Automated Evaluation of Coordination Approaches

Tibor Bosse, Mark Hoogendoorn, and Jan Treur

Vrije Universiteit Amsterdam, Department of Artificial Intelligence
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
{tbosse, mhoogen, treur}@cs.vu.nl, http://www.cs.vu.nl/~{tbosse, mhoogen, treur}

Abstract. How to coordinate the processes in a complex component-based software system is a nontrivial issue. Many different coordination approaches exist, each with its own specific advantages and drawbacks. To support their mutual comparison, this paper proposes a formal methodology to automatically evaluate the performance of coordination approaches. This methodology comprises (1) creation of simulation models of coordination approaches, (2) execution of simulation experiments of these models applied to test examples, and (3) automated evaluation of the models against specified requirements. Moreover, in a specific case study, the methodology is used to evaluate some coordination approaches that originate from various disciplines.

1 Introduction

Coordinating 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 the problem how all these fragments can be combined into a coherent system. To solve such a problem, many different coordination approaches have been proposed, each having its advantages and drawbacks. Important questions when choosing such a coordination approach are the suitability, correct functioning, and efficiency of the approach for the particular component-based system.

This paper presents a methodology to enable a comparison of such factors for the different coordination approaches in a series of test examples. First of all, this methodology allows for the creation of simulation models for each of the coordination approaches. Secondly, it comprises an engine which simulates the different coordination approaches for a variety of test examples. Finally, the methodology consists of an automatic evaluation of the outcome of the simulations against specified requirements (e.g. successfulness and efficiency).

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 new approaches to coordination of complex component-based software systems. As a first step, this paper evaluates a number of such approaches in a specific case study, to see to what extent they provide satisfactory solutions.

First, in Section 2 the methodology and supporting software tools are described. In Section 3 a number of coordination approaches obtained from the literature in various disciplines are briefly introduced. Section 4 describes a set of test examples that can be used as input for the evaluation of the coordination approaches. In Section 5 the simulations that were undertaken to evaluate the usefulness of the coordination approaches for the test examples are briefly discussed. Section 6 presents the results, and Section 7 is a final discussion.

2 Evaluation Method

To explore possibilities to address the coordination problem, an evaluation methodology, supported by a software environment, has been created: (a) a number of *coordination approaches* are selected, (b) a number of *test examples* representing specific software component configurations are chosen, (c) based on each of these coordination approaches a *simulation model* is formally specified, (d) related to the test examples, relevant *requirements* are formally specified in the form of relevant dynamic properties, (e) *simulations* are performed where selected coordination approaches are applied to the chosen test examples, resulting in a number of simulation traces, and (f) the simulation traces are *evaluated* (automatically) for the specified requirements.

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, e.g., 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 but representative test examples have been identified. As a source, the library of workflow patterns described in [1] has been used. The examples given there have been extended with input and output data and information flow channels.

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 approach 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 and the resulting simulation traces can be logically analysed, supported by another software environment.

To evaluate the resulting simulation traces, in the first place it is needed to identify the relevant properties, serving as requirements, on which such an evaluation should be based. A number of aspects can be covered in such requirements. 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, then it is not effective. A second aspect to evaluate is efficiency: to what extent time is wasted in the process to obtain the eventual goals. 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.

To support the evaluation method described a software environment is used: to logically specify simulation models and to execute these models in order to get simulation traces, and to specify relevant dynamic properties and to check such properties against simulation traces. For the simulation part, the language LEADSTO is used [6], based on a variant of Executable Temporal Logic [4]. The basic building blocks of this language are causal relations of the format $\alpha \rightarrow\!\!\!\!\rightarrow_{e, f, g, h} \beta$, which 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.

where α and β are 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. For the analysis part the language TTL is used [7]. This predicate logical language supports formal specification and analysis of dynamic properties, covering both qualitative and quantitative aspects. TTL is built on atoms referring to states, time points and traces. A *state* of a process for (state) ontology Ont is an assignment of truth values to the set of ground atoms in the ontology. The set of all possible states for ontology Ont is denoted by STATES(Ont). To describe sequences of states, a fixed *time frame* T is assumed which is linearly ordered. A *trace* γ over 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 *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 state in γ at time point t is denoted by state(γ, t). These states can be related to state properties via the formally defined satisfaction relation |=, comparable to the Holds-predicate in the Situation

Calculus: state(γ, t) $\models$ p denotes that state property p holds in trace γ at time t. Based on these statements, dynamic properties can be formulated in a formal manner in a sorted first-order predicate logic, using quantifiers over time and traces and the usual first-order logical connectives such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$. 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 formal verification of such properties against a set of (simulated or empirical) traces.

3 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 3.1 focusses on the *behavior networks* approach by Pattie Maes [17]. Section 3.2 describes Selfridge's *pandemonium* model [22], and Section 3.3 addresses the decision-making techniques known as *voting methods* [18]. These approaches were chosen for two reasons. First, because they are well-known approaches in the (wider) literature in various disciplines on coordination. Second, because together they more or less cover the area of different coordination approaches: the behavior networks use a rather *global* and sequential strategy (i.e., the approach determines which component is activated based on global information concerning all components), whereas voting methods and (especially) the pandemonium model use a *local* and possibly nonsequential strategy (i.e., the components involved only use information about themselves or their direct neighbours to determine which component is activated).

3.1 Behavior Networks

Behavior networks have been introduced by Pattie Maes in 1989. She distinguishes *competence modules* within a system, where each module is specified by a tuple containing four elements: (1) a list of preconditions to be fulfilled before a competence module can become active; (2) the competence module's action in terms of an add list; (3) the competence module's actions in terms of a delete list; (4) a level of activation. A competence module is said to be executable in case the list of preconditions is fulfilled. A network of competence modules is created via three types of links: successor links (a link from x to y for every element on the add list of x which is on the preconditions list of y), predecessor links (a link from x to y for every element on the precondition list of x which is on y's add list), and conflictor links (a link from x to y for every element on the precondition list of y which is on x's delete list). Through these links the competence modules activate and inhibit each other, so that "after some time the activation energy accumulates in the modules that represent the 'best' actions to take given the current situation and goals" [17]. The patterns of

these spreading activations among modules, as well as the input of new activation energy into the network, is determined by the state of the environment and goals via three ways: activation by state (add activation to modules that (partially) match the current state), activation by goals (add activation to modules which (partially) achieve the goals), and inhibition by protected goals (remove activation from modules that (partially) remove the protected goals). Thereafter, activation spreads through the network via activation of successors, activation of predecessors, and inhibition of conflictors. After having spread the activation, a decay phase makes sure the overall activation remains constant within the network. Once performed, a competence module fires in case it is executable, the activation is over the threshold that has been set, and it is the competence module with the highest activation. In case the module indeed fires, its activation goes to 0, and all thresholds return to their normal value. In case no module fires, the threshold is reduced by 10%. For more mathematical details, see [17].

3.2 The Pandemonium Model

In 1958, Selfridge proposes an approach he calls pandemonium, to enable pattern recognition [22]. This is a system composed of primitive constructs called *demons*, each representing a possible pattern. Once an image is presented, each of the demons computes the similarity of the image with the pattern it represents, and gives an output depending monotonically on that similarity. Finally, a decision demon selects the pattern belonging to the demon whose output is largest.
Jackson [14] extends this idea to a theory of mind. Besides demons involved in perception, he also identifies demons that cause external actions and demons that act internally on other demons. Jackson pictures the demons as living in a stadium. Almost all of them are the crowd, cheering on the performers. The remainder of the demons 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 replaces one of those currently performing which is sent 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. The system starts off with initial built-in links between the demons. New links are made between demons, and existing links are strengthened in proportion to the time they have been together on the field, plus the gain of the system (i.e., when all is going well, the gain is higher).

3.3 Voting Methods

The concept of *voting* refers to a wide collection of techniques that are used to describe decision-making processes involving multiple agents. Although originating from political science, voting methods are currently used within a number of domains, including game theory (where they are used as methods for conflict resolution) and pattern recognition (where they are used to combine classifier outputs).

The general idea of voting methods is rather intuitive, and is comparable to the techniques used in elections. Consider a set of agents N, and a set of possible outcomes S of an election. Each agent $i \in N$ has preferences over the outcomes: $\leq_i \subseteq S \times S$. The voting approach uses a function F that selects a candidate outcome S, given the preferences of the voters. A simple instance of F would be to count all votes, and to select the outcome with the highest amount of votes. However, a large number of (more complex) voting approaches exist. These can roughly be divided into three classes: *unweighed voting methods* in which each vote carries equal weight, *confidence voting methods* in which voters can express a degree of preference for a candidate, and *ranked voting methods* in which the voters are asked for a preference ranking over the candidates. See [18] for an overview of different voting methods.
As mentioned above, voting methods are currently used in many different domains, such as game theory and pattern recognition. In this paper it will be explored whether they are of any use to solve coordination problems in complex (component-based) software systems. To this end, the electorate will be filled in by certain components, and the candidates by the possible activations of components.

4 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 [1]. These patterns can be seen as building blocks for more complex patterns occurring in real-life component-based systems. In total, seven test examples have been described, two of which are discussed below. A test example consists of a number of components, called {C1, C2,..}, and several types of data, called {d1, d2,..}. Different components need different data as input, and create different data as output. The complete set of test examples is described in [5].

Pattern 1 - Sequence

The first pattern is straightforward: it involves three components. After completion of the first component, the second component is activated, and after completion of the second, the third component is activated.

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 is given in Figure 1. 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 approach 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 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. Another assumption is that data cannot be removed. Thus, once data is written to the shared repository, it will stay there. Other approaches such as explicit communication channels can however easily be incorporated into the methodology.

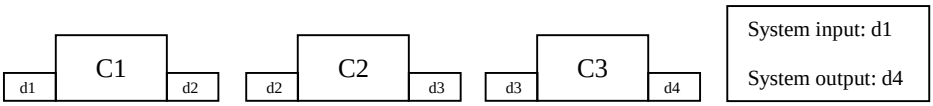

Fig. 1. Test example 1 - Sequence

Pattern 7 - Synchronizing Merge

Pattern 7 involves four components. After completion of the first component, there is a choice between the second and third component: either one of them can be activated, or both. In case one of them is activated, the fourth component is activated after this component has completed. In case both of them are activated, the fourth component is activated after both have completed.

The test example that was created on the basis of this pattern is shown in Figure 2. As can be seen in the figure, in this 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 variant (called C4) with d4 as input, one variant (called C5) with d5 as input, and one variant (called C6) with d4 and d5 as input.

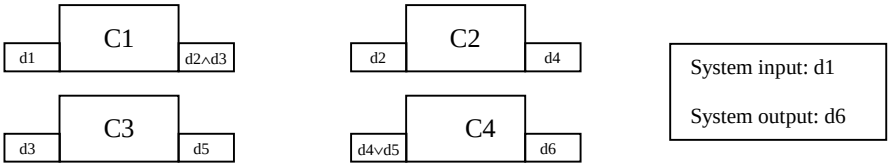

Fig. 2. Test example 7 - Synchronizing Merge

5 Simulation

To compare the coordination approaches described in Section 3 against the above test examples, a number of simulation experiments have been performed. First, the three selected coordination approaches have been implemented in the LEADSTO simulation language, see [5]. Next, the implemented simulation models have been applied to the test examples. The simulation models for the behavior networks, the pandemonium, and the voting method, are addressed, respectively, in Section 5.1, 5.2,

and 5.3. For each simulation model, an example simulation trace (resulting from applying the model to test example 7) is provided.

5.1 Behavior Networks Simulation

The simulation model for Maes' behavior networks is created on the basis of the mathematical model as presented in [17]. There is one difference: within the simulation model, the lowering of the threshold is not performed, as the available data does not change due to external influences (i.e., the highest executable component will remain the highest until a component has been activated). Therefore, the highest executable component is simply selected, avoiding unnecessary computation. The LEADSTO specification for the approach roughly corresponds to the description in Section 3.1.

Figure 3 presents a simulation trace that has resulted from executing the approach on test example 7. Initially, the data present is set to d1: data(d1). Furthermore, the goal is set to d6 for this particular scenario: goal(d6). Before starting, the activation value, referred to as the alpha value of the components 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, c1, 0), alpha(0, c2, 0), alpha(0, c3, 0), alpha(0, c4, 0), alpha(0, c5, 0), and alpha(0, c6, 0). Thereafter calculations are performed to determine the activity within the different components: The input from the current state is calculated (i.e. given the current data available, calculate the activation caused for the different components) as well as the input from the goals. Since only C4, C5, and C6 have a goal as an output, these components are the only ones to receive activation through this source. Due to the fact that the previous alpha value is 0, no activation is spread around the network. The next alpha value for the six components present in the system is therefore obtained by simply summing up the input from the goals and state per component, and normalizing it to 1: alpha(1, c1, 0.25), alpha(1, c2, 0), alpha(1, c3, 0), alpha(1, c4, 0.25), alpha(1, c5, 0.25), and alpha(1, c6, 0.25). As a result, component C1 is activated, as this is the executable component with the highest alpha value: activated(c1). Due to the activity of component C1, its output data is generated, which is shown in the trace: the presence of data d2 and d3: data(d2) and data(d3).

A new round of computation is performed; the input from the goals remains the same, as these have not changed. However, the input from the current state changes, due to the additional data d2 and d3 being present. Furthermore, activation is now spread through the network, since the previous alpha values are non-zero. After calculation and normalisation the following alpha values are the result: alpha(1, c1, 0.0425532), alpha(1, c2, 0.255319), alpha(1, c3, 0.255319), alpha(1, c4, 0.148936), alpha(1, c5, 0.148936), and alpha(1, c6, 0.148936). Since both C2 and C3 are executable and have the highest alpha value, one of them is randomly selected; in Figure 3 this is component C2.

As can be seen in the figure, after activation of C2, component C3 is activated. Finally, C4 is activated, outputting the goal data, which results in termination.

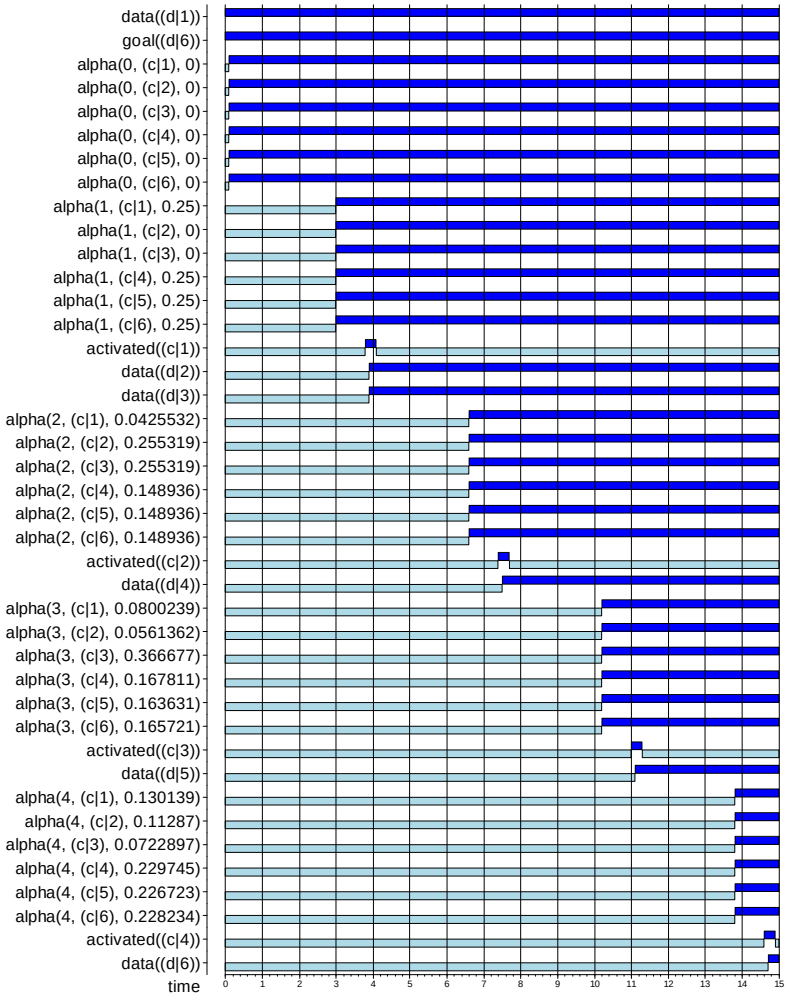

Fig. 3. Simulation Trace - Behavior Networks against Test Example 7

5.2 Pandemonium Simulation

The pandemonium is used as described in Section 3.2, 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. Whenever new data has been added to the repository, a new round starts in which all components can *shout*. The idea is that, the more urgent a component thinks it is for him to be activated, the louder it will shout. The component that shouts loudest will be allowed to start processing. In case two components shout with exactly the same strength, then either the first component, or the second component, or both are activated (this decision is made randomly, with equal probabilities). When a

component is activated, this results in the component adding its output data to the *shared repository* (see Section 4), and the start of a new round.

To determine how loud they will shout, the components make use of a shout function. For different variants of the pandemonium model, different shout functions may be used. In the current model, each component 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 highest i1 for the set of components (represented by max_i)
- the highest o1 for the set of components (represented by max_o)

Given these elements, the shout value (i.e., the strength with which a component 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, β1, β2, β3, and β4 are real numbers between 1 and 1.5, indicating the importance of the corresponding factor. Several settings have been tested for these parameters. In the examples shown here, β1=1.4, β2=1.3, β3=1.1, and β4=1.2. Since the factors can never exceed 1, the shout value sv will be a value between 0 and 1.

Figure 4 depicts the simulation trace that has resulted from applying the pandemonium approach to test example 7. As the figure shows, initially the only data that is present is d1: data(d1). Based on this data, every component starts shouting. Component C1 shouts loudest (with strength 0.47, whilst the others shout with strength 0.0): shout(c1, 0.466516), shout(c2, 0.0), ..., shout(c6, 0.0). Thus, component C1 is selected to become active: active_component(c1). As a result, C1 creates data d2 and d3, which are stored at the repository as well: data(d2), data(d3). Then again, every component starts shouting. This time, both component C2 and C3 shout loudest (with strength 0.20, whilst the others shout with strength 0.0): shout(c1, 0.0), shout(c2, 0.203063), shout(c6, 0.0). As a result, both component C2 and C3 are selected to become active: active_component(c2), active_component(c3). Note that this selection is based on the assumption that multiple components may be activated at the same time. If this is not allowed, the approach would select one of the components at random. Next, component C2 creates data d4, and component C3 creates data d5. These data are stored at the repository: data(d4), data(d5). Again, every component starts shouting. Component C6 (which is a specific variant of C4, see the description of the example) shouts loudest (with strength 0.44): shout(c1, 0.0), shout(c2, 0.0), shout(c6, 0.435275). Thus, component C6 is selected to become active: active_component(c6). Eventually, component C6 creates data d6, which is stored at the repository: data(d6). Since d6 is the goal data, at this point the process terminates.

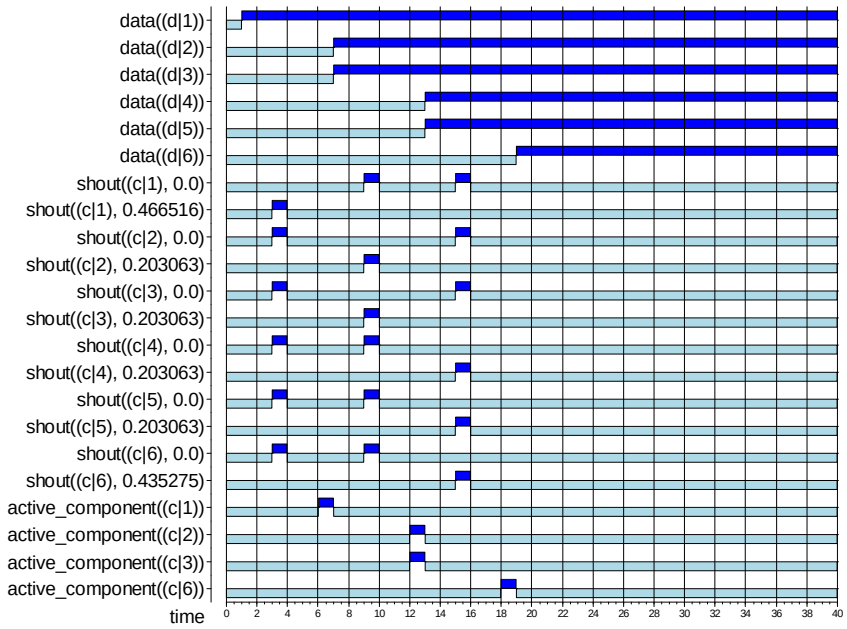

Fig. 4. Simulation Trace - Pandemonium against Test Example 7.

5.3 Voting Simulation

The simulation of the voting method uses the same assumptions as the pandemonium method, with one difference: instead of shouting, all components can *vote*. The idea is that each component can vote on only one component (possibly on itself). After all components have voted, the votes are counted, and the component with most votes will be allowed to start processing. To determine on whom they will vote, the components make use of a *voting procedure*. For different variants of the voting method, different voting procedures may be used. In the current model, each component follows the following procedure:

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

Note that this approach assumes a *local* perspective of the components. This means that each component only has knowledge about itself and its direct neighbours. For

example, each component knows which other components need the data that it produces as input, but does not know which data the other components produce as output.

Figure 5 depicts the simulation trace that has resulted from applying the voting approach to test example 7. Initially the only data that is present is d1: data(d1). Based on this data, every component starts voting: vote_for(c1, c1), vote_for(c2, c1), vote_for(c3, c1), vote_for(c4, c2). Component C1 receives 3 votes, component C2 receives one vote, and the other components receive no votes. Thus, component C1 is selected to become active: active_component(c1). As a result, C1 creates data d2 and d3, which are stored at the repository as well: data(d2), data(d3). Then again, every component starts voting: vote_for(c1, c3), vote_for(c2, c2), vote_for(c3, c3), vote_for(c4, c3). Component C3 receives 3 votes, component C2 receives one vote, and the other components receive no votes. Thus, component C3 is selected to become active: active_component(c3). Next, component C3 creates data d5, which is stored at the repository: data(d5). Voting starts again: vote_for(c1, c2), vote_for(c2, c2), vote_for(c3, c5), vote_for(c4, c2). Component C2 receives 3 votes, component C5 (which is a specific variant of C4) receives one vote, and the others receive no votes. Thus, component C2 is now selected to become active: active_component(c2). Component C2 creates data d4, which is stored at the repository: data(d4). In the next round, the components vote as follows: vote_for(c1, c2), vote_for(c2, c6), vote_for(c3, c6), vote_for(c4, c6). Component C6 (which is a specific variant of C4) receives 3 votes, component C2 receives one vote, and the others receive no votes. Consequently, component C6 is selected to become active: active_component(c6). Eventually, component C6 creates data d6, which is stored at the repository: data(d6). Since d6 is the goal data, at this point the process terminates.

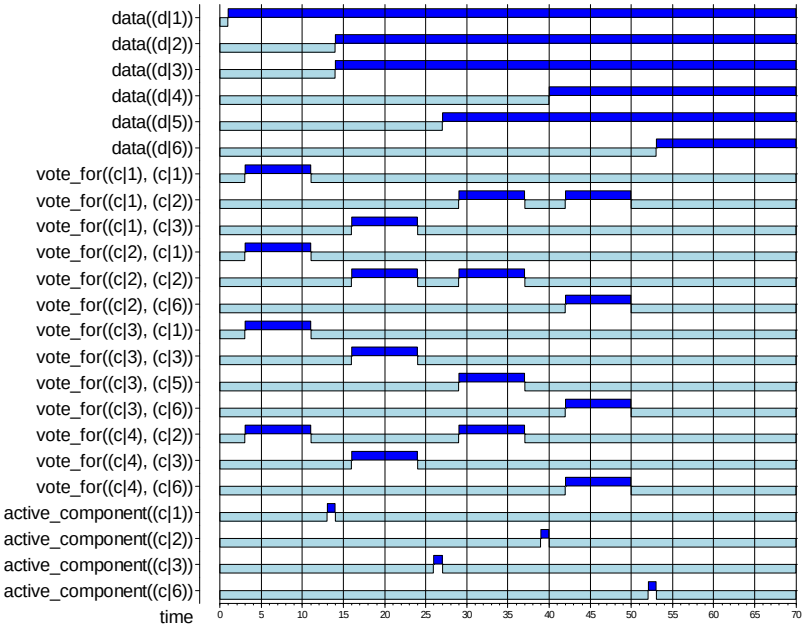

Fig. 5. Simulation Trace - Voting against Test Example 7.

6 Evaluation

This section addresses the evaluation of the performance for the different approaches that have been simulated as described above. 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 approach. Finally, patterns can be specified which are (allowed) to occur in the component configurations used as test examples, and it can be checked whether a coordination approach indeed identifies these patterns. To enable automated checking of the results of the approaches, a formal specification of the different types of properties is required. For this purpose, the language TTL introduced in Section 2 is used. After such a formal description has been obtained, the automated TTL-checker can be used to see how well the approach performs.

6.1 Successfulness

The first property to be checked is called successfulness. Informally, this property states that in the trace γ all goal data d will eventually be derived. Formally:

successfulness(γ:TRACE) ≡
$\forall$t:TIME, d:DATA [state(γ, t) = goal(d) $\Rightarrow$
$\exists$t2:TIME [t2 $\geq$ t $\wedge$ state(γ, t2) |= data(d)]]

The results of automatically checking this property against the traces that were generated in the simulation show that all approaches eventually find the solution for the examples that have been used. Prerequisite is that there must exist at least one path to the solution.

6.2 Efficiency

Efficiency can be viewed from multiple perspectives. First, one can look at the efficiency of the solution path found by the approach. For now, it is assumed that each component takes an equal amount of time to obtain its output. Therefore, the most efficient solution is simply the solution in which the least amount of components have been activated. Another way to describe efficiency is the efficiency of the approach itself, i.e., the amount of computation time the approach needs to generate a solution. The approach taken in this section is to check whether the shortest activation path is used to reach the goals that are set. For the formalisation of this property, it is assumed that the length of the shortest path is known for the particular example being checked:

efficiency(γ:TRACE, shortest_path:INTEGER) ■
successfulness(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) ≡
tb < te ∧ state(γ,te) |≠ activated(c) ∧
[∀t tb≤t<te ⇒ state(γ,t) |= activated(c)] ∧
∃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) ≡
∃c:COMPONENT, tb:TIME, te:TIME
has_activation_interval(γ, c:COMPONENT, tb:TIME, te:TIME) ∧
[∀c2:COMPONENT, tb2:TIME, te2:TIME
[has_activation_interval(γ, c2:COMPONENT, tb2:TIME, te2:TIME) ⇒ c = c2 ∧ tb = tb2 ∧ te = te2]]

Table 1 shows the outcome of checking the property efficiency in the TTL Checker for the generated traces. A plus indicates that in all generated traces the efficient solution was found; a minus indicates that no efficient solution is found in at least one of the generated traces.

Table 1. Efficiency of the different approaches 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. This is mainly due to the fact that the voting components have only *local* information. As a result, their voting behaviour is not always fully rational. This problem could be solved by allowing a more global perspective for the components.

For the synchronizing merge and the multi-choice (which can be described as the synchronizing merge without component C4), the behavior networks approach 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 approach could probably prevent this from occurring. Furthermore, in the synchronizing merge case, both C2 and C3 are activated whereas C4 only needs one input to generate 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 networks: sometimes both C2 and C3 are activated simultaneously, whilst only one of them is required.

The voting method 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 method is sometimes inefficient with respect to the synchronizing merge. Here, again the same situation occurs as with the behavior networks and the pandemonium: sometimes both C2 and C3 are activated, where only one of them is necessary.

6.3 Specifying and Checking Patterns.

As has been mentioned, certain expected patterns can be specified for component configuration examples, and it can be checked whether these patterns are indeed found by the different approaches. For the test examples used in this document, the component configuration specifications originate from workflow patterns. Therefore, the patterns taken for the test 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 [2] were used and specified in TTL, for example the before relation:

before(b1:TIME, e1:TIME, b2:TIME, e2:TIME) ≡ e1 < b2

Below, workflow patterns 1 and 7 are specified using TTL expressions. For both patterns, all traces are first summed up in an informal fashion (according to perspective 1 above). After that, the formal TTL expressions specifying the constraints between the activation intervals of the different components are shown (according to perspective 2). The complete set of seven TTL expressions can be found in [5].

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)

Pattern 7 - Synchronizing Merge

Possible traces: ABD, ACD, ABCD, ABCD, A B|C D.
Here, "B|C" indicates that B and C are activated simultaneously.

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)]

Automated checks have pointed out that the behavior networks, pandemonium, and voting approaches always find the patterns that have been identified. In the parallel split case, the success of the voting approach 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 his perspective, a trace satisfies a pattern when the components as prescribed by the patterns 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 activations) a *closed world assumption* version of the property has been specified as well.

Since the abstract way of modelling used here is not computationally expensive, checking a property against a trace on average took no more than one second. For more information about the complexity of checking TTL expressions, see [7].

7 Discussion

To conclude, this paper presented a formal methodology to evaluate and compare the performance of different coordination approaches. The methodology comprises the creation of simulation models for the coordination approaches, the execution of simulation experiments of these models applied to test examples, and their automated evaluation against specified requirements. In a specific case study, the methodology was used to evaluate three well-known coordination approaches from the literature. During this case study, the simulation approach turned out quite beneficial. Within a reasonable time, a nontrivial number of approaches have been tested against a nontrivial number of cases: 3 x 7 = 21 combinations have been explored. Furthermore, the automated checks of dynamic properties against generated traces have turned out useful to evaluate the simulations for the different approaches against requirements. Finally, an existing library of workflow patterns [1] turned out an

appropriate source for cases to be explored, although their specification also needs to cover data flow aspects. It was not too difficult to add such data flow aspects.

Concerning the specific case study performed, the voting, pandemonium and behavior networks approach have been thoroughly evaluated with respect to a number of relevant performance indicators, namely successfulness, efficiency, and pattern checks. All approaches turned out effective in finding the solution in all cases. However, none of the approaches is always efficient for all patterns. The behavior networks and pandemonium approaches 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 approach 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 approaches also find the patterns specified for each of the component configuration examples.

All in all, when comparing the different coordination approaches, 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 approach 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 component only needs information about its own input and output data. In between is the voting approach, which needs information about itself and its direct neighbours. When comparing the approaches on required computation time, the behavior networks approach takes far more computation time than the other approaches. This has two causes: first, due to the fact that all global information is used within the approach, it has a lot more information to take into consideration. Second, both for the voting and pandemonium approach the calculations per component can be performed in parallel, which can not be done in the behavior networks approach.

Work related to the approach presented in this paper can, first of all, be found in the field of action selection mechanisms (also called behaviour coordination mechanisms) in robotics. Pirjanian [19] presents an overview of several mechanisms used in that particular field, including a classification of these mechanisms. He identifies two main streams: arbitration and command fusion. In the arbitration approach, one behaviour is arbitrarily selected from a group of competing ones, giving it the ultimate control. For command fusion mechanisms however, recommendations are combined from multiple behaviours to form a control action that represents their consensus. The behaviour networks approach as presented by Maes [17] is an example of an arbitration mechanism, whereas both voting and the pandemonium model can be placed in the command fusion category. Tyrrell [23] presents a comparison between several mechanisms for action selection, using a simulator of an animal world. The comparison approach is however not formal like the approach presented in this paper. Furthermore, the framework for comparison is not generic, but developed for a specific case study, making it hard to generalise the results obtained. Another related field can be found within multi-agent systems, where coordination mechanisms play an essential role to ensure a proper functioning of the system as a whole. These coordination mechanisms address types of interactions and

agreements between the different agents that were not considered in this paper. For a comparison between different coordination mechanisms in agent systems, see for example [8]. Concerning other related work, coordination models and languages for interfacing between components often focus on how different components within a software system can interact, see for example [3]. Due to the assumption of data being available and interpretable for all components, these component interaction models have not been considered in this paper, but can easily be incorporated in the methodology.

The methodology presented in this paper is supported by two software environments: the LEADSTO environment for simulation [6], and the TTL environment for verification of properties [7]. For simulation, various other approaches exist, such as the Dynamical Systems Theory [20], Executable Temporal Logic [4], PLC automata [9], qualitative reasoning (see, e.g., [10]), and stochastic pi-calculus (as used in [12]). For verification of properties, alternative approaches are standard temporal languages such as LTL and CTL [13], and calculi like the situation calculus [21] and the event calculus [15]. See, respectively, [6] and [7] for an extensive comparison of LEADSTO and TTL with these approaches.

Finally, the work as reported has led to a number of ideas for further research. 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 (and possibly better) approaches. Some possible future extensions are allowing *preference* for certain components, allowing a *dynamic environment*, and enabling the components to process *partial data*.

Acknowledgements

This work has been performed as part of a project funded by CAMS-Force Vision, the software development company associated with the Royal Netherlands Navy. Moreover, the authors are grateful to Egon van den Broek, Rob Duell, Andy van der Mee, and Bas Vermeulen for various fruitful discussions.

References

1. Aalst, W.M.P. van der, Hofstede, A.H.M. ter, Kiepuszewski, B., and Barros, A.P. Workflow Patterns. QUT Technical report FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.
2. Allen, J. F. Maintaining knowledge about temporal intervals. In: *Communications of the ACM*, 26, 1983, pp. 832-843.
3. Arbab, F. Reo: A Channel-based Coordination Model for Component Composition, *Mathematical Structures in Computer Science*, Cambridge University Press, vol. 14, No. 3, 2004, pp. 329-366.
4. Barringer, H., Fisher, M., Gabbay, D., Owens, R., and Reynolds, M. *The Imperative Future: Principles of Executable Temporal Logic*, John Wiley & Sons, 1996.

5. Bosse, T., Hoogendoorn, M., and Treur, J. Coordination Approaches for Complex Software Systems. Technical report 06-04ASRAI, Vrije Universiteit Amsterdam, Amsterdam, 2006. URL: http://hdl.handle.net/1871/9195.
6. Bosse, T., Jonker, C.M., Meij, L. van der, and Treur, J. 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.
7. Bosse, T., Jonker, C.M., Meij, L. van der, Sharpanskykh, A, and Treur, J. A Temporal Trace Language for the Formal Analysis of Dynamic Properties. Technical Report, Vrije Universiteit Amsterdam, Department of Artificial Intelligence, 2006.
8. Bourne, R., Shoop, K., and Jennings, N. Dynamic evaluation of coordination mechanisms for autonomous agents. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence*, Lecture Notes in Artificial Intelligence. Springer Verlag, 2001, pp. 155-168.
9. 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. Springer-Verlag, 1997, pp. 111-125.
10. Forbus, K.D. *Qualitative process theory*. Artificial Intelligence, vol. 24, no. 1-3, 1984, pp. 85-168.
11. Franklin, S. Artificial Minds, MIT Press, Cambridge Massachusetts, 1997.
12. Gardelli, L., Viroli, M., Omicini, A.: On the Role of Simulations in Engineering Self-Organizing MAS: the Case of an Intrusion Detection System in TuCSoN. In: *3rd International Workshop "Engineering Self-Organising Applications" (ESOA)*, 2005, pp. 161-175.
13. Goldblatt, R. *Logics of Time and Computation*, 2nd edition, CSLI Lecture Notes 7, 1992.
14. Jackson, J.V. Idea for a Mind, *SIGGART Newsletter*, no 181, 1987, pp. 23-26.
15. Kowalski, R. and Sergot, M.A. A logic-based calculus of events, *New Generation Computing*, 4, 1986, pp. 67-95.
16. Lindsay, P. H., and Norman, D. A. *Human Information Processing: An Introduction to Psychology*. Academic Press, Inc., New York, 1977.
17. Maes, P. How to do the right thing. *Connection Science*, 1989. 1(3): pp. 291-323.
18. Ordeshook, P. *Game theory and political theory: An Introduction*. Cambridge: Cambridge University Press, 1986.
19. Pirjanian, P. *Behavior coordination mechanisms -- state-of-the-art*. Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems, School of Engineering, University of Southern California, October 1999.
20. Port, R.F., and Gelder, T. van (eds.) *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge, Mass, 1995.
21. Reiter, R. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, Cambridge MA: MIT Press, 2001.
22. Selfridge, O. G. Pandemonium: a paradigm for learning in mechanization of thought processes. In *Proceedings of a Symposium Held at the National Physical Laboratory*, London, November 1958, pp. 513-526.
23. Tyrrell, T. *Computational Mechanisms for Action Selection*, PhD thesis, University of Edinburgh, 1993.