A Specification Language for Coordination in Agent Systems

Tibor Bosse, Mark Hoogendoorn, Radu Serban, and Jan Treur
Vrije Universiteit Amsterdam, Department of Artificial Intelligence
De Boelelaan 1081a, 1081 HV Amsterdam, the Netherlands
{tbosse, mhoogen, serbanr, treur}@cs.vu.nl

Abstract

This paper introduces an executable coordination specification language, with which both pre-specified and more flexible and generic agent coordination approaches can be expressed. An iterative process was taken to define this language. First, useful language elements were defined, after which example coordination approaches were specified using this language. The language was extended incrementally with new language elements whenever new concepts were required to enable specification of the example coordination approaches. The approaches were simulated and tested using particular test cases. Finally, an evaluation of the coordination approaches was performed by means of formal verification.

1. Introduction

A component-based system is a generic term for a variety of systems, such as, among others, agent-based systems. As component-based software systems become increasingly complex, so does the specification of coordination for such a system. In addition, software systems can be dynamic, in the sense that components dynamically enter or leave the system. As a result, for such complex dynamic systems, exhaustively specifying the activation sequences of components, for example in a traditional, centralized manner, is no longer an option: the components that are available for computation and their ideal activation sequence are not known in advance. Furthermore, it is not always possible or desirable to have coordination information available in a global or centralized manner.

As a consequence, more generic and flexible coordination approaches have been proposed, including pandemonium models [11], behavior networks [9], and voting models [10]. In contrast to the more traditional approaches, which are based on qualitative, logical specifications, such alternative methods usually involve quantitative, numerical calculation methods, and often work in a more decentralized manner. In [2] a methodology for the evaluation and comparison of such coordination approaches has been proposed, and a number of such approaches have been evaluated.

The transition from a traditional way of specifying coordination by pre-defined coordination sequences to such more generic and flexible coordination strategies is not a trivial matter. Current coordination specification languages as, for example, described in [5;7] are typically unable to specify such coordination approaches. Moving towards a new approach for coordination requires a more expressive coordination language, allowing, for example, more generic types of expressions with variables and quantifiers, and numerical relationships. To address this problem, this paper proposes a coordination language that can express both pre-defined coordination sequences, as well as generic, flexible coordination approaches.

2. View on Coordination

In Figure 1, the viewpoint taken in this paper on coordination in a component based system is shown. From a conceptual perspective, the processing done by a component for coordination purposes can be distinguished from the actual processing of data to fulfill a coordination-independent computation. On the top level, above the dashed line, the coordination part of the entire system is shown. On this *coordination level*, reasoning takes place about the coordination within the component-based system. This process can be either a centralized or a distributed process. Input for this coordination process is coordination information received from the various components and links, whereas the output of this coordination level is coordination information for the components and links within the component-based system. On the *data level*, the components and links themselves are shown. Each component has two input layers: one for coordination

information (the upper square at the left side of the component), and one for data information (the lower square). Output is generated on both levels, depicted by the squares at the right side of a component.

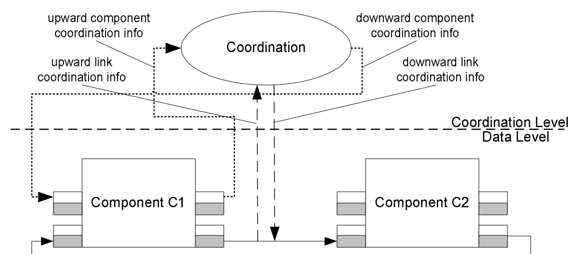

Figure 1. Levels of Coordination within a System

Each link on the data level connects the data output of a component to the data input of another component or can receive and generate coordination information. Note that this is a conceptual picture at an abstract level. There is no commitment in the degree to which the coordination reasoning process itself is centralized or distributed over the components. This is a parameter that is left open in this conceptual picture.

3. Coordination Language

This section presents the actual coordination specification language. The language is a reified temporal order-sorted predicate logic language; cf. [6]. This means that state ontologies are defined to express state terms, and on top of that a time ontology is used so that by full predicate logic expressivity temporal statements can be specified.

Sort	Description
STATE	A sort for states.
TIME	Sort indicating time.
TRACE	A trace indicates a time ordered sequence of states.
STATPROP	A sort for terms indicating state properties
REAL	Sort for real numbers.
INFO_TYPE	A type of information, which can possibly be a grouping of multiple other information types. Furthermore, it can contain information elements that specify a specific value of an element within this information type.
COMPONENT	A component within the component-based system (can for instance be an agent).
FOCUS	An identifier of a focus that has been set for a particular architectural object.
INFO_ELEMENT	A specific element of information, such as explained under INFO_TYPE.
SIGN	Indicates whether an INFO_ELEMENT holds (indicated by 'pos'), or does not hold ('neg').
SIGNED_INFO_ELEMENT	An INFO_ELEMENT grouped with a SIGN.

Table 1. Partial List of Sorts used in language

Relations	Description	Part
state: TRACE x TIME → STATE	This function indicates the state of a trace at a specific time point	T
holds_just_before: STATPROP x TIME x TRACE	STATPROP holds just before the time point specified within the specified trace.	T
estimated_processing_time: REAL	Indicates the estimated (maximum) processing time required by a component in order to produce outputs from available inputs.	SC
link_type_relation: INFO_TYPE x COMPONENT x INFO_TYPE x COMPONENT	A communication link exists from the INFO_TYPE[1] of the output of the first COMPONENT[1], to the input of the INFO_TYPE[2] of the second COMPONENT[2].	SD
input_information_type: INFO_TYPE	The INFO_TYPE is an input information type.	SD
awake	Information can be processed.	DC
asleep	Information cannot be processed.	DC
busy	The component is currently busy with processing	DC
non_busy	The component is not busy with processing.	DC
is_input_focus: FOCUS	The input focus set defined by FOCUS is currently in use.	DC
currently_needed_input_ for_output: INFO_TYPE x INFO_TYPE	The set of input types in INFO_TYPE is still expected in order to produce an output element of the type INFO_TYPE	DD
information_at_input: SIGNED_INFO_ELEMENT	The SIGNED_INFO_ELEMENT is available at the input.	DD
information_at_output: SIGNED_INFO_ELEMENT	The SIGNED_INFO_ELEMENT is available at the output.	DD

Table 2. Partial List of Predicates used in language

The main distinction the language makes is between ontologies for component characteristics (static) and for states (dynamic). Moreover, ontologies are distinguished by whether they address coordination information or data information. Furthermore, ontologies are related to their use within input, output or internal states. Finally, in addition to the state ontologies, a generic time ontology is used, to specify temporal relations, and a generic (support) ontology is included for elementary relations and functions such as ordering and calculations for numbers. Due to a lack of space, only examples of different parts of the language are shown, the full language specification is presented in [12]. Table 1 shows the sorts that are used throughout the example predicates presented in this section, whereas Table 2 shows the example predicates. The example predicates are classified according to the part of the language they belong to: time ontology (T); ontology for static coordination characteristics on the coordination level (SC) and data level (SD), and the ontology for dynamic coordination characteristics on the coordination level (DC), and data level (DD); In

order to reason about such predicates on the meta-level of the system, the following predicates are proposed:

selected_control_aspect_for :	DYN_OBJECT_EL x COMPONENT
monitored_control_aspect_for :	DYN_OBJECT_EL x COMPONENT
control_aspect, data_aspect :	DYN_OBJECT_EL

Here, DYN_OBJECT_EL represents a meta-description of the relations in the language expressed above. These can be used at the output interface of the coordination level, the input interface of the coordination level, and the meta-level of the (input and output) coordination interface of components at the data level.

4. Relation to TTL and LEADSTO

In this section it is shown how the coordination language relates to the Temporal Trace Language (TTL) [3], by adding certain state ontologies and definable temporal predicates. For TTL, verification tools are available that can as a result be used for the coordination language as well. Moreover, it is shown how an executable sublanguage of the coordination language can be considered a specialization of TTL's sublanguage LEADSTO by adding pre-specified state ontologies. As a result, the simulation tools available for the LEADSTO language [4] can be used.

In TTL, ontologies for states are formalized as sets of symbols in sorted predicate logic. For any state ontology Ont, the ground atoms form the set of *basic state properties* BSTATPROP(Ont). Basic state properties can be defined by predicates. The *state properties* based on a certain ontology Ont are formalized by the propositions (using conjunction, negation, disjunction, implication, and quantification) made from the basic state properties and constitute the set STATPROP(Ont). For the coordination language the pre-specified state ontologies as presented in Section 3 are available.

In order to express dynamics in TTL, important concepts are *states*, *time points*, and *traces*. A *state* S is an indication of which basic state properties are true and which are false, i.e., a mapping S: BSTATPROP(Ont) $\rightarrow$ {true, false}. The set of all possible states for ontology Ont is denoted by STATES(Ont). Moreover, a fixed *time frame* T is assumed which is linearly ordered. Then, a *trace* γ over a state ontology Ont and time frame T is a mapping γ : T $\rightarrow$ STATES(Ont), i.e., a sequence of states γ_t ($t \in T$) in STATES(Ont). The set of all traces over ontology Ont is denoted by TRACES(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, a certain state at time point t is denoted by state(γ, t). These states can be related to state properties via the formally defined satisfaction relation, indicated by the infix predicate |=, comparable to the Holds-predicate in the Situation Calculus. Thus, state(γ, t) |= 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 predicate logic, using the usual logical connectives such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and the quantifiers $\forall$, $\exists$ (e.g., over traces, time and state properties). The set DYNPROP(Ont, γ) is the subset of DYNPROP(Ont) consisting of formulae with γ occurring in which is either a constant or a variable without being bound by a quantifier. Note that the predicates of the time ontology introduced in Section 3 can be defined in terms of atoms of the form state(γ, t) |= p. For example,

holds_at(p, t, γ)	$\equiv$	state(γ, t) |= p
holds_during(p, t1, t2, γ)	$\equiv$	$\forall$t [t1≤t<t2 $\Rightarrow$ holds_at(p, t, γ)]

To model direct temporal dependencies between two state properties, not the expressive language TTL, but the simpler LEADSTO format is used. This is an executable format that can be used to obtain a specification of a simulation model in terms of 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 time interval with duration* g,
then after some delay (between e *and* f time units*)*
state property β *will hold for a time interval of length* h.

For a precise definition of the LEADSTO format in terms of the language TTL, see [3].

5. Case Study

To verify whether the language presented above can indeed be used to specify a variety of coordination approaches (from pre-specified to more generic, flexible approaches), a number of such coordination approaches have been specified using the language. This section presents one of these approaches and shows how it can be specified in the coordination language. Specification for other coordination approaches such as the pandemonium [11] can be found in [12]. Note that it is assumed that all links continuously forward the data they receive, and that all foci of the components are set to a particular default value, such as "derive all possible information". The specification is specified in LEADSTO.

The case study chosen is coordination by means of the principle of *backwards goal propagation*, which is a flexible way of specifying coordination. This approach works in a centralized fashion, having knowledge about the entire system. Backward goal propagation starts to reason from the goal to be

achieved, looks which components can achieve this goal (if the goal is not already achieved), and what specific input they need for that. Next, it is determined whether the input of these components is present, and in case it is not, other components that generate that specific information are derived. This process continues until a component is reached that can be activated (because its inputs are already present), or no such component can be found.

Note that the approach described below assumes that, for each information type, the preferred component to derive that information is known, for instance, based upon the characteristics of the components that can produce the information. A strategy would be to select the component requiring the least processing time. In LEADSTO the approach can be specified as follows (for sake of brevity, some rules, e.g. addressing setting of input/output foci of the components, are not shown):

BGP1: If at least one element of a particular information type needs to be derived according to the goal, then this is a required information type for the system to derive.

∀F:FOCUS, R:REAL, IT:INFO_TYPE
[[monitored_control_aspect_for(is_output_focus(F), SYSTEM)&
monitored_control_aspect_for(
info_type_in_focus_has_qualifier(IT, F, one_element, R), SYSTEM)]
$\rightarrow_{0,0,1,1}$ required_information_type(IT)]

BGP2: In case a certain information type is a required type, and a component C is preferred to be used to derive such information, then this component can potentially become active.

∀IT:INFO_TYPE, C:COMPONENT
[[required_information_type(IT) & preferred_component_for(C, IT)]
$\rightarrow_{0,0,1,1}$ potential_activation(C, IT)]

BGP3: In case a component has the potential to become active, and is not missing any information to derive the required output for which it is potentially active, then the component is selected to become awake.

∀C:COMPONENT, IT1:INFO_TYPE
[[potential_activation(C, IT1) &
∀IT2:INFO_TYPE [¬currently_needed_input_for_output(IT2, IT1)]]
$\rightarrow_{0,0,1,1}$ selected_control_aspect_for(awake, C)]

Note that the component also needs to be closed to input updates for all information types.

BGP4: In case a component has the potential to become active for information type IT1, and needs information type IT2 as input to derive IT1, then IT2 is required as well.

∀C:COMPONENT, IT1, IT2:INFO_TYPE
[[potential_activation(C, IT1) &
currently_needed_input_for_output(IT2, IT1)]
$\rightarrow_{0,0,1,1}$ required_information_type(IT2)]

6. Simulation Results

To assess for particular coordination approaches whether they can be expressed using the coordination language introduced in this paper, these coordination approaches can be shown to work by means of simulations (using the LEADSTO simulation tool cf. [4]), e.g., on the following *test example*:

A system consisting of three regular components C1, C2, and C3, where four information types are present, namely d1, ..., d4. Component C1 can perform two operations, namely calculate the value of information type d2 based upon the value of d1 and the value of d4 based upon the value of d1 and d3, whereas C2 and C3 can compute d3 from d2. Performance characteristics for the components (see Table 2) are as follows: C1 has an estimated processing time of 4 whereas C2 requires 10 and C3 1 time unit. All components are interconnected via links. The overall goal of the system is set to outputting an element of the information type d4 which needs to be achieved within 12 time steps after data has been received from the environment.

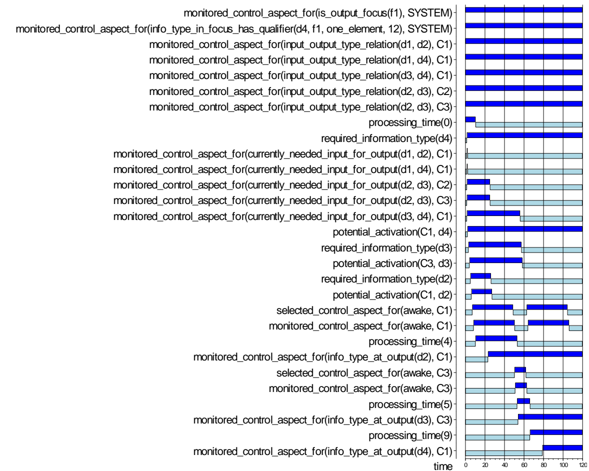

Figure 2. Partial Simulation trace

To investigate how well the coordination approach presented in Section 5 can be applied to such a test example, simulation runs have been performed. Figure 2 shows a partial trace of the behavior of the backward goal propagation approach, when applied to the test example. The left side of the figure denotes the atoms that occur during the simulation whereas the right side indicates a time line, where a black box indicates that the atom is true at that time point and a gray box indicates false. In the trace, it is initially derived that information type d4 is a required information type: required_information_type(d4). Since component C1 is the only one capable of deriving d4, it is the preferred component, and therefore it is derived that C1 can potentially become active: potential_activation(C1, d4). On the coordination level, it is monitored that C1 needs input of information type d3 in order to derive d4: monitored_control_aspect_for(currently_needed_input_for_ouput(d3, d4), C1). As a result, it is derived that d3 is also a required information type. Now there is a choice: potential activation of C2 or C3, since both can deliver information of type d3. On the system level however, a preference has been specified for component C3.

Consequently, C3 can potentially become active: potential_activation(C1, d3). This reasoning continues until a component has been found for which all of its necessary inputs are present. In our example this is component C1, which is therefore set to awake. As a result, d2 is present, which causes component C3 to become awake. Finally, after C3 has derived d3, C1 is activated again, deriving d4 and thereby causing the goal to be achieved. Note that the computation time used by this algorithm is 9: processing_time(9).

7. Evaluation

To evaluate the performance of the coordination approaches, desired properties can be specified in TTL (again using the ontology specified in Section 3) and automatically verified against the generated simulation traces using the *TTL checker* [3].

A first check that can be performed is to investigate whether the goal that has been set for the system as a whole has been reached within a deadline R:

P1: Successfulness within duration

For all time points t, if at t the system has a particular output focus and target qualifier, then a conjunction of signed info elements exists at the output that entails this target qualifier and all of the signed info elements within the conjunction have been derived by a component within a certain duration R.

```
∀t1:TIME, F:FOCUS, TQ:TARGET_QUALIFIER, QE:TQ_EXPRESSION
 [ [ state(γ, t1) |= monitored_control_aspect_for(
                         is_output_focus(F),  SYSTEM) &
    state(γ, t1) |= monitored_control_aspect_for(
                         focus_has_qualifier(F, TQ), SYSTEM) &
    state(γ, t1) |= monitored_control_aspect_for(
                         has_expression(TQ, TQE), SYSTEM) ]
 ⇒ ∃t2:TIME ≥ t1, SIEC :SIGNED_INFO_ELEMENT_CONJUCTION
   [ t2 ≤ t1 + R & entails(SIEC, TQE) &
    ∀SIE:SIGNED_INFO_ELEMENT
    [ state(γ, t2) |= is_conjunct_of(SIE, SIEC) ⇒
      state(γ, t2) |= monitored_control_aspect_for(
         information_at_output(SIE),  SYSTEM) ] ] ]
```

Besides the successfulness of a system, the efficiency of the outcome can be investigated as well. In order to determine whether the most efficient route has been found, information is needed about what is this most efficient route in the particular test example, which can be calculated for instance, using a critical path method. For the sake of brevity, this property has been omitted. Both properties were however shown to be satisfied for the trace shown in Figure 2.

8. Discussion

This paper presents a coordination specification language that allows for the specification of both pre-defined coordination approaches and more generic, flexible approaches. It has been shown how such flexible coordination approaches can be specified in this language. The approach has been applied to a test example, and evaluated using formal verification.

Besides the coordination approaches analyzed in this paper, other flexible approaches exist, such as behavior networks [9] and voting [10]. As future work it will be investigated whether these approaches can be specified in the language presented in this paper. Other coordination models, such as Linda [1], can be seen as a sort of assembly coordination language useful for implementing higher-level coordination languages [5]. It might be interesting to explore whether the language presented in this paper can also be implemented using such models.

9. References

[1] Ahuja, S., Carriero, N., and Gelernter, D., Linda and friends. *IEEE Computer*, 19(8), pp. 26–34, 1986.

[2] Bosse, T., Hoogendoorn, M., and Treur, J., Automated Evaluation of Coordination Approaches, In: Ciancarini, P. and H. Wiklicky, H. (eds.), *Proc. Coordination'06*, LNCS 4038, 2006, pp. 44-62.

[3] Bosse, T., Jonker, C.M., Meij, L. van der, Sharpanskykh, A., and Treur, J., Specification and Verification of Dynamics in Cognitive Agent Models. In: Nishida, T. (ed.), *Proc. IAT'06*. IEEE Computer Society Press, 2006, pp. 247-254.

[4] Bosse, T., Jonker, C.M., Meij, L. van der, and Treur, J., A Language and Environment for Analysis of Dynamics by SimulaTiOn. *Int. J. of AI Tools*, 16(3), 2007, pp. 435-464.

[5] Ciancarini, P., Coordination Models and Languages as Software Integrators, ACM Computing Surveys, 28(2), 1996, pp. 300-302.

[6] Galton, A. Operators vs Arguments: The Ins and Outs of Reification. *Synthese*, vol. 150, 2006, pp. 415-441.

[7] Gavrila, I.S., and Treur, J., A formal model for the dynamics of compositional reasoning systems. In: A.G. Cohn (ed.), Proc. ECAI'94, Wiley and Sons, 1994, pp. 307-311.

[8] Kowalski, R., Algorithm = Logic + Control, Communications of the ACM, Vol. 22, 1979, pp. 424 – 436.

[9] Maes, P., How to do the right thing. Connection Science, 1989. 1(3), pp. 291-323.

[10] Ordeshook, P. *Game theory and political theory: An Introduction*. Cambridge: Cambridge University Press, 1986.

[11] Selfridge, O. G., 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.

[12] Bosse, T., Hoogendoorn, M., Serban, R., Treur, J., A Coordination Specification Language for Complex Software Systems - T.R., 2007, http://www.cs.vu.nl/~tbosse/coord/tr07.pdf