, since global information may not always be (easily) available.

For the chosen method to explore different coordination approaches, it has been found that the simulation approach is quite useful. Within a reasonable time a nontrivial number of approaches can be evaluated against a nontrivial number of cases: 3 x 7 = 21 combinations have been explored. Similarly, the analysis of simulation results based on automated support for the evaluation of properties in traces has turned out useful.

Furthermore, workflow patterns turned out a useful source for cases to be explored, although their specification needs also to cover data flow aspects. It was not too difficult to add such data flow aspects.

Possible further steps

The work as reported has also led to a number of ideas for further research. First, while the specific coordination approaches borrowed from other disciplines were found to have value, no attempts have been made yet to come up with refinements, extensions or improvements of these approaches, or, inspired by these approaches, to design completely new approaches. It may well be possible to design approaches that perform still better. A number of possible extensions to take into account are the following:

- Experimenting with more different parameter settings in the current coordination approaches (for example, changing the β values in the Pandemonium algorithm).
- Actually changing the algorithms (for example, in the shout function of the Pandemonium algorithm, replace the '*' operators by '+', or enabling the voting algorithm to have more global information).

- Taking preferences for certain components into account. For example, suppose that two rather similar components are allowed to be activated, but only one of them is needed. In that case, it would be useful if there were a specific criterion by which one of them could be selected. An example of such a criterion is the quality of the component, or its expected execution time.
- Allowing a dynamic environment, which requires updating of information. For example, the aspect that certain information loses its value after some time, because it is no longer up to date, is an interesting one.
- Allowing the components to reason with partial data. For example, suppose that a component has d1 and d2 as its input data, but that only d1 is present. Then it would be useful if the component could already start reasoning with d1.
- Adding loops to the test examples. This would place additional demands on the coordination approaches in the sense that they will have to deal with multiple activations of the same component.
- Taking into account how recently a component has been active. This would be a useful addition in combination with the above issue of multiple activations. For example, in that case, it might be undesirable if components are activated twice in a row.

Adding these kinds of elements will definitely pose further challenges for any coordination approach.

Moreover, a special category of approaches that can be considered are adaptive control approaches. Such approaches do not need a prespecified coordination specification, but create one during processing. This option would solve the fourth issue mentioned above, and is an interesting area to investigate further.

In addition, to specify a coordination approach, two distinctions might be useful to make:

- the distinction between object information and processing, and coordination information and processing
- the distinction between a coordination specification and a generic coordination algorithm that acts as an interpreter on the coordination specification

The first distinction is an extension of the classical distinction between control (flow) and data (flow). The second distinction allows to separate a more declarative part of a coordination specification from the dynamics of its use. Such a coordination specification may involve declarative representation of properties of components such as the quality of a component, for example, its reliability, processing speed, and the uncertainty of its output.

Another area that can be investigated further is compositionality of coordination specification. It is possible to proceed as follows. First, extend workflow patterns to software coordination patterns by adding data flow aspects (for example, specified in Petrinet style). Next, use these as building blocks to create an overall control specification, based on some principles of compositionality.

Finally, still another area that can be investigated is verification and validation of a coordination specification. More specific evaluation criteria can be developed. For example, it can be automatically verified whether for each occurring system state, there is a next step that can be undertaken. Last but not least, after a more focused view has been developed on a coordination approach to be adopted, testing it for a real software system would be interesting.

12 References

Aalst, W.M.P. van der, Hofstede, A.H.M. ter, Kiepuszewski, B., and Barros, A.P., (2002). Workflow Patterns. QUT Technical report. FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.

Alpaydin, E., Kaynak, C., and Alimoglu, F., (2000). Cascading multiple classifiers and representations for optical and pen-based handwritten digit recognition. In *International Workshop on Frontiers of Handwriting Recognition VII*, p. 453-462, 2000.

Black, D., (1968). *The Theory of Committees and Elections*. Cambridge University Press, 1968.

Borda, J.-C. de, (1781). *Memoire sur les Elections au Scrutin*. Histoire de l'Academie Royale des Sciences, Paris, 1781.

Bosse, T., Jonker, C.M., Mey, L. van der, and Treur, J., (2005). LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn. In: Eymann, T., et al. (eds.), *Proc. of the Third German Conference on Multi-Agent System Technologies, MATES'05*. Lecture Notes in Artificial Intelligence, vol. 3550. Springer Verlag, 2005, pp. 165-178.

Brazier, F.M.T., Dunin-Keplicz, B., Jennings, N.R., and Treur, J., (1995). Formal Specification of Multi-Agent Systems: a Real World Case. In: Lesser V. (ed.), *Proceedings of the First International Conference on Multi-Agent Systems, ICMAS'95*, MIT Press, Menlo Park, VS, 1995, pp. 25-32.

Brazier, F.M.T., Dunin Keplicz, B., Jennings, N., and Treur, J., (1997). DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, vol. 6, Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, (M. Huhns and M. Singh, eds.), 1997, pp. 67-94.

Brazier, F.M.T., Jonker, C.M., and Treur, J., (1998/2004). Principles of Compositional Multi-agent System Development. In: J. Cuena (ed.), *Proceedings of the 15th IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, IT&KNOWS'98,* 1998, pp. 347-360. Published as: Principles of Component-Based Multi-Agent Systems Design. In: Cuena, J., Demazeau, Y., Garcia, A., and Treur, J. (eds.), *Knowledge Engineering and Agent Technology*. IOS Press, 2004, pp. 89-113.

Brazier, F.M.T., Jonker, C.M., and Treur, J., (2002a). Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, vol. 41, 2002, pp. 1-28.

Brazier, F.M.T., Jonker, C.M., and Treur, J., (2002b). Dynamics and Control in Component-Based Agent Models. *International Journal of Intelligent Systems*, vol. 17, 2002, pp. 1007-1048.

Brazier, F.M.T., Treur, J., Wijngaards, N.J.E., and Willems, M., (1999). Temporal semantics of compositional task models and problem solving methods. *Data and Knowledge Engineering*, vol. 29, 1999, pp. 17-42.

Breiman, L., (1994). *Bagging predictors*. Technical Report 421, Department of Statistics, University of California, Berkeley, California 94720, USA, September 1994.

Dietterich, T.G., (2000). Ensemble methods in machine learning. In J. Kittler and F. Roli, editors, *Multiple Classifier Systems*, p. 1-15, 2000.

Franklin, S., (1997). Artificial Minds, MIT Press, Cambridge Massachusetts, 1997.

Gavrila, I.S., and Treur, J., (1994). A formal model for the dynamics of compositional reasoning systems. In: A.G. Cohn (ed.), Proc. of the 11th European Conference on Artificial Intelligence, ECAI'94, Wiley and Sons, 1994, pp. 307-311.

Giacinto, G., and Roli, F., (2000). Dynamic classifier selection. In *Multiple Classifier Systems*, p. 177-189, 2000.

Günter, S., and Bunke, H. (2004). Optimization of weights in a multiple classifier handwritten word recognition system using a genetic algorithm. *Electronic Letters on Computer Vision and Image Analysis*, 3(1):25-41, 2004.

Ho, T. K., (1992). *A Theory of Multiple Classifier Systems And Its Application to Visual Word Recognition*. PhD thesis, Graduate School of State University of New York, Buffalo, May 1992.

Ho, T. K., Hull, J. Jj., and Srihari, N., (1994). Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66-75, 1994.

Jackson, J.V., (1987). Idea for a Mind, *SIGGART NEwsletter*, no 181, pp. 23-26, 1987.

Kittler, J., and Alkoot, F. M., (2001). Relationship of sum and vote fusion strategies. In J. Kittler and F. Rolli, editors, *Multiple Classifier Systems*, p. 339-348. Springer, 2001.

Kuncheva, L. I., (2002). A theoretical study on six classifier fusion strategies. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(2):281-286, Febuary 2002.

Lam, L., and Suen, C. Y., (1995). Optimal combinations of pattern classifiers. *Pattern Recognition Letters*, 16(9):945-954, 1995.

Langevelde, I.A. van, Philipsen, A.W., and Treur J., (1992). Formal Specification of Compositional Architectures. In: B. Neumann (ed.), Proc. of the 10th European Conference on Artificial Intelligence, ECAI'92, Wiley and Sons, 1992, pp. 272-276.

Lindsay, P. H., and Norman, D. A., (1977). *Human Information Processing: An Introduction to Psychology*. Academic Press, Inc., New York, 1977.

Maes, P., (1989). How to do the right thing. Connection Science, 1989. 1(3): p. 291-323.

Nanson, E. J., (1882). Methods of election. In *Transactions and Proceedings of the Royal Society of Victoria*, *18*, 197-240, 1882.

Minsky, M., (1985). Society of the Mind, Simon and Schuster, New York, 1985.

Ornstein, R., (1986). Multimind, Houghton Mofflin, Boston, 1986.

Powalka, R. K., Sherkat, N., and Whitrow, R. J., (1995). Multiple recognizer combination topologies. In M. L. Simner (ed.), *Basic and Applied Issues in Handwriting and Drawing Research*, p. 128-129, 1995.

Selfridge, O. G., (1958). Pandemonium: a paradigm for learning in mechanization of thought processes. In *Proceedings of a Symposium Held at the National Physical Laboratory*, pages 513-526, London, November 1958.

Vuurpijl, L. G., and Schomaker, L. R. B., (1998). Multiple-agent architectures for the classification of handwritten text. In *International Workshop on Frontiers of Handwriting Recognition VI*, p. 335-346, August 1998.

Vuurpijl, L. G., and Schomaker, L. R. B., (2000). Two-stage character classification: A combined approach of clustering and support vector classifiers. In *International Workshop on Frontiers of Handwriting Recognition VII*, p. 423-432, 2000.

Xu, L., Krzyzak, A., and Suen, C. Y., (1992). Methods of combining multiple classifiers and their applications to handwriting recognition. *IEEE Transactions on Systems, Man and Cybernetics*, 22(3):418-435, 1992.

A Leadsto paper

LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn

Tibor Bosse[1], Catholijn M. Jonker[2], Lourens van der Meij[1], and Jan Treur[1]

[1] Vrije Universiteit Amsterdam, Department of Artificial Intelligence,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
`{tbosse, lourens, treur}@cs.vu.nl`
`http://www.cs.vu.nl/~{tbosse, lourens, treur}`
[2] Nijmegen Institute for Cognition and Information, Division Cognitive Engineering,
Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
`C.Jonker@nici.kun.nl`

Abstract. This paper presents the language and software environment LEADSTO that has been developed to model and simulate the dynamics of Multi-Agent Systems (MAS) in terms of both qualitative and quantitative concepts. The LEADSTO language is a declarative order-sorted temporal language, extended with quantitative means. Dynamics of MAS can be modelled by specifying the direct temporal dependencies between state properties in successive states. Based on the LEADSTO language, a software environment was developed that performs simulations of LEADSTO specifications, generates simulation traces for further analysis, and constructs visual representations of traces. The approach proved its value in a number of projects within different domains of MAS research.

1 Introduction

Two important phases in the development of Multi-Agent Systems are the Design phase and the Implementation phase. In principle, the result of the Design phase is a high-level description (a model) of the system to be developed which, when encoded in some programming language, solves a particular problem. To this end, the problem is decomposed into modules, of which the functions and interfaces are specified in detail [10]. Then, the result of the Design phase, the (technical) specification, can serve as a starting point for the Implementation phase. However, an important problem is the validation of this specification: can it be proven that the specification shows the expected behaviour (e.g. as described by requirements) before it is actually implemented? Especially when the specification is given in terms of abstract high-level concepts this is a non-trivial task.

To contribute to the validation of Multi-Agent System specifications, this paper introduces the language and software environment LEADSTO. LEADSTO can be used to model the *dynamics* of systems to be designed, on the basis of highly abstract process descriptions. If those dynamics are modelled correctly, the LEADSTO software environment can use them for *simulation* of the desired behaviour of the system. Although such simulations are no formal proof, they can contribute to an informal validation of the specification: by performing a number of simulations, it can be tested whether the behaviour of the specification is satisfactory. Therefore, LEADSTO may be an important tool to bridge the gap between the Design and the Implementation phase.

Generally, in simulations various formats are used to specify basic mechanisms or causal relations within a process, see e.g., [1], [5], [9]. Depending on the domain of application such basic mechanisms need to be formulated quantitatively or qualitatively. Usually, within a given application explicit boundaries can be given in which the mechanisms take effect. For example, "from the time of planting an avocado pit, it takes 4 to 6 weeks for a shoot to appear".

As mentioned above, in order to simulate a system to be designed, it is important to model its *dynamics*. When considering current approaches to modelling dynamics, the following two classes can be identified: *logic-oriented* modelling approaches, and *mathematical* modelling approaches, usually based on difference or differential equations. Logic-oriented approaches are good for expressing qualitative relations, but less suitable for working with quantitative relationships. Mathematical modelling approaches (e.g., Dynamical Systems Theory [9]), are good for the quantitative relations, but expressing conceptual, qualitative relationships is very difficult. In this article, the LEADSTO language (and software environment) is proposed as a language combining the specification of qualitative and quantitative relations.

In Section 2, the LEADSTO language is introduced. Section 3 provides examples from existing case studies in which LEADSTO has been applied. Section 4 describes the tools that support the LEADSTO modelling environment in detail. In particular, the LEADSTO Property Editor and the LEADSTO Simulation Tool are discussed. Section 5 compares the approach to related modelling approaches, and Section 6 is a conclusion.

2 Modelling Dynamics in LEADSTO

Dynamics can be modelled in different forms. Based on the area within Mathematics called calculus, the Dynamical Systems Theory (DST) [9] advocates to model dynamics by continuous state variables and changes of their values over time, which is also assumed continuous. In particular, systems of differential or difference equations are used. This may work well in applications where the world states can be modelled in a quantitative manner by real-valued state variables and the world's dynamics shows continuous changes in these state variables that can be modelled by mathematical relationships between real-valued variables.

Not for all applications dynamics can be modelled in a quantitative manner as required for DST. Sometimes qualitative changes form an essential aspect of the dynamics of a process. For example, to model the dynamics of reasoning processes in Intelligent Agents usually a quantitative approach will not work. In such processes states are characterised by qualitative state properties, and changes by transitions between such states. For such applications often qualitative, discrete modelling approaches are advocated, such as variants of modal temporal logic; e.g., [6]. However, using such non-quantitative methods, the more precise timing relations are lost too.

For the approach used in this paper, it was decided to consider time as continuous, described by real values, but to allow both quantitative and qualitative state properties. The approach subsumes approaches based on simulation of differential or difference equations, and discrete qualitative modelling approaches, but also combines them. For example, it is possible to model the exact (real-valued) time interval for which some qualitative property holds. Moreover, the relationships between states over time are described by either logical or mathematical means, or a combination thereof. This is explained below in more detail.

Dynamics is considered as evolution of states over time. The notion of state as used here is characterised on the basis of an ontology defining a set of properties that do or do not hold at a certain point in time. For a given (order-sorted predicate logic) ontology Ont, the propositional language signature consisting of all *state ground atoms* (or *atomic state properties*) based on Ont is denoted by APROP(Ont). The *state properties* based on a certain ontology Ont are formalised by the propositions that can be made (using conjunction, negation, disjunction, implication) from the ground atoms. A *state* S is an indication of which atomic state properties are true and which are false, i.e., a mapping S: APROP(Ont) $\rightarrow$ {true, false}.

To specify simulation models a temporal language has been developed. This language (the LEADSTO language) enables one to model direct temporal dependencies between two state properties in successive states, also called *dynamic properties*. A specification of dynamic properties in LEADSTO format has as advantages that it is executable and that it can often easily be depicted graphically. The format is defined as follows. Let α and β be state properties of the form 'conjunction of atoms or negations of atoms', and e, f, g, h non-negative real numbers. In the LEADSTO language the notation $\alpha \twoheadrightarrow_{e, f, g, h} \beta$ (also see Figure 1), means:

If state property α *holds for a certain time interval with duration g, then after some delay (between e and f) state property* β *will hold for a certain time interval of length h.*

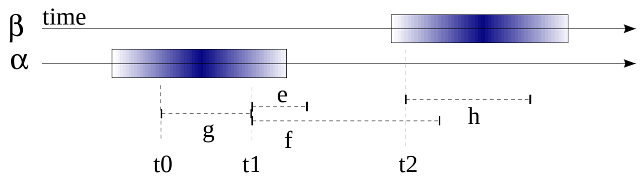


Fig. 1. The timing relationships

An example dynamic property that uses the LEADSTO format defined above is the following: "observes(agent_A, food_present) $\twoheadrightarrow_{2,\ 3,\ 1,\ 1.5}$ belief(agent_A, food_present)". Informally, this example expresses the fact that, if agent A observes that food is present during 1 time unit, then after a delay between 2 and 3 time units, agent A will believe that food is present during 1.5 time units. In addition, within the LEADSTO language it is possible to use sorts, variables over sorts, real numbers, and mathematical operations, such as in "has_value(x, v) $\twoheadrightarrow_{e, f, g, h}$ has_value(x, v*0.25)".

Next, a *trace* or *trajectory* γ over a state ontology Ont is a time-indexed sequence of states over Ont (where the time frame is formalised by the real numbers). A LEADSTO expression $\alpha \twoheadrightarrow_{e, f, g, h} \beta$, holds for a trace γ if:

$\forall$t1: [$\forall$t [t1–g $\leq$ t < t1 $\Rightarrow$ α holds in γ at time t] $\Rightarrow$ $\exists$d [e $\leq$ d $\leq$ f & $\forall$t' [t1+d $\leq$ t' < t1+d+h $\Rightarrow$ β holds in γ at time t']

An important use of the LEADSTO language is as a specification language for simulation models. As indicated above, on the one hand LEADSTO expressions can be considered as logical expressions with a declarative, temporal semantics, showing what it means that they hold in a given trace. On the other hand they can be used to specify basic mechanisms of a process and to generate traces, similar to Executable Temporal Logic (cf. [1]).

Finally, the LEADSTO format can be graphically depicted in a causal graph-like format, such as in Figure 2. Here, state properties are indicated by circles and LEADSTO relationships by arrows. An arc denotes a conjunction between state properties. Agents are indicated by dotted boxes. Circles that are depicted within an agent denote its internal (mental) state properties. Circles that are depicted on the left or right border of an agent denote, respectively, its input and output state properties, and circles that are depicted outside an agent denote state properties of the external world. Notice that this simple form leaves out the timing parameters e, f, g, h. A more detailed form can be obtained by placing the timing parameters in the picture as labels for the arrows. For more details about the LEADSTO language, see Section 4.

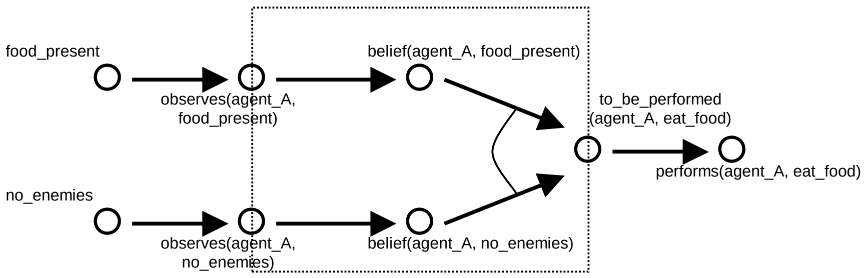

food_present
observes(agent_A, food_present)
belief(agent_A, food_present)
to_be_performed (agent_A, eat_food)
performs(agent_A, eat_food)
no_enemies
observes(agent_A, no_enemies)
belief(agent_A, no_enemies)

Fig. 2. Example of a graphical representation of two LEADSTO properties

3 Applications

The LEADSTO environment has been applied in a number of research projects in different domains. In this section, an example LEADSTO specification is given for a specific domain: a Multi-Agent System for ant behaviour, adopted from [3]. The world in which the ants live is described by a labeled graph as depicted in Figure 3. Locations are indicated by A, B,…, and edges by E1, E2,… The ants move from location to location via edges; while passing an edge, pheromones are dropped. The objective of the ants is to find food and bring this back to their nest. In this example there is only one nest (at location A) and one food source (at location F).

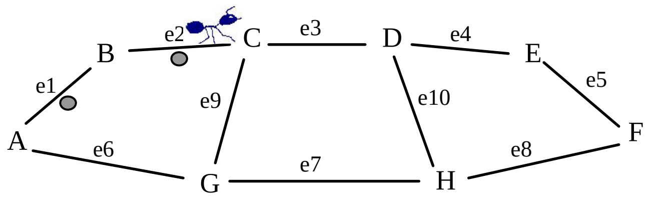


Fig. 3. An ants world

In [3], the dynamics of this system are formalised in LEADSTO, and some simulations are shown for different situations. A number of LEADSTO expressions that have been used for the simulation are shown in Box 1. For the complete specification, see [3].

In Figure 4 an example of a resulting simulation trace is shown. The upper part of the figure shows qualitative information; the lower part shows quantitative information. Time is on the horizontal axis. In the upper part, the state properties are on the vertical axis. Here, a dark box on top of the line indicates that the property is true during that time period, and a lighter box below the line indicates that the property is false. For example, the state property to_be_performed(ant2, pick_up_food) is true from time point 20 to 21. Because of space limitations, only a selection of important state properties was depicted. In the lower part, different instantiations of state property pheromones_at_E1(X) are shown, with different (real) values for X. For example, from time point 1 to 7 the amount of pheromones on E1 is 0.0.

LP5 (Selection of Edge)

This property models (part of) the edge selection mechanism of the ants. It expresses that, when an ant a observes that it is at location l coming from edge e0, and there are two other edges connected to that location, then the ant goes to the edge with the highest amount of pheromones. Formalisation:

observes(a, is_at_location_from(l, e0)) and neighbours(l, 3) and connected_to_via(l, l1, e1) and observes(a, pheromones_at(e1, i1)) and connected_to_via(l, l2, e2) and observes(a, pheromones_at(e2, i2)) and e0 ≠ e1 and e0 ≠ e2 and e1 ≠ e2 and i1 > i2 $\twoheadrightarrow_{0,0,1,1}$ to_be_performed(a, go_to_edge_from_to(e1, l1))

LP9 (Dropping of Pheromones)

This property expresses that, if an ant observes that it is at an edge e from a location l to a location l1, then it will drop pheromones at this edge e. Formalisation:

observes(a, is_at_edge_from_to(e, l, l1)) $\twoheadrightarrow_{0,0,1,1}$ to_be_performed(a, drop_pheromones_at_edge_from(e, l))

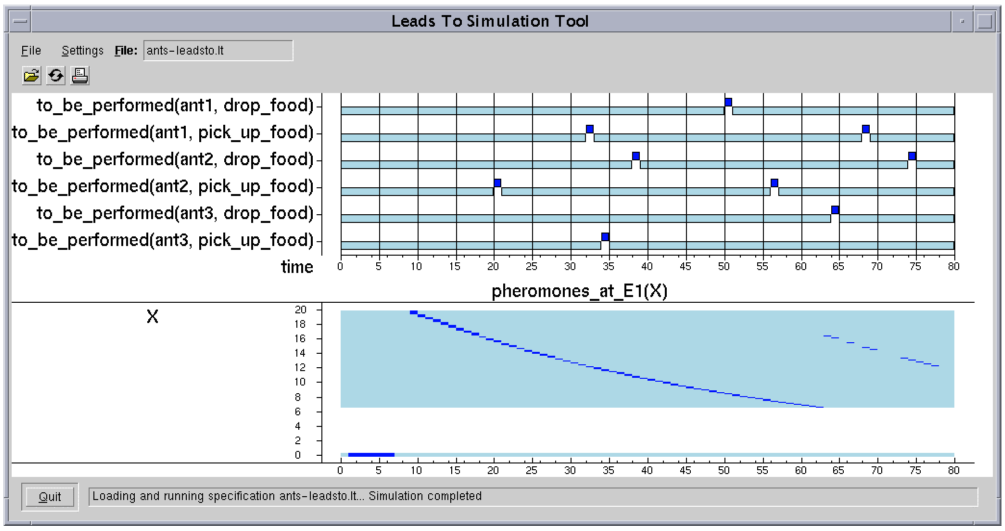
LP13 (Increment of Pheromones)

This property models (part of) the increment of the number of pheromones at an edge as a result of ants dropping pheromones. It expresses that, if an ant drops pheromones at edge e, and no other ants drop pheromones at this edge, then the new number of pheromones at e becomes i*decay+incr. Here, i is the old number of pheromones, decay is the decay factor, and incr is the amount of pheromones dropped. Formalisation:

to_be_performed(a1, drop_pheromones_at_edge_from(e, l1)) and ∀l2 not to_be_performed(a2, drop_pheromones_at_edge_from(e, l2)) and ∀l3 not to_be_performed(a3, drop_pheromones_at_edge_from(e, l3)) and a1 ≠ a2 and a1 ≠ a3 and a2 ≠ a3 and pheromones_at(e, i) $\twoheadrightarrow_{0,0,1,1}$ pheromones_at(e, i*decay+incr)

Box 1. Example LEADSTO specification

Although this picture provides a very simple example (involving only three ants), it demonstrates the power of LEADSTO to combine (real-valued) quantitative concepts with (conceptual) qualitative concepts.



Loading and running specification ants-leadsto.lt... Simulation completed

Fig. 4. Example simulation trace

Thus, Figure 4 shows an easy to read (important for the communication with the domain expert), compact, and executable representation of an informal model for ant behaviour.

Moreover, the example demonstrates the power of conceptual modelling based on highly abstract process descriptions. In less than 3 pages of code, the global dynamics of ant behaviour are so well defined that the specification actually runs. The specification took only a couple of days to construct, making the LEADSTO approach valuable for proof-of-concept simulations, thus important for Agent-Oriented Software Engineering.

Finally, note that the ant example does not fully exploit the power of to use real-valued time parameters (in fact, most of the rules use the values 0,0,1,1 for the parameters e, f, g, h, see Box 1). Nevertheless, in a number of other domains the use of real-valued time parameters turned out to be beneficial, since it allows for more realistic simulations of dynamic processes. An example domain where this was the case, is the domain of adaptive agents based on classical conditioning, see [2].

4 Tools

In this section, the LEADSTO software environment is presented. Basically, this environment consists of two programs: the *Property Editor* (a graphical editor for constructing and editing LEADSTO specifications) and the *Simulation Tool* (for performing simulations of LEADSTO specifications, generating data-files containing traces for further analysis, and showing traces). Apart from the LEADSTO language constructs introduced in Section 2 the LEADSTO software has a number of other language constructs. Section 4.1 discusses some details. Next, Section 4.2 introduces the Property Editor and Section 4.3 deals with the Simulation Tool. Section 4.4 describes the algorithm used to generate simulations. Finally, Section 4.5 provides some implementation details and discusses possible improvements for the future.

4.1 Details of the LEADSTO language

There are various representations of LEADSTO specifications. A graphical representation is shown in Section 4.2 when discussing the Editor. In this section all language constructs are discussed using a formal representation, based on the way specifications are stored.

Variables. The language uses typed variables in various constructs. A variable is represented as <Var-Name>:<Sort>.

Sorts. Sorts may be defined as a set of instances that may be specified: sortdef(<Sort-Name>, [<Term>,…]). There are also built-in sorts such as integer, real, and ranges of integers represented as for example between(2,10).

Atoms. Atoms may be terms built up from names with argument lists where each argument must be a term or a variable, for example: belief(x:AGENT, food_present).

LEADSTO rules. LEADSTO rules are introduced in Section 2. They are represented as:

leadsto([<Vars>,] <Antecedent-Formula>, <Consequent-Formula>, <Delay>, where
<Delay> := efgh(<E-Range>,<F-Range>, <G-Range>,<H-Range>))[1]
<Vars> := "[" <Variable>,... "]"

For example, $\alpha \twoheadrightarrow_{0, 0, 1, 1} \beta$ is represented as leadsto(alfa, beta, efgh(0,0,1,1)). Variables occurring in LEADSTO rules must be explicitly declared as <Variable> entries.

Formulae. LEADSTO rules contain formulae. The current implementation allows conjunctions and universal quantification over typed variables. Some variables are global, encom-

[1] The reason for grouping the delay is to make it easier to use delay constants.

passing the whole rule. Other - local - variables are part of universal quantification of some conjunction. The first kind of variables may be of infinite types. Currently, local variables must be of finite types. Some of these restrictions – such as on not allowing disjunction – will be removed in a next version. This will have no effect on the performance of the algorithm discussed in Section 4.4, but will make the details of the algorithm more complex. Other restrictions with respect to variables of infinite type will remain.

Time/Range. Time and Range values occurring in LEADSTO rules and interval constructs may be any number or expression evaluating to a number.

Constants. Constants may be defined using the following construct: constant(<Name>, <Value>). A constant(C1, a(1)) entry in a specification will lead to C1 being substituted by a(1) everywhere in the specification.

Intervals. During simulation, some atom values will be derived from LEADSTO rules. Others are not defined by rules but represent constant values of atoms over a certain time range. They are expressed as: interval([<Vars>,]<Range>,<LiteralConjunction>). Periodically reoccurring constant values are represented as: periodic([<Vars>,]<Range>,<Period>,<LiteralConjunction>), where

```
<Range> := range(<Start-Time>,<End-Time>)
<Vars>  := "[" <Variable>,... "]"
<Period> : an expression or constant or variable representing a number.
<LiteralConjunction> := <Literal> { and <Literal> }*
<Literal> := <Atom> | not <Atom>
```

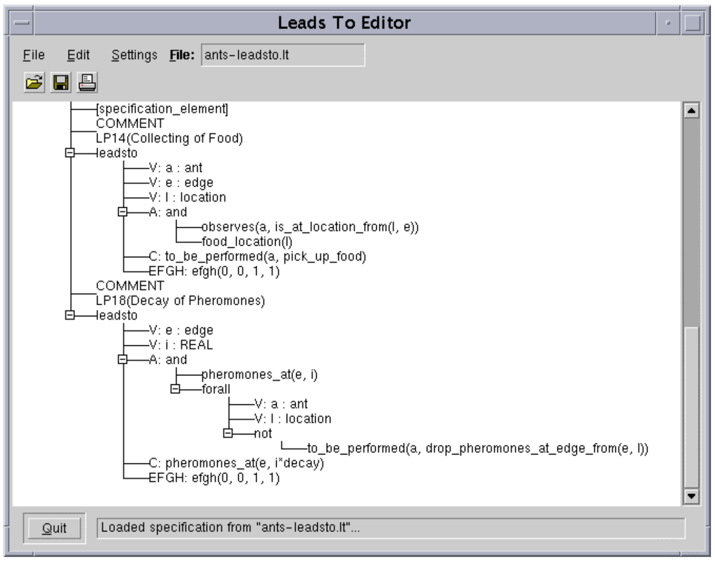
For example, an entry interval([X:between(1,2)], range(10,20), a(X)) makes a(1) and a(2) true in the time range (10,20). Likewise, an entry periodic(P, range(0,1), 10) makes P true in time ranges (0,1), (10,11), (20,21), and so on.

Simulation Range. The time range over which the simulation must be run is expressed by means of the constructs start_time(<Time>) and end_time(<Time>).

Visualisation of Traces. The construct display(<Tag-Name>, <Property>) is used to specify details of how to display the traces. The <Tag-Name> argument makes it possible to define multiple views of a trace. The active view may be specified from within the User Interface of the Simulation Tool. A number of properties may be specified, for showing or hiding certain atoms, for sorting atoms, for grouping atoms into a graph, and so on.

4.2 Property Editor

The Property Editor provides a user-friendly way of building and editing LEADSTO specifications. It was designed in particular for laymen and students. The tool has been used successfully by students with no computer science background and by users with little computer experience. By means of graphical manipulation and filling in of forms a LEADSTO specification may be constructed. The end result is a saved LEADSTO specification file, containing entries discussed in section 4.1. Figure 5 gives an example of how LEADSTO specifications are presented and may be edited with the Property Editor. This screenshot corresponds to (part of) the specification given in Box 1.



File: ants-leadsto.lt

```
[specification_element]
COMMENT
LP14(Collecting of Food)
leadsto
    V: a : ant
    V: e : edge
    V: l : location
    A: and
        observes(a, is_at_location_from(l, e))
        food_location(l)
    C: to_be_performed(a, pick_up_food)
    EFGH: efgh(0, 0, 1, 1)
COMMENT
LP18(Decay of Pheromones)
leadsto
    V: e : edge
    V: i : REAL
    A: and
        pheromones_at(e, i)
        forall
            V: a : ant
            V: l : location
            not
                to_be_performed(a, drop_pheromones_at_edge_from(e, l))
    C: pheromones_at(e, i*decay)
    EFGH: efgh(0, 0, 1, 1)
```

Loaded specification from "ants-leadsto.lt"...

Fig. 5. The LEADSTO Property Editor

4.3 Simulation Tool

Figure 6 gives an overview of the Simulation Tool and its interaction with the LEADSTO Property Editor.

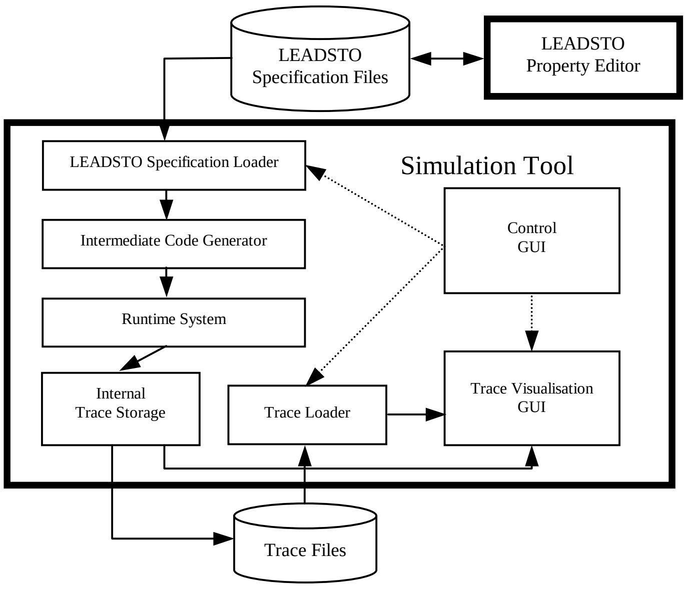


Fig 6. Simulation Tool Architecture

The bold rectangular borders define the separate tools. The lines with arrows represent data transport; the dashed arrows represent control. The Property Editor is used to generate and store LEADSTO specification files. The Simulation Tool loads these specification files. The overall control of the Simulation Tool is handled by the *Control-GUI* component. The Simulation Tool can perform the following activities:

- Loading LEADSTO specifications, performing a simulation and displaying the result.
- Loading and displaying existing traces (without performing simulation).
- Adjusting the visualisation of traces.

Loading and simulating a LEADSTO specification is handled in four steps:

1. The *Specification Loader* loads the specification.
2. The *Intermediate Code Generator* initialises the trace situation with values defined by interval and periodic entries in the specification. The LEADSTO rules are preprocessed: constants are substituted, universal quantifications are expanded and the rules are partially compiled into Prolog calls.
3. The actual simulation is performed by the *Runtime System*. This is the part that contains the algorithm, discussed in the next section.
4. At the end of a simulation the result is stored internally by the *Internal Trace Storage* component. The result can be saved as a trace file containing the evolution over time of truth values of all atoms occurring in the simulation, and will be visualised by the *Trace Visualisation GUI*. In principle, traces are three-valued, using the truth values true, false, and unknown. Saved trace files can be inspected later by the simulation tool and can be used by other tools, e.g., for automated analysis.

Note that visualisation of traces is integrated into the Simulation Tool through the *Trace Visualisation GUI* component. It is possible to select what atoms must be shown and in what order (sorting) etc. Figure 4 is an example of the visualisation of the result of a simulation.

4.4 Simulation Engine Algorithm

In this section a sketch of the simulation algorithm is given. The core of the semantics is determined by the LEADSTO rules, for example leadsto(alpha,beta, efgh(e, f, g, h)) or (in the notation of Section 2) $\alpha \twoheadrightarrow_{e, f, g, h} \beta$. The state properties α, β are internally normalised. Currently, only state properties that can be simplified to conjunctions of literals are allowed.

Restrictions on delays
The parameters g and h are time intervals, they must be >= 0. The algorithm allows only causal rules, e,f >= 0. Allowing e,f < 0 would lead to non-causal behaviour (any trace situation could have an effect arbitrarily in the past) and an awkward simulation algorithm. The causal nature of the semantics of LEADSTO rules results in a straightforward algorithm: at each time point, a bound part of the past of the trace (the maximum of all g values of all rules) determines the values of a bound range of the future trace (the maximum of f + h over all LEADSTO rules).

Outline of the algorithm
First all interval and periodic entries are handled by setting the ranges of atoms according to their definition. Next, for the algorithm a time variable HandledTime is introduced: all LEADSTO rules with antecedent range up to HandledTime have fired. The idea is to propagate HandledTime until HandledTime >= EndTime[2] via the following steps:

1. At a certain HandledTime, a value for NextTime is calculated. This will be the first time in the future after HandledTime that firing of a LEADSTO rule with its g-interval (see Figure 1) extending past HandledTime may have effect in the form of some conse-

[2] EndTime is the time up to which the simulation should be run.

quent atom set. The time increment will be at least as big as the minimum of e + h over all LEADSTO rules.

2. An (optional) Closed World Assumption is performed for all selected atoms in the range (HandledTime, NextTime), i.e., all unknown atoms in this range are made false.
3. All LEADSTO rules are applied for which the range of the antecedent ends before or overlaps with NextTime.
4. Set HandledTime := NextTime
5. Continue with step 1 until HandledTime >= EndTime

4.5 Implementation Details

The complexity of the current algorithm is proportional to the number of LEADSTO rules in the specification, to the number of incremental time steps of the algorithm (which is at most equal to the length of the simulation divided by the minimum of e + h over all LEADSTO rules) and (at most) to the number of matching antecedent atoms per LEADSTO rule (limited by the number of atoms set during the simulation). A number of optimizations already improve the performance, such as only considering antecedent atoms that have matching values in the (HandledTime, NextTime) time range and not considering LEADSTO rules that have been tested to not fire until some time in the future.

The software was written in SWI-Prolog/XPCE, and consists of approximately 20000 lines of code. The approach for the design and implementation has been to first focus on a complete implementation that is easily adaptable, with acceptable performance for the current users. For an impression of the performance: the simulation of Section 3 took two seconds on a regular Personal Computer. More complex LEADSTO simulations have been created that take about half an hour to run. For example: one simulation with 170 LEADSTO rules, 2000 time steps, with 15000 atoms set, took 45 minutes.

There is room for further performance improvement of the algorithm. One possible improvement is to increase the time increment NextTime – HandledTime introduced in the algorithm above. Global analysis of dependency of LEADSTO rules should improve the performance, for instance by trying to eliminate simple rules with small values of their e + h parameters. Furthermore, the LEADSTO language is being extended with constructs for probabilistic rules, and with constructs for systematically generating traces of LEADSTO specifications for a range of parameters.

5 Related Work

In the literature, a number of modelling approaches exist that have similarities to the approach discussed in this paper. Firstly, there is the family of approaches based on differential or difference equations (see, e.g., [9]). In these approaches, to simulate processes by mathematical means, difference equations are used, for example, of the form: $\Delta x = f(x)\,\Delta t$ or $x(t + \Delta t) = x(t) + f(x(t))\,\Delta t$. This can be modelled in the LEADSTO language as follows (where d is Δt): has_value(x, v) $\twoheadrightarrow_{d,\, d,\, d,\, d}$ has_value(x, v+f(v)*d). This shows how the LEADSTO modelling language subsumes modelling approaches based on difference equations. In addition to those approaches the LEADSTO language allows to express qualitative and logical aspects.

Another modelling approach, Executable Temporal Logic [1], is based on temporal logic formulae of the form $\varphi \;\&\; \chi \Rightarrow \psi$, where φ is a past formula, χ a present formula and ψ a future formula. In comparison to this format, the LEADSTO format is more expressive in the sense that it allows order-sorted logic for state properties, and allows one to express quantitative aspects. Moreover, the explicitly expressed timing parameters go beyond Executable Temporal Logic. On the other hand, within Executable Temporal Logic it is allowed to refer to different past states at different points in time, and thus to model more complex relationships over time. For the LEADSTO language the choice has been made to model only the basic mechanisms of a process (e.g., the direct causal relations), like in modelling approaches based on difference equations, and not to model the more complex mechanisms.

The Duration Calculus [11] is a modal logic for describing and reasoning about the real-time behaviour of dynamic systems, where states change over time and are represented by functions from time (reals) to the Boolean values (0 and 1). It is an extension of Interval Temporal Logic [7], but with continuous time, and uses integrated durations of states as interval temporal variables. Assuming finite variability of state functions (i.e., between any two time points only a finite number of state changes occurs), the axioms and rules of Duration Calculus constitute a complete logic (relative to Interval Temporal Logic). A number of interesting tools have been created around (subsets of) Duration Calculus, see, e.g., [8] for information on model checking duration calculus formulae. Duration Calculus itself is not directly used for creating executable models, but environments for executable code exist (e.g., PLC automata, see [4]) for which a semantics is given in Duration Calculus.

Another family of modelling approaches based on causal relations is the class of *qualitative reasoning* techniques (see, e.g., [5]). The main idea of these approaches is to represent quantitative knowledge in terms of abstract, qualitative concepts. Like the LEADSTO language, qualitative reasoning can be used to perform simulation. A difference with LEADSTO is that it is a purely qualitative approach, and that it is less expressive with respect to temporal and quantitative aspects.

6 Conclusion

This article presents the language and software environment LEADSTO that has been developed to model and simulate the dynamics of Multi-Agent Systems on the basis of highly abstract process descriptions. If those dynamics are modelled correctly, the LEADSTO software environment can use them for simulation of the desired behaviour of the system. Although such simulations are no formal proof, they can contribute to an informal validation of the specification: by performing a number of simulations, it can be tested whether the behaviour of the specification is satisfactory. Therefore, LEADSTO may be an important tool to bridge the gap between the Design and the Implementation phase.

Within LEADSTO, dynamics can be modelled in terms of both qualitative and quantitative concepts. It is, for example, possible to model differential and difference equations, and to combine those with discrete qualitative modelling approaches. Existing languages are either not accompanied by a software environment that allows simulation of the model, or do not allow the combination of both qualitative and quantitative concepts.

The language LEADSTO is a declarative order-sorted temporal language extended with quantitative notions (like integer, and real). Time is considered linear, continuous, described by real values. Dynamics can be modelled in LEADSTO as evolution of states over time, i.e., by modelling the direct temporal dependencies between state properties in successive states. The use of durations in these temporal properties facilitates the modelling of such temporal dependencies. In principle, accurately modelling the dynamics of processes may require the use of a dense notion of time, instead of the more practiced variants of discrete time. The problem in a dense time frame of having an infinite number of time points between any two time points is tackled in LEADSTO by the assumption of “Finite Variability” (see Section 5 and, e.g., [11]). Furthermore, main advantages of the LEADSTO language are that it is executable and allows for graphical representation.

The software environment LEADSTO was developed especially for the language. It features a dedicated Property Editor that proved its value for laymen, students and expert users. The core component is the Simulation Tool that performs simulations of LEADSTO specifications, generates simulation traces for further analysis, and visualises the traces.

The approach proved its value in a number of research projects in different domains. It has been used to analyse and simulate behavioural dynamics of agents in cognitive science (e.g., human reasoning, creation of consciousness, diagnosis of eating disorders), biology (e.g., cell decision processes, the dynamics of the heart), social science (e.g., organisation dynamics, incident management), and artificial intelligence (e.g., design processes, ant colony behaviour). LEADSTO is so rich that it can be used to model phenomena from diverse perspectives. It has, for example, been used to model cognitive processes from a psychological/BDI perspective and from a physical/neurological perspective. For more publications about these applications, the reader is referred to the authors' homepages.

References

1. Barringer, H., M. Fisher, D. Gabbay, R. Owens, & M. Reynolds (1996). *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press Ltd. and John Wiley & Sons.
2. Bosse, T., Jonker, C.M., Los, S.A., Torre, L. van der, and Treur, J., Formalisation and Analysis of the Temporal Dynamics of Conditioning. In: J.P. Mueller and F. Zambonelli (eds.), *Proc. of the Sixth Int. Workshop on Agent-Oriented Software Engineering, AOSE'05*. To appear, 2005.
3. Bosse, T., Jonker, C.M., Schut, M.C., and Treur, J, Simulation and Analysis of Shared Extended Mind. In: Davidsson, P., Gasser, L., Logan, B., and Takadama, K. (eds.), *Proc. of the First Joint Workshop on Multi-Agent and Multi-Agent-Based Simulation, MAMABS'04*, 2004, pp. 191-200.
4. Dierks, H. PLC-automata: A new class of implementable real-time automata. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development (ARTS'97)*, volume 1231 of Lecture Notes in Computer Science, pages 111-125. Springer-Verlag, 1997.
5. Forbus, K.D. *Qualitative process theory*. Artificial Intelligence, vol. 24, no. 1-3, 1984, pp. 85-168.
6. Meyer, J.J.Ch., and Treur, J. (volume eds.), *Agent-based Defeasible Control in Dynamic Environments.* Series in Defeasible Reasoning and Uncertainty Management Systems (D. Gabbay and Ph. Smets, series eds.), vol. 7. Kluwer Academic Publishers, 2002.
7. Moszkowski, B., and Manna, Z. Reasoning in Interval Temporal Logic. In Clarke, E., and Kozen, D., editors, *Proceedings of the Workshop on Logics of Programs*, volume 164 of LNCS, pages 371–382, Pittsburgh, PA, June 1983. Springer Verlag.
8. Pandya, P.K., Model checking CTL[DC]. In: *Proceedings of TACAS 2001*, Genova, LNCS 2031, Springer-Verlag, April 2001.
9. Port, R.F., Gelder, T. van (eds.) (1995). *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge, Mass.
10. Vliet, H., van. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd, 2000.
11. Zhou, C., Hoare, C.A.R., and Ravn, A.P. *A Calculus of Durations*, Information Processing Letter, 40, 5, pp. 269-276, 1991.

B TTL paper

A Temporal Trace Language for the Formal Analysis of Dynamic Properties

Tibor Bosse[1], Catholijn M. Jonker[2], Lourens van der Meij[1], and Jan Treur[1]

[1] Vrije Universiteit Amsterdam, Department of Artificial Intelligence,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
`{tbosse, lourens, treur}@cs.vu.nl`
`http://www.cs.vu.nl/~{tbosse, lourens, treur}`
[2] Radboud Universiteit Nijmegen, Nijmegen Institute for Cognition and Information,
Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
`C.Jonker@nici.ru.nl`

Abstract. Within many domains, among which biological and cognitive areas, multiple interacting processes occur with dynamics that are hard to handle. Current approaches to analyse the dynamics of such processes, often based on differential equations, are not always successful. As an alternative to differential equations, this paper presents the predicate logical Temporal Trace Language (TTL) for the formal specification and analysis of dynamic properties. This language supports the specification of both qualitative and quantitative aspects, and therefore subsumes specification languages based on differential equations. A special software environment has been developed for TTL, featuring both a Property Editor for building and editing TTL properties and a Checking Tool that enables the formal verification of properties against a set of traces. TTL has a number of advantages, among which a high expressivity and the possibility to define sublanguages for simulation and verification of entailment relations. TTL proved its value in a number of projects within different domains.

1 Introduction

Within many domains, among which biological and cognitive areas, multiple interacting processes occur with dynamics that are hard to handle. Modelling the dynamics of such processes poses real challenges for biologists and cognitive scientists. Currently, within the areas mentioned, differential equations are among the techniques most often used to address this challenge, with limited success. For example, in the area of intracellular processes, hundreds or more reaction parameters (for which reliable values are rarely available) are needed to model the processes in question (Teusink *et al.*, 2000). Thus, describing these processes in terms of differential equations can seriously compromise the feasibility of the model. Likewise, in the area of Cognitive Science it is advocated to take the Dynamical Systems Theory (DST, see e.g., Port and Gelder, 1995), which is also based on differential equations, as a point of departure. Many convincing examples have illustrated the usefulness of DST; however, they often only address lower-level cognitive processes such as sensory or motor processing. Areas for which a quantitative approach based on DST offers less are the dynamics of higher-level processes with mainly a qualitative character, such as reasoning, complex task performance, and certain capabilities of language processing.

As illustrated by these examples, within several disciplines it is felt that more abstract modelling techniques are required to cope with the complexity. This paper introduces the Temporal Trace Language (TTL) as such an abstract technique for the analysis of dynamic properties within complex domains.

In Section 2, some desiderata are put forward for a suitable approach for modelling complex dynamic processes, resulting in a novel perspective for the development of such an approach, based on the idea of checking dynamic properties on practically given sets of traces. Next, in Section 3, the basic concepts of the TTL language are introduced. In Section 4 it is shown how TTL can be used to express different kinds of dynamic properties. In Section 5, it is shown how dynamic properties that are expressed in related languages can be translated into TTL. Section 6 describes the tools that support the TTL modelling environment in detail. In particular, the TTL Property Editor and the TTL Checker Tool are discussed. Section 7 is a conclusion.

2 Perspective of this paper

As can be seen in the discussion about the different areas as given above, the demands for dynamic modelling approaches suitable for these areas are nontrivial. Such *desiderata for modelling languages* include:

(1) modelling at the right level of abstraction
(2) expressivity for logical relationships
(3) expressivity for quantitative relationships
(4) both discrete and continuous modelling
(5) difference and differential equations should be subsumed
(6) expressivity for dynamic properties of varying complexity, for example including adaptivity

Moreover, analysis techniques that would be desirable concern both the generation and formalisation of simulated and empirical trajectories or traces, as well as analysis of complex dynamic properties of such traces and relationships between such properties. Such *desiderata for analysis techniques* include:

(a) generating traces by simulation based on quantitative, continuous variables
(b) generating traces by simulation based on qualitative, logical notions
(c) formalisation of simulated or empirical traces
(d) analysis of properties of simulated traces
(e) analysis of properties of empirical traces
(f) analysis of relationships between (e.g., global and local) properties of traces

Taken together, the desiderata gathered above are not easy to fulfill. Sometimes they may even be considered mutually exclusive. On the one hand, high expressivity is desired, but on the other hand feasible analysis techniques are demanded. To make automated support for these analyses feasible, often the strategy is followed to limit the expressivity of the modelling language, thereby compromising on the first list of desiderata. For example, the expressivity is limited to difference and differential equations as in DST (excluding logical relationships, compromising at least (2)), or to propositional modal temporal logics (excluding numerical relationships, compromising at least (3), (5), (6)). In the former case calculus can be exploited to do simulation and analysis (Port and van Gelder, 1995), fulfilling (a) and (c) but not (b), (d), (e) and (f). In the latter case, for example, simulation can be based on a specific executable format (e.g., executable temporal logic (Barringer et al, 1996), fulfilling (b) and (c) but not (a), (d), (e) and (f)) and model checking techniques can be exploited for analysis of relationships between dynamic properties, fulfilling (d) to (f), e.g., (Clarke *et al.*, 2000; Manna and Pnueli, 1995; Stirling, 2001).

Within the literature on analysis of properties (verification), much emphasis is put on computation of entailment relations. This essentially means checking properties on the set of all theoretically possible traces of a process. To make that feasible, expressivity of the language for these properties has to be sacrificed to a large extent. However, checking properties on a practically given set of traces (instead of all theoretically possible ones) is computationally much cheaper, and therefore the language for these properties can be more expressive. Such a set can consist of one or a number of traces, obtained empirically or by simulation. By limiting the desiderata by giving up (f), but still keeping (c) to (e), a much more expressive language for properties can be dealt with; the sorted predicate logic temporal trace language TTL described in this paper is an example of this.

For simulation it is essential to have limitations to the language. Therefore, an *executable language* can be defined as a sublanguage of the *overall language* for analysis. Moreover, also *analysis languages* that allow analysis in the sense of (f) can be embedded in the overall language. Thus the picture shown in Figure 1 is obtained. At the top there is an expressive overall language, in our case TTL, which fulfills all of the desiderata for modelling languages, i.e., (1) to (6). Concerning the desiderata for analysis techniques, it fulfills (c) to (e), but sacrifices (a), (b) and (f). In addition, a sublanguage can be defined for execution (fulfilling (1) to (5) and (a) and (b)), and a sublanguage can be defined for analysis of relationships between properties in the sense of (f), thereby also fulfilling (1) and (2)[1]. For the case of TTL, one of the executable sublanguages that already exist is the LEADSTO language, cf. (Bosse *et al.*, 2005b). Moreover, for the sublanguage for analysis one could think of any standard temporal logic, such as LTL or CTL, see, e.g., (Benthem, 1983; Goldblatt, 1992).

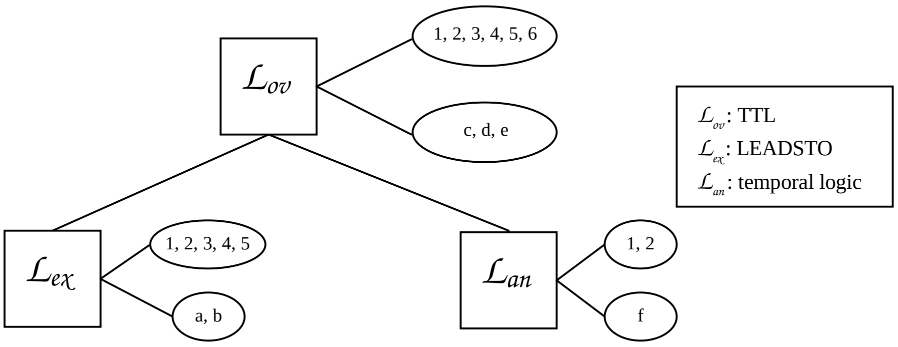


Figure 1. Embedding relationships between languages

Having the language for simulation and the languages for analysis within one subsuming language provides the possibility to have a declarative specification of a simulation model, and thus to involve a simulation model in logical analyses.

[1]In principle, such languages could also fulfill (6), but only to a certain extent. See Section 5.3 for an elaborate discussion.

3 Basic Concepts

To describe dynamics, the notion of state is important. Dynamics will be described in the next section as evolution of states over time. The notion of state as used here is characterised on the basis of an ontology defining a set of physical and/or mental (state) properties (following, among others, (Kim, 1998)) that do or do not hold at a certain point in time. These properties are often called state properties to distinguish them from dynamic properties that relate different states over time. A specific state is characterised by dividing the set of state properties into those that hold, and those that do not hold in the state. Examples of state properties are ‘the agent is hungry’, ‘the agent has pain’, ‘the agent's body temperature is 37.5° C’, or ‘the environmental temperature is 7° C’. Real value assignments to variables are also considered as possible state property descriptions. For example, in a DST approach based on variables x_1, x_2, x_3, x_4, that are related by differential equations over time, value assignments such as

$$x_1 \leftarrow 0.06$$
$$x_2 \leftarrow 1.84$$
$$x_3 \leftarrow 3.36$$
$$x_4 \leftarrow -\,0.27$$

are considered state descriptions. State properties are described by ontologies that define the concepts used.

3.1 Ontologies and State Properties

To formalise state property descriptions, ontologies are specified in a (many-sorted) first order logical format: an *ontology* is specified as a finite set of sorts, constants within these sorts, and relations and functions over these sorts (sometimes also called a signature). The example state properties mentioned above then can be defined by nullary predicates (or proposition symbols) such as hungry, or pain, or by using n-ary predicates (with $n \geq 1$) like has_temperature(body, 37.5), has_temperature(environment, 7), or has_value (x_1, 0.06).

For a given ontology Ont, the propositional language signature consisting of all *state ground atoms* based on Ont is denoted by At(Ont). The *state properties* based on a certain ontology Ont are formalised by the propositions that can be made (using conjunction, negation, disjunction, implication) from the ground atoms and constitute the set SPROP(Ont).

In many domains, it is desirable to distinguish different *agents* that are involved in the process under analysis. Moreover, it is often useful to distinguish between the internal, external, input, and output state properties of these agents. To this end, the following different types of ontologies are introduced:

- IntOnt(A): to express *internal* state properties of the agent A
- InOnt(A): to express state properties of the *input* of agent A
- OutOnt(A)): to express state properties of the *output* of the agent, and
- ExtOnt(A): to express state properties of the *external* world (for A)

For example, the state property pain may belong to IntOnt(A), whereas has_temperature(environment, 7), may belong to ExtOnt(A). The agent input ontology InOnt(A) defines properties for perception, the agent output ontology OutOnt(A) properties that indicate initiations of actions of A within the external world. The combination of InOnt(A) and OutOnt(A) is the *agent interaction ontology*, defined by InteractionOnt(A) = InOnt(A) $\cup$ OutOnt(A). The *overall ontology* for A is assumed to be the union of all ontologies mentioned above:

OvOnt(A) = InOnt(A) $\cup$ IntOnt(A) $\cup$ OutOnt(A) $\cup$ ExtOnt(A).

As yet no distinction between physical and mental internal state properties is made; the formal framework introduced in subsequent sections does not assume such a distinction. If no confusion is expected about the agent to which ontologies refer, the reference to A is sometimes left out.

3.2 Different Types of States

a) A *state* for a given ontology Ont is an assignment of truth-values {true, false} to the set of ground atoms At(Ont). The *set of all possible states* for an ontology Ont is denoted by STATES(Ont). In particular, STATES(OvOnt) denotes the set of all possible *overall states.* For the agent STATES(IntOnt) is the set of all of its possible *internal states*. Moreover, STATES(InteractionOnt) denotes the set of all *interaction states*.

b) The standard satisfaction relation $\models$ between states and state properties is used: S $\models$ p means that property p holds in state S. Here $\models$ is a predicate symbol in the language, usually used in infix notation, which is comparable to the Holds-predicate in situation calculus. For a property p expressed in Ont, the set of states over Ont in which p holds (i.e., the S with S $\models$ p) is denoted by STATES(Ont, p).

c) For a state S over ontology Ont with sub-ontology Ont', a restriction of S to Ont' can be made, denoted by S|Ont'; this restriction is the member of STATES(Ont') defined by S|Ont'(a) = S(a) if a $\in$ At(Ont'). For example, if S is an overall state, i.e., a member of STATES(OvOnt), then the restriction of S to the internal atoms, S|IntOnt is an internal state, i.e., a member of STATES(IntOnt). The restriction operator serves as a form of projection of a combined state onto one of its parts.

4 Expressing Dynamic Properties

To describe the (internal and external) dynamics of an agent, explicit reference is made to time. Dynamic properties can be formulated that relate a state at one point in time to a state at another point in time. Some examples of dynamic properties of a certain agent are shown below, using an informal (natural language) notation.

A simple example is the following dynamic property specification for belief creation based on observation:

Observational belief creation
'At any point in time t1 if the agent observes at t1 that it is raining, then there exists a point in time t2 after t1 such that at t2 the agent believes that it is raining'.

An example of another type is trust monotonicity; this dynamic property specification about the dynamics of trust over time involves the comparison of two histories:

Trust monotonicity
'For any two possible histories, the better the agent's experiences with public transportation, the higher the agent's trust in public transportation'.

These examples were kept simple; they are just meant as illustrations. No attempt was made to make them as realistic as possible. As will be explained below, TTL can be used to express such dynamic properties, and other, more sophisticated ones, in a formal manner. First, in

Section 4.1 the notion of trace is defined more explicitly. Next, in Section 4.2 it is shown in detail how dynamic properties can be expressed formally in TTL.

4.1 Time Frame and Trace

a) A fixed *time frame* T is assumed which is linearly ordered. Depending on the application, it may be dense (e.g., the real numbers), or discrete (e.g., the set of integers or natural numbers or a finite initial segment of the natural numbers), or any other form, as long as it has a linear ordering.

b) A *trace* γ over an ontology Ont and time frame T is a time-indexed set of states

$$\gamma_t \ (t \in T)$$

in STATES(Ont), i.e., a mapping

$$\gamma : T \rightarrow \text{STATES(Ont)}.$$

For the specification of dynamic properties, these definitions work fine. However, for some specific operations (such as verification), a dense time frame may cause problems, since it consists of an infinite number of time points. Therefore, in such cases *finite variability* of state functions is assumed (i.e., between any two time points only a finite number of state changes occurs). This is discussed in more detail in Section 6.

Traces can be visualised, for example as in Figure 2. Here, the time frame is depicted on the horizontal axis. The different predicates of the ontology are shown on the vertical axis. A dark box on top of the line indicates that the predicate is true during that time period, and a lighter box below the line indicates that it is false. Thus, in the example of Figure 2, predicate1 is true during the whole trace, predicate2 is true from time point 2.5 to time point 4.25, and predicate3 is true from time point 2 to 3 and from time point 8 to 10.

The set of all traces over ontology Ont is denoted by TRACES(Ont) , i.e., TRACES(Ont) = STATES(Ont)^{T}.

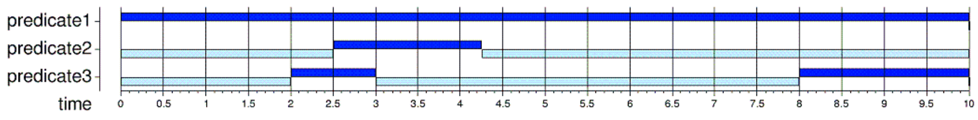


Figure 2. Example visualisation of a trace

c) A *temporal domain description* W is a given set of traces over the overall ontology, i.e.,

$$W \subseteq \text{TRACES(OvOnt)}.$$

This set represents all possible developments over time (respecting the world's laws) of the part of the world considered in the application domain.

Different traces with respect to an agent A can refer to different experiments with A involving different worlds, or different events generated in the world. For human beings one can think of a set of experiments in cognitive science, in which different experiments are not assumed to influence the behaviour of the agent. For software agents, it is possible to even erase the complete history (complete reset) and then activate the agent in a new world setting.

d) Given a trace γ over the overall ontology OvOnt, the *input state* of an agent A at time point t, i.e., γ_t |InOnt(A), is also denoted by

state(γ, t, input(A)).

Analogously, state(γ, t, output(A)) denotes the *output state* of the agent at time point t, state(γ, t, internal(A)) denotes the *internal state*, and state(γ, t, external(A)) denotes the *external world state*. If no confusion is expected about the particular agent, the reference to A can be left out, e.g., as in state(γ, t, input). Moreover, the overall state of a system (agent and environment) at a certain moment, is denoted by state(γ, t).

4.2 Dynamic Properties

To express dynamic properties in a precise manner, it is needed to make explicit references to time points and traces. Comparable to the approach in situation calculus, TTL is built on atoms referring to, e.g., traces, time and state properties. For example, 'in the output state of A in trace γ at time t property p holds' is formalised by

$\text{state}(\gamma, t, \text{output}(A)) \models p.$

Throughout the remainder of this paper, these kinds of atoms will be referred to as *Holds atoms*. Based on such Holds atoms, *Dynamic Properties* can be built using the usual logical connectives and quantification (for example, over traces, time and state properties). For example, the following dynamic properties can be expressed:

Observational belief creation

'In any trace, if at any point in time t1 the agent A observes that it is raining, then there exists a point in time t2 after t1 such that at t2 in the trace the agent A believes that it is raining'.

In formalised form:

$\forall\gamma \in W \;\; \forall t1$

$[\; \text{state}(\gamma, t1, \text{input}(A)) \models \text{observation\_result}(\text{itsraining})$

$\quad \Rightarrow \exists t2 \geq t1 \;\; \text{state}(\gamma, t2, \text{internal}(A)) \models \text{belief}(\text{itsraining}) \quad]$

Trust monotonicity

'For any two traces γ1 and γ2, if at each time point t the agent A's experience with public transportation in γ2 at t is at least as good as A's experience with public transportation in γ1 at t, then in trace γ2 at each point in time t, the A's trust is at least as high as A's trust at t in trace γ1'.

In formalised form:

$\forall\gamma 1, \gamma 2 \in W$

$[\forall t \;\; [\; \text{state}(\gamma 1, t, \text{input}(A)) \models \text{has\_value}(\text{experience}, v1)\; \&$

$\quad \text{state}(\gamma 2, t, \text{input}(A)) \models \text{has\_value}(\text{experience}, v2) \;\Rightarrow\; v1 \leq v2 \quad]$

$\Rightarrow$

$\forall t \;\; [\; \text{state}(\gamma 1, t, \text{internal}(A)) \models \text{has\_value}(\text{trust}, w1)\; \&$

$\quad \text{state}(\gamma 2, t, \text{internal}(A)) \models \text{has\_value}(\text{trust}, w2) \;\Rightarrow\; w1 \leq w2 \quad]\,]$

Instead of the term Dynamic Property, sometimes the term *TTL Formula* is used within this paper. This is especially the case in Section 6, where the focus is on the technical aspects of the language.

5 Relation to other Languages

In this section, TTL will be compared with a number of existing related languages. In Section 5.1 it is shown how differential equations can be modelled in TTL. In Section 5.2 it is shown how executable properties expressed in LEADSTO can be translated into TTL, and in Section 5.3 it is shown how properties expressed in standard Linear Temporal Logic (LTL) can be translated into TTL.

5.1 Expressing Difference and Differential Equations in TTL

As mentioned in the Introduction, especially in cognitive domains complex continuous relationships over time can be encountered. These relationships are often modelled semantically by differential equations, usually assumed to belong to the Dynamical Systems approach (DST), put forward, e.g., in (Port and Van Gelder, 1995). The question may arise whether or not such modelling techniques can be expressed in the Temporal Trace Language TTL. In this section it is shown how modelling techniques used in the dynamical systems approach, such as difference and differential equations, can be represented in TTL. First the discrete case is considered. An example of an application is the study of the use of logistic and other difference equations to model growth (and in particular growth spurts) of various cognitive phenomena, e.g., the growth of a child's lexicon between 10 and 17 months, cf. (Geert, 1995). The logistic difference equation used is:

$$L(n+1) = L(n)\ (1 + r - r\ L(n)/K)$$

Here r is the growth rate and K the carrying capacity. This equation can be expressed in our temporal trace language on the basis of a discrete time frame (e.g., the natural numbers) in a straightforward manner:

$\forall \gamma \in W\ \ \forall t$
    state(γ, t, internal) $\models$ has_value(L, v) $\Rightarrow$
    state(γ, t+1, internal) $\models$ has_value(L, v (1 + r - rv/K))

The traces γ satisfying the above dynamic property are the solutions of the difference equation. Another illustration is the dynamical model for decision-making presented in (Townsend and Busemeyer, 1995). The core of their decision model for the dynamics of the preference P for an action is based on the differential equation

$$dP(t)/dt = -s\ P(t) + c\ V(t)$$

where s and c are constants and V is a given evaluation function. One straightforward option is to use a discrete time frame and model a discretised version of this differential equation along the lines discussed above. However, it is also possible to use the dense time frame of the real numbers, and to express the differential equation directly. To this end, the following relation is introduced, expressing that $x = dy/dt$:

is_diff_of(γ, x, y) :
  $\forall t,w\ \ \forall \varepsilon>0\ \exists \delta>0\ \forall t',v,v'$
    $0 < dist(t',t) < \delta$ & state(γ, t, internal) $\models$ has_value(x, w)
        & state(γ, t, internal) $\models$ has_value(y, v)
        & state(γ, t', internal) $\models$ has_value(y, v')
        $\Rightarrow$ $dist((v'-v)/(t'-t),w) < \varepsilon$

where $dist(u,v)$ is defined as the absolute value of the difference, i.e. $u-v$ if this is ≥ 0, and $v-u$ otherwise. Using this, the differential equation can be expressed by:

is_diff_of(γ, - s P + c V, P)

The traces γ for which this statement is true are (or include) solutions for the differential equation.

Models consisting of combinations of difference or differential equations can be expressed in a similar manner. This shows how modelling constructs often used in DST can be expressed in TTL.

5.2 Expressing Executable Properties in TTL

As mentioned in Section 2, executable languages can be defined as sublanguages of TTL. An example of such a language, which was specifically designed for the simulation of dynamic processes in terms of both qualitative and quantitative concepts, is the LEADSTO language, cf. (Bosse *et al.*, 2005b). Below, it is shown how dynamic properties expressed in LEADSTO can be translated to TTL.

The LEADSTO language enables one to model direct temporal dependencies between two state properties in states at different points in time. A specification of dynamic properties in LEADSTO format has as advantages that it is executable and that it can often easily be depicted graphically. The format of LEADSTO is defined as follows. Let α and β be state properties of the form 'conjunction of atoms or negations of atoms', and e, f, g, h non-negative real numbers. In the LEADSTO language the notation $\alpha \twoheadrightarrow_{e, f, g, h} \beta$, means:

If state property α holds for a certain time interval with duration g, then after some delay (between e and f) state property β will hold for a certain time interval of length h.

In terms of TTL, the fact that the above statement holds for a trace γ can be expressed as follows:

$$\forall t1 [\forall t \ [t1-g \leq t < t1 \Rightarrow state(\gamma, t) \models \alpha] \Rightarrow \exists d \ [e \leq d \leq f \ \& \ \forall t' \ [t1+d \leq t' < t1+d+h \Rightarrow state(\gamma, t') \models \beta]$$

5.3 Expressing Standard Temporal Logics in TTL

As mentioned in Section 2, besides executable languages also languages often used for the verification of entailment relations can be defined as sublanguages of TTL. Examples of such languages are LTL and CTL, see, e.g., (Benthem, 1983; Goldblatt, 1992). In this section, it is briefly shown how dynamic properties expressed as formulae in standard temporal logics can be translated to TTL; in particular, this will be illustrated for the case of LTL. The general idea is that this can be done in a rather straightforward manner by replacing the temporal operators of LTL by quantifiers over time. For example, consider the following LTL formula:

$$G(observation\_result(itsraining) \rightarrow F(belief(itsraining)))$$

where the temporal operator G means 'for all later time points', and F 'for some later time point'. The first operator can be translated into a universal quantifier, whereas the second one can be translated into an existential quantifier. Using TTL, this formula then can be expressed, for example, as follows:

$$\forall t1 \ [\ state(\gamma, t1) \models observation\_result(itsraining) \Rightarrow \exists t2 \geq t1 \ state(\gamma, t2) \models belief(itsraining) \ ]$$

However, note that the translation is not bi-directional, i.e., it is not always possible to translate TTL expressions into LTL expressions. An example of a TTL expression that cannot be translated to LTL is the property 'Trust Monotonicity' expressed in Section 4.2. This property cannot be expressed in LTL since it involves the comparison of two different traces ($\gamma1$ and $\gamma2$ in this case). This shows that for example LTL can be considered a proper sublanguage of TTL, i.e., a sublanguage not equal to TTL. Similar observations can be made for other well-known temporal logics such as CTL.

To conclude, it was shown above that languages such as DST, LEADSTO and LTL can be seen as sublanguages of the specification language TTL. Note that this does *not* imply that all *operations* that can be done using these languages (e.g., solving differential equations specified in DST, or performing simulation based on LEADSTO) can be performed using TTL tools. Each language has its own tools to perform specific operations. The tools that were specifically implemented for TTL will be introduced in the next section.

6 Tools

The TTL language and its supporting software environment have been applied in a number of research projects in different domains. In general, the research goal in these projects was to analyse the behavioural dynamics of agents in different domains. In most of them the focus was on cognitive processes, such as human reasoning, the creation of consciousness, and design tasks. TTL was used to formalise dynamic properties of these processes at a high level of abstraction. Next, such dynamic properties (represented as TTL formulae) were automatically checked against simulated or empirical traces. This section presents the software environment that was built to support the process of specification and automated verification of dynamic properties. Basically, this software environment consists of two closely integrated tools: the Property Editor and the Checker Tool. To explain how these tools work, Section 6.1 describes more details of the TTL language from an implementation perspective. Next, Section 6.2 describes the actual operation of the tools. Finally, Section 6.3 discusses some implementation details of the Checker Tool.

6.1 Details of the TTL language

The previous sections introduced the TTL language in a somewhat informal way. However, the TTL software requires a strict representation. For instance, the implementation requires all variables in a TTL formula to be explicitly typed by specifying which sort they belong to. In this section, the TTL language is described in detail.

To enter TTL formulae in the correct format, the TTL Property Editor provides a graphical interface. The user fills in templates and builds up formulae by selecting building blocks from a menu. TTL specifications may also be supplied as plain text. The following definitions are used:

- A *TTL specification* consists of a number of user-defined property definitions and sort definitions. A property definition consists of a header (someprop(v1:s1, v2:s2), property name and formal arguments) and a body. The body is a TTL formula.
- A *TTL formula* is assembled from basic TTL formulae by conjunction, (Formula1 and Formula2), disjunction (Formula1 or Formula2), negation (not Formula), implication and quantification (forall ([v1:s1, v2:s2], Formula), exists ([v1:s1, v2:s2 < term2], Formula)).
- *Basic TTL formulae* are user-defined properties, Holds atoms, *predefined mathematical properties* (e.g. term1 = term2, term1 > term2) and built-in properties. The semantics of a user-defined property occurring in some TTL formula is one of substitution, not some kind of logic programming (recursion of properties is not allowed).
- *Holds atoms* are introduced in Section 4.2, e.g. state(trace1, t, output(ew)) $\models$ a1 $\wedge$ a2 .

- *Built-in properties* are complex properties encoded into the implementation language.
- TTL formula elements contain *terms* at various places: as restrictions on range variables, as actual parameter values in sub properties, within Holds atoms, and so on. Terms are "Prolog terms" (e.g., fn(t1,t2) , n1, t1 + t3, 1.3). Variables in terms are represented as X:sort1. Terms that are mathematical operations are evaluated, so the operands must be of an appropriate type. The functions begin(i:interval), end(i:interval), interval(t:time) and time(i:interval) introduced later are also terms that will be evaluated and substituted by their values.

For expressing more complex functions, the following building blocks are defined:

- case(Formula, Then, Else) where Formula is a TTL formula :
  f(case(Formula, Then, Else)) is equivalent to Formula and f(Then) or not Formula and f(Else).
- sum([v1:s1, v2:s2,..vn:sn], Term) where Term is a function of v1,..,vn: The sum of applying all tuples (v1,..vn) to Term.
- product(([v1:s1, v2:s2,..vn:sn], Term) where Term is a function of v1,..,vn: The product of applying all tuples (v1,..vn) to Term.

Furthermore, the language has a number of built-in *sorts* for integer, real and range of integers (sorts integer, real, between(i1:integer,i2:integer)). Sorts may be defined by enumerating their elements. There are predefined sorts for the set of all states (sort STATE) and the set of all loaded traces (sort TRACE, the temporal domain description set W introduced in Section 4.1).

TTL formulae usually contain variables referring to time, specifically to time for a state property. In case a dense time frame is used, this may cause problems for the verification process, because an infinite number of time points must be considered. To deal with this problem, in the TTL tools *finite variability* of state functions is assumed. This assumption states that between any two time points only a finite number of state changes occurs. Thus, when a property is checked against a set of traces, the software determines time-intervals during which all atoms occurring in the property are constant in all traces. A built-in sort interval enumerates these disjoint time intervals. Values of this sort are ordered. A number of primitives are introduced to translate between interval values and time values:

- begin(i:interval) refers to the first time point of interval i.
- end(i:interval) refers to the last time point of interval i.
- interval(t:time) refers to the interval in which time point t occurs.
- time(i:interval) refers to a time point that occurs in interval i.

For an example in which one of these primitives is used, see the following Holds atom:

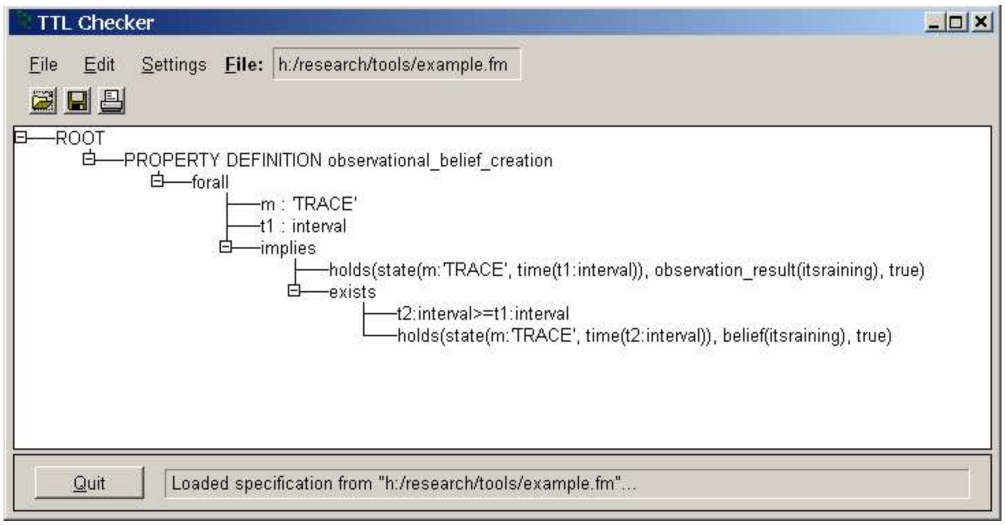
state(γ:TRACE, time(i:interval), internal) $\models$ a.

Moreover, libraries of predefined properties and functions are available, some generic, others for specific application domains.

6.2 Operation

As mentioned earlier, the TTL software environment comprises two closely integrated tools: the Property Editor and the Checker Tool. The Property Editor provides a user-friendly way of building and editing properties in the TTL language. It was designed in particular for less experienced users. By means of graphical manipulation and filling in of forms a TTL specification may be constructed (see Figure 3 for an impression). The Checker Tool can be used to check automatically whether a TTL formula holds for a set of traces. Operation of the tools involves three separate actions:

1. Loading, editing, and saving a set of TTL properties and user-defined sorts with the Property Editor (shown in Figure 3).
2. Activating the Trace Manager (not shown in Figure 3): loading and inspecting traces that will be checked and that will constitute the set of traces, the elements of sort TRACE (see section 6.1). Sources of traces can be both results of simulations such as output from the LEADSTO simulation software (see Bosse *et al.*, 2005b) and empirical traces.
3. Selecting a menu entry "Check Property" while the cursor points to a property. The property is compiled (see Section 6.3 for details) and checked, and the result is presented to the user.



h:/research/tools/example.fm

```
ROOT
  PROPERTY DEFINITION observational_belief_creation
    forall
      m : 'TRACE'
      t1 : interval
      implies
        holds(state(m:'TRACE', time(t1:interval)), observation_result(itsraining), true)
        exists
          t2:interval>=t1:interval
          holds(state(m:'TRACE', time(t2:interval)), belief(itsraining), true)
```

Loaded specification from "h:/research/tools/example.fm"...

Figure 3. The TTL Checker with Trace Loader

In addition to the above, the TTL Checker has facilities for systematically loading traces and checking properties without user interaction. The software runs on Windows, Solaris and Linux platforms.

6.3 Implementation Details of the Checker

This section describes the algorithm used by the Checker Tool in detail. Fist, a number of introductory remarks are made:

- The Checker Tool was built specifically for the process of checking TTL formulae against traces. Here, a trace consists of a finite number of state atoms, changing a finite number of times. This has the following consequences:
  - Using intervals instead of (continuous) time in TTL formulae will improve performance of the checking process (by simplifying quantification over time). Nevertheless, both options are possible.

    - Other quantification variables will often refer to arguments of state atoms. There are a finite number of such state atoms. Iterating over values occurring in the traces will often be faster than iterating over all possible values of some variable.
- Checking may involve iteration over many values. Therefore, efficient coding is important. Compiling the formula that needs to be checked into code in the implementation language will improve performance (compared to interpretation).
- Checking may involve frequent access to values of state atoms. For acceptable performance, it is important to assure efficient access to state atoms specific to the formula that is checked.

The implementation is in Prolog (SWI-Prolog, the graphical user interface uses XPCE). A query to check some TTL formula against all loaded traces is compiled into a Prolog clause, which will succeed if the formula holds. The compilation proceeds as follows:

1. Fast access to state atoms is ensured: all atoms occurring in state properties within the TTL formula are gathered. Then, the set of all traces is analysed to determine the time intervals where all those atoms are constant. An index is built for fast access to all those atom values.
2. The TTL formula is compiled into Prolog: the formula is translated by mapping conjunction, disjunction and negation onto Prolog equivalents and by transforming universal quantification into existential quantification. For every variable occurring in the property, information about whether it is bound is maintained. If the first occurrence of some variable in a conjunction is in a Holds atom, then this variable becomes bound by code that binds the variable to successive matching Holds atoms; in a following element of the conjunction, the value may be used in expressions and evaluations in other members of the conjunction. If a variable is not bound by such an occurrence, but should be bound (because it appears in some mathematical operation or comparison), the variable must be bound by generating binding code to bind the variable to successive elements of the variable sort. If the sort is infinite, an error message is generated.

The specific optimizations discussed above make it possible to check realistic dynamic properties with reasonable performance. For an impression of the performance: checking the simply property ‘Observational belief creation’ (see Section 4.2) against a single trace takes less than a second on a regular Personal Computer. Checking more complex properties may take longer. For example, a property involving 8 different time points (taken from Bosse *et al.*, 2005a) took about three minutes to check.

7 Conclusion

Within many domains, among which biological and cognitive areas, multiple interacting processes occur with dynamics that are hard to handle. Current approaches to analyse the dynamics of such processes are often based on differential equations. However, for a number of applications these approaches have serious limitations. For example, in Biology, approaches based on differential equations have problems in tackling more large-scale cellular systems. Moreover, within Cognitive Science, such approaches are not particularly suitable to model higher-level processes with mainly a qualitative character, such as reasoning and complex task performance.

To deal with these limitations, this paper presents the predicate logical Temporal Trace Language (TTL) for the formal specification and analysis of dynamic properties. Although the language has a logical foundation, it supports the specification of both qualitative and quantitative aspects, and subsumes specification languages based on differential equations.

To support the formal specification and analysis of dynamic properties, a special software environment has been developed for TTL. This environment features both a dedicated Property Editor for building and editing TTL properties and a Checking Tool that enables the formal verification of properties against a set of traces, for example obtained from experiments or simulation. Although this form of checking is not as exhaustive as model checking (which essentially means checking properties on the set of all theoretically possible traces), in return, this makes it possible to specify more expressive properties. Furthermore, more specialised languages can be defined as a sublanguage of TTL. First, for the purpose of simulation, the executable language LEADSTO has been developed (Bosse *et al.*, 2005b). Second, for the verification of entailment relations, standard temporal languages such as LTL and CTL (see, e.g., (Benthem, 1983; Goldblatt, 1992)) can be defined as sublanguages of TTL.

As mentioned above, TTL has a high expressive power. For example, the possibility of explicit reference to *time points* and *time durations* enables modelling of the dynamics of continuous real-time phenomena, such as sensory and neural activity patterns in relation to mental properties, cf. (Port and van Gelder, 1995). Also difference and differential equations can be expressed. These features go beyond the expressive power available in standard linear or branching time temporal logics.

Furthermore, the possibility to quantify over traces allows for specification of *more complex adaptive behaviours*. As within most temporal logics, reactiveness and pro-activeness properties can be specified. In addition, in our language also properties involving different types of adaptive behaviour can be expressed. An example of such a property is ‘exercise improves skill’, which is a relative property in the sense that it involves the comparison of two alternatives for the history. Another property of this type is trust monotony: ‘the better the experiences with something or someone, the higher the trust’.

The possibility to define restrictions to *local languages for parts* of a system or the world is also an important feature. For example, the distinction between internal, external and input and output languages is crucial, and is supported by the language TTL, which also entails the possibility to quantify over system parts; this allows for specification of system modification over time.

Finally, since state properties are used as first class citizens in the temporal trace language, it is possible to explicitly refer to them, and to quantify over them, enabling the specification of what are sometimes called *second-order properties*, which are used in part of the philosophical literature (e.g., Kim, 1998) to express functional roles related to mental properties or states.

The approach discussed in this paper follows the standard view on calculus (based on epsilon-delta definitions). Recently, in (Gamboa and Kaufmann, 2001) an alternative approach, following the non-standard view (based on infinitesimals) has been presented for the integration of calculus within a logical (and theorem proving) framework. It may be the case, as claimed by some researchers, that for computational purposes the non-standard view has advantages. This will be an issue for further research.

To conclude, the approach proved its value in a number of research projects in different domains. It has been used to analyse behavioural dynamics of agents in cognitive science (e.g., human reasoning, creation of consciousness, diagnosis of eating disorders), biology (e.g., cell decision processes, the dynamics of the heart), social science (e.g., organisation dynamics including organisational change, incident management), and artificial intelligence (e.g., design processes, ant colony behaviour). For more publications about these applications, the reader is referred to the authors' homepages.

References

1. Barringer, H., M. Fisher, D. Gabbay, R. Owens, & M. Reynolds (1996). *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press Ltd. and John Wiley & Sons.
2. Benthem, J.F.A.K., van (1983). The Logic of Time: A Model-theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse, Reidel, Dordrecht.
3. Bosse, T., Jonker, C.M., and Treur, J. (2005a). Representational Content and the Reciprocal Interplay of Agent and Environment. In: Leite, J., Omincini, A., Torroni, P., and Yolum, P. (eds.), *Proc. of the Second Int. Workshop on Declarative Agent Languages and Technologies, DALT'04*. Lecture Notes in Artificial Intelligence, vol. 3476. Springer Verlag, pp. 270-288.
4. Bosse, T., Jonker, C.M., Meij, L., van der, and Treur, J. (2005b). LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn (extended abstract). *Proc. of the 18th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, IEA/AIE 2005*. Lecture Notes in AI, Springer Verlag. In press.
5. Clarke, E.M., Grumberg, O., and Peled, D.A. (2000). *Model Checking*. MIT Press.
6. Gamboa, R., and Kaufmann, M. (2001). Nonstandard Analysis in ACL2. Journal of Automated Reasoning, vol. 27, pp. 323-351.
7. Geert, P. van (1995). Growth Dynamics in Development. In: (Port and van Gelder, 1995), pp. 101-120.
8. Goldblatt, R. (1992). *Logics of Time and Computation*, 2nd edition, CSLI Lecture Notes 7.
9. Kim, J. (1998). Mind in a Physical world: an Essay on the Mind-Body Problem and Mental Causation. MIT Press, Cambridge, Mass.
10. Manna, Z., and Pnueli, A. (1995). *Temporal Verification of Reactive Systems: Safety*. Springer Verlag.
11. Port, R.F., Gelder, T. van (eds.) (1995). *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge, Mass.
12. Stirling , C. (2001). *Modal and Temporal Properties of Processes*. Springer Verlag.
13. Townsend, J.T., and Busemeyer, J. (1995). Dynamic Representation in Decision Making. In: (Port and van Gelder, 1995), pp. 101-120.

C LEADSTO Specifications

C1 LEADSTO Specification of Behavior Networks

```
cwa(_)

⊟──leadsto
      ├──V: D : integer
      ├──A: data((d|D))
      ├──C: data((d|D))
      └──EFGH: efgh(0, 0, 0.1, 0.1)

CONSTANT phi=0.1

CONSTANT gamma=0.3

CONSTANT delta=0.5

CONSTANT threshold_decrease_factor=0.1

⊟──interval
      ├──R: range(0, 1)
      ⊟──F and
            ├──init
            └──current_time(1)

⊟──leadsto
      ├──V: C : integer
      ├──V: I : integer
      ⊟──A: and
      │     ├──init
      │     └──component_input_number((c|C), I)
      ⊟──C: and
      │     ├──alpha(0, (c|C), 0)
      │     └──active(0, (c|C), 0)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: V : real
      ├──A: alpha(T, (c|C), V)
      ├──C: alpha(T, (c|C), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: I : integer
      ├──A: active(T, (c|C), I)
      ├──C: active(T, (c|C), I)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ⊟──not
      │           └──time_change(T)
      ├──C: current_time(T)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: D1 : integer
      ├──V: D2 : integer
      ⊟──A: and
      │     ├──time_change(T)
      │     ├──number_of_goals(2)
      │     ├──goal((d|D1))
      │     ├──goal((d|D2))
      │     ├──D1 \= D2
      │     ⊟──not
      │           └──data((d|D1))
      ├──C: current_time(T+1)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: D1 : integer
      ⊟──A: and
      │     ├──time_change(T)
      │     ├──number_of_goals(1)
      │     ├──goal((d|D1))
      │     ⊟──not
      │           └──data((d|D1))
      ├──C: current_time(T+1)
      └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
leadsto
    V: C1 : integer
    V: C2 : integer
    V: D : integer
    A: and
        component_output((c|C1), (d|D))
        component_input((c|C2), (d|D))
        C1 \= C2
    C: successor_link((c|C1), (c|C2))
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: C1 : integer
    V: C2 : integer
    A: successor_link((c|C1), (c|C2))
    C: predecessor_link((c|C2), (c|C1))
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: C1 : integer
    V: C2 : integer
    V: D : integer
    A: and
        component_removes_data((c|C1), (d|D))
        component_input((c|C2), (d|D))
        C1 \= C2
    C: conflictor_link((c|C1), (c|C2))
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: T : integer
    A: and
        data((d|D))
        current_time(T)
    C: determine_M(T, (d|D), (c|1), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: T : integer
    V: C : integer
    A: and
        component_input((c|C), (d|D))
        not
            data((d|D))
        current_time(T)
    C: determine_M(T, (d|D), (c|1), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: T : integer
    A: and
        goal((d|D))
        current_time(T)
    C: determine_A(T, (d|D), (c|1), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: T : integer
    V: C : integer
    A: and
        component_input((c|C), (d|D))
        not
            data((d|D))
        current_time(T)
    C: determine_A(T, (d|D), (c|1), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: C : integer
    V: I : integer
    V: T : integer
    A: and
        current_time(T)
        component_input_number((c|C), I)
        I > 0
    C: determine_input_from_state(T, (d|1), (c|C), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)
```

```
leadsto
    V: C : integer
    V: I : integer
    V: T : integer
    A: and
        current_time(T)
        component_input_number((c|C), I)
        I = 0
    C: input_from_state(T, (c|C), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: C : integer
    V: I : integer
    V: T : integer
    A: and
        current_time(T)
        component_output_number((c|C), I)
        I > 0
    C: determine_input_from_goals(T, (d|1), (c|C), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: C : integer
    V: I : integer
    V: T : integer
    A: and
        current_time(T)
        component_output_number((c|C), I)
        I = 0
    C: input_from_goals(T, (c|C), 0)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: C : integer
    V: V : integer
    V: A : integer
    V: T : integer
    A: and
        current_time(T)
        determine_A(T, (d|D), (c|C), V)
        number_of_components(A)
        C <= A
        component_output((c|C), (d|D))
    C: determine_A(T, (d|D), (c|C+1), V+1)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: C : integer
    V: V : integer
    V: A : integer
    V: T : integer
    A: and
        current_time(T)
        determine_A(T, (d|D), (c|C), V)
        number_of_components(A)
        C <= A
        not
            component_output((c|C), (d|D))
    C: determine_A(T, (d|D), (c|C+1), V)
    EFGH: efgh(0, 0, 0.1, 0.1)

leadsto
    V: D : integer
    V: C : integer
    V: V : integer
    V: A : integer
    V: T : integer
    A: and
        current_time(T)
        determine_A(T, (d|D), (c|C), V)
        number_of_components(A)
        C = A+1
    C: 'A'(T, (d|D), V)
    EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
        ├──V: D : integer
        ├──V: C : integer
        ├──V: V : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_M(T, (d|D), (c|C), V)
        │       ├──number_of_components(A)
        │       ├──C <= A
        │       └──component_input((c|C), (d|D))
        ├──C: determine_M(T, (d|D), (c|C+1), V+1)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: D : integer
        ├──V: C : integer
        ├──V: V : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_M(T, (d|D), (c|C), V)
        │       ├──number_of_components(A)
        │       ├──C <= A
        │       ⊟──not
        │               └──component_input((c|C), (d|D))
        ├──C: determine_M(T, (d|D), (c|C+1), V)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: D : integer
        ├──V: C : integer
        ├──V: V : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_M(T, (d|D), (c|C), V)
        │       ├──number_of_components(A)
        │       └──C = A+1
        ├──C: 'M'(T, (d|D), V)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: D : integer
        ├──V: N : integer
        ├──V: V : real
        ├──V: C : integer
        ├──V: O : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_input_from_goals(T, (d|D), (c|C), V)
        │       ├──amount_of_data(N)
        │       ├──D <= N
        │       ├──goal((d|D))
        │       ├──component_output((c|C), (d|D))
        │       ├──'A'(T, (d|D), A)
        │       └──component_output_number((c|C), O)
        ├──C: determine_input_from_goals(T, (d|D+1), (c|C), V+gamma* (1/A)* (1/O))
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: D : integer
        ├──V: N : integer
        ├──V: V : real
        ├──V: C : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_input_from_goals(T, (d|D), (c|C), V)
        │       ├──amount_of_data(N)
        │       ├──D <= N
        │       ├──goal((d|D))
        │       ⊟──not
        │               └──component_output((c|C), (d|D))
        ├──C: determine_input_from_goals(T, (d|D+1), (c|C), V)
        └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
      ├──V: D : integer
      ├──V: V : real
      ├──V: C : integer
      ├──V: A : integer
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_input_from_goals(T, (d|D), (c|C), V)
      │     ├──amount_of_data(A)
      │     ├──D <= A
      │     ⊟──not
      │           └──goal((d|D))
      ├──C: determine_input_from_goals(T, (d|D+1), (c|C), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: D : integer
      ├──V: V : real
      ├──V: C : integer
      ├──V: A : integer
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_input_from_goals(T, (d|D), (c|C), V)
      │     ├──amount_of_data(A)
      │     └──D = A+1
      ├──C: input_from_goals(T, (c|C), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: D : integer
      ├──V: M : integer
      ├──V: V : real
      ├──V: C : integer
      ├──V: I : integer
      ├──V: A : integer
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_input_from_state(T, (d|D), (c|C), V)
      │     ├──amount_of_data(A)
      │     ├──D <= A
      │     ├──data((d|D))
      │     ├──component_input((c|C), (d|D))
      │     ├──'M'(T, (d|D), M)
      │     └──component_input_number((c|C), I)
      ├──C: determine_input_from_state(T, (d|D+1), (c|C), V+phi* (1/M)* (1/I))
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: D : integer
      ├──V: V : real
      ├──V: C : integer
      ├──V: A : integer
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_input_from_state(T, (d|D), (c|C), V)
      │     ├──amount_of_data(A)
      │     ├──D <= A
      │     ├──data((d|D))
      │     ⊟──not
      │           └──component_input((c|C), (d|D))
      ├──C: determine_input_from_state(T, (d|D+1), (c|C), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: D : integer
      ├──V: V : real
      ├──V: C : integer
      ├──V: A : integer
      ├──V: T : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_input_from_state(T, (d|D), (c|C), V)
      │     ├──amount_of_data(A)
      │     ├──D <= A
      │     ⊟──not
      │           └──data((d|D))
      ├──C: determine_input_from_state(T, (d|D+1), (c|C), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
        ├──V: D : integer
        ├──V: V : real
        ├──V: C : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_input_from_state(T, (d|D), (c|C), V)
        │       ├──amount_of_data(A)
        │       └──D = A+1
        ├──C: input_from_state(T, (c|C), V)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: C : integer
        ├──V: I : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       └──component_input_number((c|C), I)
        ├──C: determine_executable(T, (c|C), (d|1), 0)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: C : integer
        ├──V: D : integer
        ├──V: I : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_executable(T, (c|C), (d|D), I)
        │       ├──amount_of_data(A)
        │       ├──D <= A
        │       ├──data((d|D))
        │       └──component_input((c|C), (d|D))
        ├──C: determine_executable(T, (c|C), (d|D+1), I+1)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: C : integer
        ├──V: D : integer
        ├──V: I : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_executable(T, (c|C), (d|D), I)
        │       ├──amount_of_data(A)
        │       ├──D <= A
        │       ⊟──not
        │               └──data((d|D))
        ├──C: determine_executable(T, (c|C), (d|D+1), I)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: C : integer
        ├──V: D : integer
        ├──V: I : integer
        ├──V: A : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_executable(T, (c|C), (d|D), I)
        │       ├──amount_of_data(A)
        │       ├──D <= A
        │       ⊟──not
        │               └──component_input((c|C), (d|D))
        ├──C: determine_executable(T, (c|C), (d|D+1), I)
        └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
        ├──V: C : integer
        ├──V: D : integer
        ├──V: I : integer
        ├──V: A : integer
        ├──V: N : integer
        ├──V: T : integer
        ⊟──A: and
        │       ├──current_time(T)
        │       ├──determine_executable(T, (c|C), (d|D), I)
        │       ├──amount_of_data(A)
        │       ├──D = A+1
        │       ├──component_input_number((c|C), N)
        │       └──N = I
        ├──C: executable(T, (c|C))
        └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
     ├──V: C : integer
     ├──V: D : integer
     ├──V: I : integer
     ├──V: A : integer
     ├──V: N : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_exacutable(T, (c|C), (d|D), I)
     │    ├──amount_of_data(A)
     │    ├──D = A+1
     │    ├──component_input_number((c|C), N)
     │    └──I < N
     ⊟──C: not
     │    └──executable(T, (c|C))
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: A : integer
     ├──V: AL : real
     ├──V: T : integer
     ├──V: O : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_bw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    ⊟──not
     │    │    └──data((d|D))
     │    ├──component_input((c|C1), (d|D))
     │    ├──component_output((c|C2), (d|D))
     │    ├──'A'(T, (d|D), A)
     │    ├──alpha(T-1, (c|C1), AL)
     │    └──component_output_number((c|C2), O)
     ├──C: determine_spreads_bw(T, (d|D+1), (c|C1), (c|C2), V+AL* (1/A* (1/O)))
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_bw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    ⊟──not
     │    │    └──data((d|D))
     │    ⊟──not
     │         └──component_input((c|C1), (d|D))
     ├──C: determine_spreads_bw(T, (d|D+1), (c|C1), (c|C2), V)
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_bw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    ⊟──not
     │    │    └──data((d|D))
     │    ⊟──not
     │         └──component_output((c|C2), (d|D))
     ├──C: determine_spreads_bw(T, (d|D+1), (c|C1), (c|C2), V)
     └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_bw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    └──data((d|D))
     ├──C: determine_spreads_bw(T, (d|D+1), (c|C1), (c|C2), V)
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_bw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    └──D = N+1
     ├──C: spreads_backwards(T, (c|C1), (c|C2), V)
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: D1 : integer
     ├──V: D2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──component_input((c|C1), (d|D1))
     │    └──component_input((c|C2), (d|D2))
     ├──C: determine_spreads_bw(T, (d|1), (c|C1), (c|C2), 0)
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: M : integer
     ├──V: AL : real
     ├──V: T : integer
     ├──V: I : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_fw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    ⊟──not
     │    │    └──data((d|D))
     │    ├──component_output((c|C1), (d|D))
     │    ├──component_input((c|C2), (d|D))
     │    ├──'M'(T, (d|D), M)
     │    ├──alpha(T-1, (c|C1), AL)
     │    └──component_input_number((c|C2), I)
     ├──C: determine_spreads_fw(T, (d|D+1), (c|C1), (c|C2), V+AL* (phi/gamma* (1/M)* (1/I)))
     └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
     ├──V: D : integer
     ├──V: N : integer
     ├──V: V : real
     ├──V: C1 : integer
     ├──V: C2 : integer
     ├──V: T : integer
     ⊟──A: and
     │    ├──current_time(T)
     │    ├──determine_spreads_fw(T, (d|D), (c|C1), (c|C2), V)
     │    ├──amount_of_data(N)
     │    ├──D <= N
     │    ⊟──not
     │    │    └──data((d|D))
     │    ⊟──not
     │         └──component_output((c|C1), (d|D))
     ├──C: determine_spreads_fw(T, (d|D+1), (c|C1), (c|C2), V)
     └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
    ├──V: D : integer
    ├──V: N : integer
    ├──V: V : real
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: T : integer
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──determine_spreads_fw(T, (d|D), (c|C1), (c|C2), V)
    │   ├──amount_of_data(N)
    │   ├──D <= N
    │   ⊟──not
    │   │   └──data((d|D))
    │   ⊟──not
    │       └──component_input((c|C2), (d|D))
    ├──C: determine_spreads_fw(T, (d|D+1), (c|C1), (c|C2), V)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
⊟──leadsto
    ├──V: D : integer
    ├──V: N : integer
    ├──V: V : real
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: T : integer
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──determine_spreads_fw(T, (d|D), (c|C1), (c|C2), V)
    │   ├──amount_of_data(N)
    │   ├──D <= N
    │   └──data((d|D))
    ├──C: determine_spreads_fw(T, (d|D+1), (c|C1), (c|C2), V)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
⊟──leadsto
    ├──V: D : integer
    ├──V: N : integer
    ├──V: V : real
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: T : integer
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──determine_spreads_fw(T, (d|D), (c|C1), (c|C2), V)
    │   ├──amount_of_data(N)
    │   └──D = N+1
    ├──C: spreads_forwards(T, (c|C1), (c|C2), V)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
⊟──leadsto
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: D1 : integer
    ├──V: D2 : integer
    ├──V: T : integer
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──component_input((c|C1), (d|D1))
    │   └──component_input((c|C2), (d|D2))
    ├──C: determine_spreads_fw(T, (d|1), (c|C1), (c|C2), 0)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
⊟──leadsto
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: T : integer
    ├──V: V : real
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──spreads_forwards(T, (c|C1), (c|C2), V)
    │   └──executable(T, (c|C1))
    ├──C: spreads_fw((c|C1), (c|C2), T, V)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
⊟──leadsto
    ├──V: C1 : integer
    ├──V: C2 : integer
    ├──V: T : integer
    ├──V: V : real
    ⊟──A: and
    │   ├──current_time(T)
    │   ├──spreads_forwards(T, (c|C1), (c|C2), V)
    │   ⊟──not
    │       └──executable(T, (c|C1))
    ├──C: spreads_fw((c|C1), (c|C2), T, 0)
    └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: V : real
       ├──V: T : integer
       ⊟──A: and
       │      ├──current_time(T)
       │      ├──spreads_backwards(T, (c|C1), (c|C2), V)
       │      ⊟──not
       │             └──executable(T, (c|C1))
       ├──C: spreads_bw((c|C1), (c|C2), T, V)
       └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: T : integer
       ├──V: V : real
       ⊟──A: and
       │      ├──current_time(T)
       │      ├──spreads_backwards(T, (c|C1), (c|C2), V)
       │      └──executable(T, (c|C1))
       ├──C: spreads_bw((c|C1), (c|C2), T, 0)
       └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: T : integer
       ├──V: S : real
       ├──V: N : integer
       ├──V: V : real
       ⊟──A: and
       │      ├──sum_spreads_fw((c|C1), (c|C2), T, S)
       │      ├──number_of_components(N)
       │      ├──C1 <= N
       │      └──spreads_fw((c|C1), (c|C2), T, V)
       ├──C: sum_spreads_fw((c|C1+1), (c|C2), T, S+V)
       └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: T : integer
       ├──V: S : real
       ├──V: N : integer
       ⊟──A: and
       │      ├──sum_spreads_fw((c|C1), (c|C2), T, S)
       │      ├──number_of_components(N)
       │      └──C1 = N+1
       ├──C: sum_spreads_fw((c|C2), T, S)
       └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: T : integer
       ├──V: S : real
       ├──V: N : integer
       ├──V: V : real
       ⊟──A: and
       │      ├──sum_spreads_bw((c|C1), (c|C2), T, S)
       │      ├──number_of_components(N)
       │      ├──C1 <= N
       │      └──spreads_bw((c|C1), (c|C2), T, V)
       ├──C: sum_spreads_bw((c|C1+1), (c|C2), T, S+V)
       └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
       ├──V: C1 : integer
       ├──V: C2 : integer
       ├──V: T : integer
       ├──V: S : real
       ├──V: N : integer
       ⊟──A: and
       │      ├──sum_spreads_bw((c|C1), (c|C2), T, S)
       │      ├──number_of_components(N)
       │      ├──C1 <= N
       │      ⊟──not
       │             └──predecessor_link((c|C1), (c|C2))
       ├──C: sum_spreads_bw((c|C1+1), (c|C2), T, S)
       └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
      ├──V: C1 : integer
      ├──V: C2 : integer
      ├──V: T : integer
      ├──V: S : real
      ├──V: N : integer
      ⊟──A: and
      │     ├──sum_spreads_bw((c|C1), (c|C2), T, S)
      │     ├──number_of_components(N)
      │     └──C1 = N+1
      ├──C: sum_spreads_bw((c|C2), T, S)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: I : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     └──component_input_number((c|C), I)
      ⊟──C: and
      │     ├──sum_spreads_bw((c|1), (c|C), T, 0)
      │     └──sum_spreads_fw((c|1), (c|C), T, 0)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: A : real
      ├──V: AC : real
      ├──V: V1 : real
      ├──V: V2 : real
      ├──V: V3 : real
      ├──V: V4 : real
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──alpha(T-1, (c|C), A)
      │     ├──active(T-1, (c|C), AC)
      │     ├──input_from_state(T, (c|C), V1)
      │     ├──input_from_goals(T, (c|C), V2)
      │     ├──sum_spreads_bw((c|C), T, V3)
      │     └──sum_spreads_fw((c|C), T, V4)
      ├──C: decay(T, (c|C), A* (1-AC)+V1+V2+V3+V4)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: V : real
      ⊟──A: and
      │     ├──current_time(T)
      │     └──decay(T, (c|C), V)
      ├──C: sum_decay(T, (c|1), 0)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: V : real
      ├──V: N : integer
      ├──V: D : real
      ⊟──A: and
      │     ├──sum_decay(T, (c|C), V)
      │     ├──current_time(T)
      │     ├──number_of_components(N)
      │     ├──C <= N
      │     └──decay(T, (c|C), D)
      ├──C: sum_decay(T, (c|C+1), V+D)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: V : real
      ├──V: N : integer
      ⊟──A: and
      │     ├──sum_decay(T, (c|C), V)
      │     ├──current_time(T)
      │     ├──number_of_components(N)
      │     └──C = N+1
      ├──C: sum_decay(T, V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: S : real
      ├──V: D : real
      ⊟──A: and
      │     ├──sum_decay(T, S)
      │     └──decay(T, (c|C), D)
      ⊟──C: and
      │     ├──alpha(T, (c|C), D/S)
      │     └──determine_highest_alpha(T, (c|1), (c|0), 0)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: B : integer
      ├──V: V : real
      ├──V: N : integer
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_highest_alpha(T, (c|C), (c|B), V)
      │     ├──number_of_components(N)
      │     ├──C <= N
      │     ⊟──not
      │           └──executable(T, (c|C))
      ├──C: determine_highest_alpha(T, (c|C+1), (c|B), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: B : integer
      ├──V: V : real
      ├──V: N : integer
      ├──V: A : real
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_highest_alpha(T, (c|C), (c|B), V)
      │     ├──number_of_components(N)
      │     ├──C <= N
      │     ├──alpha(T, (c|C), A)
      │     ├──executable(T, (c|C))
      │     └──A < V
      ├──C: determine_highest_alpha(T, (c|C+1), (c|B), V)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: B : integer
      ├──V: V : real
      ├──V: N : integer
      ├──V: A : real
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_highest_alpha(T, (c|C), (c|B), V)
      │     ├──number_of_components(N)
      │     ├──C <= N
      │     ├──executable(T, (c|C))
      │     ├──alpha(T, (c|C), A)
      │     └──A = V
      ⊟──C: and
      │     ├──yes
      │     ⊟──PXOR
      │           ⊟──Prob 0.5
      │           │     └──determine_highest_alpha(T, (c|C+1), (c|C), A)
      │           ⊟──OTHERWISE
      │                 └──determine_highest_alpha(T, (c|C+1), (c|B), A)
      └──EFGH: efgh(0, 0, 0.1, 0.1)

⊟──leadsto
      ├──V: T : integer
      ├──V: C : integer
      ├──V: B : integer
      ├──V: V : real
      ├──V: N : integer
      ├──V: A : real
      ⊟──A: and
      │     ├──current_time(T)
      │     ├──determine_highest_alpha(T, (c|C), (c|B), V)
      │     ├──number_of_components(N)
      │     ├──C <= N
      │     ├──executable(T, (c|C))
      │     ├──alpha(T, (c|C), A)
      │     └──A > V
      ├──C: determine_highest_alpha(T, (c|C+1), (c|C), A)
      └──EFGH: efgh(0, 0, 0.1, 0.1)
```

```
⊟─leadsto
    ├─V: T : integer
    ├─V: C : integer
    ├─V: B : integer
    ├─V: V : real
    ├─V: N : integer
    ⊟─A: and
    │   ├─current_time(T)
    │   ├─determine_highest_alpha(T, (c|C), (c|B), V)
    │   ├─number_of_components(N)
    │   └─C = N+1
    ├─C: highest_alpha(T, (c|B), V)
    └─EFGH: efgh(0, 0, 0.1, 0.1)

⊟─leadsto
    ├─V: T : integer
    ├─V: B : integer
    ├─V: V : real
    ⊟─A: and
    │   ├─current_time(T)
    │   ├─highest_alpha(T, (c|B), V)
    │   └─B = 0
    ├─C: selection_failed_at_time(T)
    └─EFGH: efgh(0, 0, 0.1, 0.1)

⊟─leadsto
    ├─V: T : integer
    ├─V: B : integer
    ├─V: V : real
    ⊟─A: and
    │   ├─current_time(T)
    │   ├─highest_alpha(T, (c|B), V)
    │   ⊟─not
    │   │   └─highest_alpha(T, (c|C), V)
    │   └─B \= 0
    ⊟─C: and
    │   ├─active(T, (c|B), 1)
    │   └─activated((c|B))
    └─EFGH: efgh(0, 0, 0.1, 0.1)

⊟─leadsto
    ├─V: C : integer
    ├─A: activated((c|C))
    ├─C: deactivated((c|C))
    └─EFGH: efgh(1, 1, 0.1, 0.1)

⊟─leadsto
    ├─V: T : integer
    ├─V: B : integer
    ├─V: V : real
    ├─V: C : integer
    ├─V: V2 : real
    ⊟─A: and
    │   ├─current_time(T)
    │   ├─highest_alpha(T, (c|B), V)
    │   ├─B \= 0
    │   ├─alpha(T, (c|C), V2)
    │   └─B \= C
    ├─C: active(T, (c|C), 0)
    └─EFGH: efgh(0, 0, 0.1, 0.1)

OTHER :  display(_, sort_atoms_time_global)

OTHER :  display(_, show_atoms(data((_|_))))

OTHER :  display(_, show_atoms(active(_, _, _)))

OTHER :  display(_, show_atoms(activated(_|_)))

OTHER :  display(_, show_atoms(deactivated(_|_)))

OTHER :  display(_, show_atoms(decay(_, _, _)))

OTHER :  display(_, show_atoms(alpha(_, _, _)))

OTHER :  display(_, show_atoms(executable(_, _)))

OTHER :  display(_, show_atoms(sum_spread_bw(_, _, _)))

OTHER :  display(_, show_atoms(sum_spread_bw(_, _, _)))

OTHER :  display(_, show_atoms(sum_spread_fw(_, _, _)))

OTHER :  display(_, show_atoms(current_time(_)))
```

C2 LEADSTO Specification of Pandemonium

end_time(40)

CONSTANT end=40

```
⊟──leadsto
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ├──A: initial_data((d|D1)xor (d|D2))
       ⊟──C: and
       │      ├──new_data
       │      ⊟──PXOR
       │             ⊟──Prob 0.5
       │             │      └──data((d|D1))
       │             ⊟──OTHERWISE
       │                    └──data((d|D2))
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ├──A: component_output((c|C), (d|D1)xor (d|D2))
       ⊟──C: and
       │      ├──dummy
       │      ⊟──PXOR
       │             ⊟──Prob 0.5
       │             │      └──component_output((c|C), (d|D1))
       │             ⊟──OTHERWISE
       │                    └──component_output((c|C), (d|D2))
       └──EFGH: efgh(0, 0, 1, end)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ⊟──not
       │      │      └──data((d|D))
       │      ├──component_input_number((c|C), 1)
       │      └──component_input((c|C), (d|D))
       ├──C: component_input_present((c|C), 0)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D))
       │      ├──component_input_number((c|C), 1)
       │      └──component_input((c|C), (d|D))
       ├──C: component_input_present((c|C), 1)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ⊟──not
       │      │      └──data((d|D1))
       │      ⊟──not
       │      │      └──data((d|D2))
       │      ├──component_input_number((c|C), 2)
       │      ├──component_input((c|C), (d|D1))
       │      ├──component_input((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_input_present((c|C), 0)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D1))
       │      ⊟──not
       │      │      └──data((d|D2))
       │      ├──component_input_number((c|C), 2)
       │      ├──component_input((c|C), (d|D1))
       │      ├──component_input((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_input_present((c|C), 1)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D1))
       │      ├──data((d|D2))
       │      ├──component_input_number((c|C), 2)
       │      ├──component_input((c|C), (d|D1))
       │      ├──component_input((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_input_present((c|C), 2)
       └──EFGH: efgh(0, 0, 1, 1)
```

```
⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ⊟──not
       │      │      └──data((d|D))
       │      ├──component_output_number((c|C), 1)
       │      └──component_output((c|C), (d|D))
       ├──C: component_output_present((c|C), 0)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D))
       │      ├──component_output_number((c|C), 1)
       │      └──component_output((c|C), (d|D))
       ├──C: component_output_present((c|C), 1)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ⊟──not
       │      │      └──data((d|D1))
       │      ⊟──not
       │      │      └──data((d|D2))
       │      ├──component_output_number((c|C), 2)
       │      ├──component_output((c|C), (d|D1))
       │      ├──component_output((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_output_present((c|C), 0)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D1))
       │      ⊟──not
       │      │      └──data((d|D2))
       │      ├──component_output_number((c|C), 2)
       │      ├──component_output((c|C), (d|D1))
       │      ├──component_output((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_output_present((c|C), 1)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: D1 : between(1, data)
       ├──V: D2 : between(1, data)
       ⊟──A: and
       │      ├──new_data
       │      ├──data((d|D1))
       │      ├──data((d|D2))
       │      ├──component_output_number((c|C), 2)
       │      ├──component_output((c|C), (d|D1))
       │      ├──component_output((c|C), (d|D2))
       │      └──D1 \= D2
       ├──C: component_output_present((c|C), 2)
       └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: i : integer
       ⊟──A: and
       │      ├──component_input_number((c|C), i)
       │      └──component_input_present((c|C), i)
       ├──C: component_allowed((c|C))
       └──EFGH: efgh(3, 3, 1, 1)

⊟──leadsto
       ├──V: C : between(1, components)
       ├──V: i1 : integer
       ├──V: i2 : integer
       ├──V: o1 : integer
       ├──V: o2 : integer
       ⊟──A: and
       │      ├──component_input_number((c|C), i1)
       │      ├──component_input_present((c|C), i2)
       │      ├──component_output_number((c|C), o1)
       │      ├──component_output_present((c|C), o2)
       │      ⊟──not
       │             └──termination
       ├──C: shout((c|C), (i2/i1)^1.4* (1-o2/o1)^1.3* (i1/max_input)^1.1* (o1/max_output)^1.2)
       └──EFGH: efgh(0, 0, 1, 1)
```

```
leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: x1 : real
    V: x2 : real
    A: and
        shout((c|C1), x1)
        shout((c|C2), x2)
        x1 >= x2
    C: weak_better_than((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: x1 : real
    V: x2 : real
    A: and
        shout((c|C1), x1)
        shout((c|C2), x2)
        x1 > x2
    C: strong_better_than((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: x1 : real
    A: shout((c|C1), x1)
    C: strong_better_than((c|C1), (c|C1))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: X : between(1, components)
    A: forall
        V: Y : between(1, components)
        weak_better_than((c|X), (c|Y))
    C: possible_component((c|X))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    A: and
        possible_component((c|C1))
        possible_component((c|C2))
        C1 > C2
    C: and
        dummy
        PXOR
            Prob 0.33
                active_component((c|C1))
            Prob 0.33
                active_component((c|C2))
            OTHERWISE
                and
                    active_component((c|C1))
                    active_component((c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: X : between(1, components)
    A: forall
        V: Y : between(1, components)
        strong_better_than((c|X), (c|Y))
    C: active_component((c|X))
    EFGH: efgh(1, 1, 1, 1)

leadsto
    V: C : between(1, components)
    V: D : between(1, data)
    A: and
        active_component((c|C))
        component_allowed((c|C))
        component_output((c|C), (d|D))
    C: and
        data((d|D))
        new_data
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: D : between(1, data)
    A: data((d|D))
    C: data((d|D))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: D : between(1, data)
    A: and
        data((d|D))
        goal_data((d|D))
    C: termination
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: D1 : between(1, data)
    V: D2 : between(1, data)
    A: and
        data((d|D1))
        data((d|D2))
        goal_data(and((d|D1), (d|D2)))
    C: termination
    EFGH: efgh(0, 0, 1, 1)
```

C3 LEADSTO Specification of Voting

```
end_time(end)

CONSTANT end=70

⊟──leadsto
        ├──V: D1 : between(1, data)
        ├──V: D2 : between(1, data)
        ├──A: initial_data((d|D1)xor (d|D2))
        ⊟──C: and
        │       ├──new_data
        │       ⊟──PXOR
        │               ⊟──Prob 0.5
        │               │       └──data((d|D1))
        │               ⊟──OTHERWISE
        │                       └──data((d|D2))
        └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D1 : between(1, data)
        ├──V: D2 : between(1, data)
        ├──A: component_output((c|C), (d|D1)xor (d|D2))
        ⊟──C: and
        │       ├──dummy
        │       ⊟──PXOR
        │               ⊟──Prob 0.5
        │               │       └──component_output((c|C), (d|D1))
        │               ⊟──OTHERWISE
        │                       └──component_output((c|C), (d|D2))
        └──EFGH: efgh(0, 0, 1, end)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D : between(1, data)
        ⊟──A: and
        │       ├──new_data
        │       ⊟──not
        │       │       └──data((d|D))
        │       ├──component_input_number((c|C), 1)
        │       └──component_input((c|C), (d|D))
        ├──C: component_input_present((c|C), 0)
        └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D : between(1, data)
        ⊟──A: and
        │       ├──new_data
        │       ├──data((d|D))
        │       ├──component_input_number((c|C), 1)
        │       └──component_input((c|C), (d|D))
        ├──C: component_input_present((c|C), 1)
        └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D1 : between(1, data)
        ├──V: D2 : between(1, data)
        ⊟──A: and
        │       ├──new_data
        │       ⊟──not
        │       │       └──data((d|D1))
        │       ⊟──not
        │       │       └──data((d|D2))
        │       ├──component_input_number((c|C), 2)
        │       ├──component_input((c|C), (d|D1))
        │       ├──component_input((c|C), (d|D2))
        │       └──D1 \= D2
        ├──C: component_input_present((c|C), 0)
        └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D1 : between(1, data)
        ├──V: D2 : between(1, data)
        ⊟──A: and
        │       ├──new_data
        │       ├──data((d|D1))
        │       ⊟──not
        │       │       └──data((d|D2))
        │       ├──component_input_number((c|C), 2)
        │       ├──component_input((c|C), (d|D1))
        │       ├──component_input((c|C), (d|D2))
        │       └──D1 \= D2
        ├──C: component_input_present((c|C), 1)
        └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
        ├──V: C : between(1, components)
        ├──V: D1 : between(1, data)
        ├──V: D2 : between(1, data)
        ⊟──A: and
        │       ├──new_data
        │       ├──data((d|D1))
        │       ├──data((d|D2))
        │       ├──component_input_number((c|C), 2)
        │       ├──component_input((c|C), (d|D1))
        │       ├──component_input((c|C), (d|D2))
        │       └──D1 \= D2
        ├──C: component_input_present((c|C), 2)
        └──EFGH: efgh(0, 0, 1, 1)
```

```
⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: D : between(1, data)
     ⊟──A: and
     │    ├──new_data
     │    ⊟──not
     │    │    └──data((d|D))
     │    ├──component_output_number((c|C), 1)
     │    └──component_output((c|C), (d|D))
     ├──C: component_output_present((c|C), 0)
     └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: D : between(1, data)
     ⊟──A: and
     │    ├──new_data
     │    ├──data((d|D))
     │    ├──component_output_number((c|C), 1)
     │    └──component_output((c|C), (d|D))
     ├──C: component_output_present((c|C), 1)
     └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: D1 : between(1, data)
     ├──V: D2 : between(1, data)
     ⊟──A: and
     │    ├──new_data
     │    ⊟──not
     │    │    └──data((d|D1))
     │    ⊟──not
     │    │    └──data((d|D2))
     │    ├──component_output_number((c|C), 2)
     │    ├──component_output((c|C), (d|D1))
     │    ├──component_output((c|C), (d|D2))
     │    └──D1 \= D2
     ├──C: component_output_present((c|C), 0)
     └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: D1 : between(1, data)
     ├──V: D2 : between(1, data)
     ⊟──A: and
     │    ├──new_data
     │    ├──data((d|D1))
     │    ⊟──not
     │    │    └──data((d|D2))
     │    ├──component_output_number((c|C), 2)
     │    ├──component_output((c|C), (d|D1))
     │    ├──component_output((c|C), (d|D2))
     │    └──D1 \= D2
     ├──C: component_output_present((c|C), 1)
     └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: D1 : between(1, data)
     ├──V: D2 : between(1, data)
     ⊟──A: and
     │    ├──new_data
     │    ├──data((d|D1))
     │    ├──data((d|D2))
     │    ├──component_output_number((c|C), 2)
     │    ├──component_output((c|C), (d|D1))
     │    ├──component_output((c|C), (d|D2))
     │    └──D1 \= D2
     ├──C: component_output_present((c|C), 2)
     └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: i : integer
     ⊟──A: and
     │    ├──component_input_number((c|C), i)
     │    └──component_input_present((c|C), i)
     ├──C: component_allowed((c|C))
     └──EFGH: efgh(components+4, components+4, 1, 1)

⊟──leadsto
     ├──V: C : between(1, components)
     ├──V: i1 : integer
     ├──V: i2 : integer
     ├──V: o1 : integer
     ├──V: o2 : integer
     ⊟──A: and
     │    ├──component_input_number((c|C), i1)
     │    ├──component_input_present((c|C), i2)
     │    ├──component_output_number((c|C), o1)
     │    ├──component_output_present((c|C), o2)
     │    ├──i1 = i2
     │    ├──o1 > o2
     │    ⊟──not
     │         └──termination
     ├──C: vote_for((c|C), (c|C))
     └──EFGH: efgh(0, 0, 1, 1)
```

```
leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: D : between(1, data)
    V: i1 : integer
    V: i2 : integer
    V: o1 : integer
    V: o2 : integer
    A: and
        component_input_number((c|C1), i1)
        component_input_present((c|C1), i2)
        component_output_number((c|C1), o1)
        component_output_present((c|C1), o2)
        i1-i2 = 1
        not
            data((d|D))
        component_input((c|C1), (d|D))
        component_output((c|C2), (d|D))
        not
            termination
    C: vote_for((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)
leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: C3 : between(1, components)
    V: D1 : between(1, data)
    V: D2 : between(1, data)
    V: i1 : integer
    V: i2 : integer
    V: o1 : integer
    V: o2 : integer
    A: and
        component_input_number((c|C1), i1)
        component_input_present((c|C1), i2)
        component_output_number((c|C1), o1)
        component_output_present((c|C1), o2)
        i1-i2 = 2
        not
            data((d|D1))
        component_input((c|C1), (d|D1))
        component_output((c|C2), (d|D1))
        not
            data((d|D2))
        component_input((c|C1), (d|D2))
        component_output((c|C3), (d|D2))
        D1 > D2
        not
            termination
    C: and
        dummy
        PXOR
            Prob 0.5
                vote_for((c|C1), (c|C2))
            OTHERWISE
                vote_for((c|C1), (c|C3))
    EFGH: efgh(0, 0, 1, 1)
leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: D : between(1, data)
    V: i1 : integer
    V: i2 : integer
    V: o1 : integer
    V: o2 : integer
    A: and
        component_input_number((c|C1), i1)
        component_input_present((c|C1), i2)
        component_output_number((c|C1), o1)
        component_output_present((c|C1), o2)
        i1 = i2
        o1 = o2
        o1 = 1
        component_output((c|C1), (d|D))
        component_input((c|C2), (d|D))
        not
            termination
    C: vote_for((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)
```

```
⊟──leadsto
      ├──V: C1 : between(1, components)
      ├──V: C2 : between(1, components)
      ├──V: C3 : between(1, components)
      ├──V: D1 : between(1, data)
      ├──V: D2 : between(1, data)
      ├──V: i1 : integer
      ├──V: i2 : integer
      ├──V: o1 : integer
      ├──V: o2 : integer
      ⊟──A: and
      │     ├──component_input_number((c|C1), i1)
      │     ├──component_input_present((c|C1), i2)
      │     ├──component_output_number((c|C1), o1)
      │     ├──component_output_present((c|C1), o2)
      │     ├──i1 = i2
      │     ├──o1 = o2
      │     ├──o1 = 2
      │     ├──component_output((c|C1), (d|D1))
      │     ├──component_input((c|C2), (d|D1))
      │     ├──component_output((c|C1), (d|D2))
      │     ├──component_input((c|C3), (d|D2))
      │     ├──D1 > D2
      │     ⊟──not
      │           └──termination
      ⊟──C: and
      │     ├──dummy
      │     ⊟──PXOR
      │           ⊟──Prob 0.5
      │           │     └──vote_for((c|C1), (c|C2))
      │           ⊟──OTHERWISE
      │                 └──vote_for((c|C1), (c|C3))
      └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
      ├──V: C1 : between(1, components)
      ├──V: C2 : between(1, components)
      ⊟──A: and
      │     ├──vote_for((c|C1), (c|C2))
      │     ⊟──not
      │           └──count_for(components+1)
      ├──C: vote_for((c|C1), (c|C2))
      └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
      ├──V: C1 : between(1, components)
      ├──V: C2 : between(1, components)
      ⊟──A: and
      │     ├──vote_for((c|C1), (c|C2))
      │     ⊟──not
      │           └──counting_started
      ⊟──C: and
      │     ⊟──forall
      │     │     ├──V: C : between(1, components)
      │     │     └──votes((c|C), 0)
      │     ├──count_for(1)
      │     └──counting_started
      └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
      ├──V: C1 : between(1, components)
      ├──V: C2 : between(1, components)
      ├──V: V : between(0, components)
      ⊟──A: and
      │     ├──counting_started
      │     ├──count_for(C1)
      │     ├──vote_for((c|C1), (c|C2))
      │     └──votes((c|C2), V)
      ⊟──C: and
      │     ├──counting_started
      │     ├──count_for(C1+1)
      │     └──votes((c|C2), V+1)
      └──EFGH: efgh(0, 0, 1, 1)

⊟──leadsto
      ├──V: C1 : between(1, components)
      ├──V: C2 : between(1, components)
      ├──V: V : between(0, components)
      ⊟──A: and
      │     ├──counting_started
      │     ├──count_for(C1)
      │     ⊟──not
      │     │     └──vote_for((c|C1), (c|C2))
      │     └──votes((c|C2), V)
      ⊟──C: and
      │     ├──counting_started
      │     ├──count_for(C1+1)
      │     └──votes((c|C2), V)
      └──EFGH: efgh(0, 0, 1, 1)
```

```
leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: x1 : between(0, components)
    V: x2 : between(0, components)
    A: and
        count_for(components+1)
        votes((c|C1), x1)
        votes((c|C2), x2)
        x1 >= x2
    C: weak_better_than((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    V: x1 : between(0, components)
    V: x2 : between(0, components)
    A: and
        count_for(components+1)
        votes((c|C1), x1)
        votes((c|C2), x2)
        x1 > x2
    C: strong_better_than((c|C1), (c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: x1 : between(0, components)
    A: and
        count_for(components+1)
        votes((c|C1), x1)
    C: strong_better_than((c|C1), (c|C1))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: X : between(1, components)
    A: forall
        V: Y : between(1, components)
        weak_better_than((c|X), (c|Y))
    C: possible_component((c|X))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: C1 : between(1, components)
    V: C2 : between(1, components)
    A: and
        possible_component((c|C1))
        possible_component((c|C2))
        C1 > C2
    C: and
        dummy
        PXOR
            Prob 0.33
                active_component((c|C1))
            Prob 0.33
                active_component((c|C2))
            OTHERWISE
                and
                    active_component((c|C1))
                    active_component((c|C2))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: X : between(1, components)
    A: forall
        V: Y : between(1, components)
        strong_better_than((c|X), (c|Y))
    C: active_component((c|X))
    EFGH: efgh(1, 1, 1, 1)

leadsto
    V: C : between(1, components)
    V: D : between(1, data)
    A: and
        active_component((c|C))
        component_allowed((c|C))
        component_output((c|C), (d|D))
    C: and
        data((d|D))
        new_data
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: D : between(1, data)
    A: data((d|D))
    C: data((d|D))
    EFGH: efgh(0, 0, 1, 1)

leadsto
    V: D : between(1, data)
    A: and
        data((d|D))
        goal_data((d|D))
    C: termination
    EFGH: efgh(0, 0, 1, 1)
```

```
leadsto
    V: D1 : between(1, data)
    V: D2 : between(1, data)
    A: and
        data((d|D1))
        data((d|D2))
        goal_data(and((d|D1), (d|D2)))
    C: termination
    EFGH: efgh(0, 0, 1, 1)
```