Available online at www.sciencedirect.com

ScienceDirect

Advanced Engineering Informatics 20 (2006) 261–278

ADVANCED ENGINEERING INFORMATICS

www.elsevier.com/locate/aei

Emergent short-term forecasting through ant colony engineering in coordination and control systems

Paul Valckenaers [*], Hadeli, Bart Saint Germain, Paul Verstraete, Hendrik Van Brussel

K.U.Leuven – P.M.A., Celestijnenlaan 300B, B-3001 Leuven, Belgium

Received 9 January 2006; accepted 19 January 2006

Abstract

This manuscript presents a design for the emergent generation of short-term forecasts in multi-agent coordination and control systems. Food foraging behavior in ant colonies constitutes the main inspiration for the design. A key advantage is the limited exposure of the software agents in the coordination and control system. Each agent corresponds to a counterpart in the underlying system and can be developed and maintained exclusively based on knowledge about its counterpart. This approach to make non-local information available without exposing the software agents beyond their local scope is the research contribution and focus of the discussion in this paper. The research team applies this design to multi-agent manufacturing control systems and to supply network coordination systems, but its intrinsic applicability is broader.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Multi-agent systems; Manufacturing control; Coordination; Emergent forecasting

1. Introduction

An important feature of an advanced coordination and control system is the ability to plan ahead, thus avoiding problems before they occur and offering the possibility to optimize system performance. Accordingly, forecasting or predicting the behavior of the underlying system – controlled and coordinated by the control system – is a necessary precondition for the realization of this highly desirable functionality. This manuscript discusses how to implement this in an innovative manner.

The conventional manner to achieve forecasting and optimization is to incorporate a planning system in the coordination and control system. Unfortunately, such planning systems require a model of the underlying system in specific formats (e.g., a linear program), which requires frequent attention by expert personnel. These experts must both fully understand the underlying system and master the planning technology to perform the proper translation and generate the data to feed the planning system. Overall, these systems are strongly coupled and mostly centralized. This weakness has limited these planning systems to the control of manufacturing facilities with high capital investment, where the never-ending expenses of expert support are justified.

Because of the above drawback, heterarchical control systems emerged in research prototypes [1,2], and simple dispatching systems, as well as push and pull control systems appeared in industry. These systems fail to anticipate any future behavior of the underlying system but forgo the need for experts to feed the planning system with data in a proper format. This makes these solutions very commonplace today. Simplicity is their decisive advantage.

Research on multi-agent coordination and control systems aims to combine the advantages of both approaches. It aims to optimize the performance of the underlying systems without needing constant attention of the experts

[*] Corresponding author. Tel.: +32 16 322483; fax: +32 16 322987.
E-mail address: Paul.Valckenaers@mech.kuleuven.be (P. Valckenaers).

whenever changes or disturbances occur in the underlying system or its environment. Market mechanisms and negotiation have been applied [3,4] and achieved increased flexibility compared to conventional push and pull control. However, they failed to anticipate future behavior of the underlying system except in special cases. This can be explained from the fact that most research prototypes fail to construct explicit forecasts and that the coordination field in markets is one-dimensional (price versus service). In other words, they developed superior dispatching systems but failed to systematically look into the future, which typically makes regular retuning a necessity. Recent research aims to remedy this myopia through machine learning techniques [5].

This paper discusses research results that allow the coordination and control system to construct a forecast in which agents reflect relevant entities in the underlying system, and such agents can be developed solely on the basis of self-knowledge of the corresponding entity. The forecasts emerge out of the interactions of the agents. Thus, there is no need for experts that bridge the gap between planning systems and the underlying system. It suffices to have local expertise of single entities in the underlying system (e.g., conveyor belt systems) and to apply the approach that is addressed in this paper. Importantly, an agent reflecting an entity in the underlying system only needs maintenance when this entity changes, not when something changes elsewhere in the underlying system or its environment. Likewise, such agent can be employed wherever the corresponding entity is present, regardless of the variation in its surroundings. Overall, the approach remedies the weaknesses of existing approaches short of the fact that performance optimization remains an add-on. Indeed, the forecasting only handles a part of the planning task: it does not take decisions but predicts what will happen when the decisions are given, accounting for changes, disturbances and the relevant system properties.

The research results, discussed in this paper, address the synthesis of multi-agent coordination and control systems, which the research team applies to manufacturing control and supply network coordination systems. The applicability of the approach is however broader. This system engineering technology has the following intrinsic preconditions for applicability:

- The underlying system – controlled and coordinated by the multi-agent system – evolves several orders of magnitude slower than the control/coordination system. For instance, a manufacturing control system is able to transmit and process millions of messages in the time needed to transport a pallet between production units.
- The underlying system and the control/coordination system do not compete for technical resources in their environment. For instance, the control and coordination of a telecommunication network fails to comply with this precondition; both systems compete for the communication bandwidth of the network.
- There is significant economic leverage covering the efforts for implementing a better control/coordination.
- The entities in the underlying system have computationally efficient self-models.
- The coordination and control system is able to observe and direct the entities in the underlying system to an adequate extent.

Importantly, the research results in this paper have no direct contribution regarding the overall performance of a manufacturing system. The contribution is indirect analogous to the manner in which map technology – e.g., the projection method invented by Mercator – only contributes indirectly to navigation performance: navigators using maps still have to make all routing decisions. The maps help to assess the expected consequences of routing decisions and thus provide guidance for routing decisions. Both technologies fail to deliver an end-user-level solution; they only provide a partial or intermediate solution, which is fundamentally unavoidable for flexible and highly reusable subsystems [6].

Furthermore, relevant performance or success criteria for the discussion in this paper are unrelated to the performance of the manufacturing system. In analogy with cartographic technologies, key performance issues are (1) to accurately and effectively reflect reality and (2) to support compose-ability and extend-ability. As in cartography, the correctness of subsystems is verifiable and validate-able by inspecting the corresponding reality. Likewise, overlapping functionality is easily handled (navigators use whichever map suits their purposes best), and extending a subsystem (e.g., adding indications for one-way streets) only requires negligible subsystem modifications beyond the addition itself. These goals are mainly achieved and demonstrated by design, not by test results and experiments. Evidently, the proposed coordination and control technology needs to be deployed within well-performing manufacturing control systems to become successful in industry (much like cartographic technologies need to enable excellent routing in the hands of a navigator). However, this is out of scope for the research focus and discussion in this paper.

The paper first discusses the main agents in the coordination and control system, and in particular the agents' roles and responsibilities in the agent society. Next, the forecasting mechanism is presented. Finally, conclusions and future research is addressed, including a concise discussion of the iterations that have led to the current design.

2. Agents reflecting reality

A key element for achieving the above characteristics is to compose the multi-agent coordination and control system from agents corresponding to relevant entities in the underlying system – agents must not be functions. Furthermore, these agents have to reflect their section of reality while making minimal assumptions about the remainder of the underlying system; more precisely, an agent designer may only rely on invariants of the remainder of the system. This approach guarantees compose-ability and extend-ability by design. Likewise, it enables verification and validation by local inspection of the corresponding reality. The open research issue is the extent to which useful functionality can be provided without violating these principles (and how much ad hoc development work remains to be done when deploying such a coordination system).

This section discusses how the PROSA reference architecture designates which parts of the underlying reality these agents reflect. Moreover, it discusses how PROSA contributes to the above-mentioned performance and success criteria.

2.1. PROSA agents

The reference architecture acronym PROSA [7] denotes *product-resource-order-staff architecture* and refers to the composing types of agents. The structure of systems designed along the PROSA architecture is composed of three types of basic agents: order agents, product agents, and resource agents. These basic agents are structured using object-oriented concepts like aggregation and specialization. Staff agents, like in human organizations, can be added to assist the basic agents with expert knowledge.

The basic PROSA agents, which are actually agent types, divide the underlying world, which is being coordinated and controlled, in three aspects. The motivation for this subdivision is that these aspects are comparatively decoupled, and that the subdivision increases the user community (re-usability) of the agent software that is developed along these lines. The three aspects are:

- *What*: resources in the underlying world;
- *How*: recipes (manners in which resources may be used);
- *When-and-where*: ongoing tasks that execute a recipe on resources.

This subdivision is reflected in a PROSA coordination and control system by, respectively, the resource, product and order agents. These agents are discussed below.

A resource agent corresponds to a resource in the underlying system. It controls its resource but also reflects the resource to the remainder of the multi-agent coordination and control system. The overall coordination and control system issues commands to a resource through its resource agent, but also acquires information about this resource through this agent. This information can be both static (e.g., technical specifications of equipment) and dynamic (e.g., current machine status). This includes information about the present, the past, and even about the future (see Section 3). Importantly, deep and detailed knowledge about the resource itself in combination with general knowledge about the application environment/domain must suffice to implement and maintain the resource agent software. Furthermore, this software must be reusable wherever the corresponding resource is installed. Table 1 lists examples of resource agents within application domains that qualify against the intrinsic preconditions in Section 1.

A product agent corresponds to a product model/type in the underlying world (e.g., Ford Mondeo or Opel Astra). The product agent holds the knowledge on how to create instances of its type (i.e., recipe in a broad sense). A product agent has however no knowledge about the product instances that are created; it does not know how many instances have been created, nor how many instances are in the process of being created and in what state these instances are.

A product agent holds the process and product knowledge to assure the correct making of its product with sufficient quality. The product agent comprises functionalities that are traditionally covered by product design, process planning, and quality assurance. A product agent knows what the known permitted sequences of operations on suitable resources are to create product instances. In its most primitive implementation, a product agent is a list of operations to be executed on known resources. More realistically, a product agent will be an information system in which a team of human experts is integrated, where these experts are able to expand the collection of known and valid ways to produce a certain product in function of resource availability and other concerns.

Table 1
Sample resource agents (corresponding part in the underlying system)

Manufacturing	Supply networks	Traffic	Electricity
Material receiving station	Raw material supplier	Airport	Generator based on wind energy
Drilling workstation	Manufacturing plant	Roundabout	CHP generator
Thermal treatment workstation	Re-packaging facility	Road segment	Fast-response generator (gas)
Packaging station	Wholesale distributor	Harbor	Nuclear plant
Shipment station	Shops/stores	Home (garage)	Hydro-electrical storage
Automatic storage and retrieval system	Distribution center	Car parking lot	Climate control system (consumer)
FIFO pallet buffer	Truck fleet	Work place	Transmission line
Part buffer	Truck	Railway station	Distribution unit
Overhead crane	Aircraft	Car	
Conveyor belt	Container	Train	
AGV fleet		Ship	

The main purpose of having product agents in a PROSA implementation is to shield the order agents from technological concerns. Conversely, order agents shield the product agents from resource allocation and logistic concerns. Product agents are unaware of product instances. Product agents are unaware of the availability of processing capacity on resources. Conversely, order agents always query their product agent to know what the permissible processing operations are; they do not possess the technical know-how to decide by themselves.

Also, note that the term product is to be understood in a broad sense. For instance, there typically will be a product agent that knows how to perform planned maintenance on the underlying manufacturing system. Indeed, this product agent knows which actions/operations have to be executed on which resources to perform a maintenance cycle. Stronger, there will be a product agent for each type of maintenance (light/heavy, planned/unplanned, etc.). Notice how the PROSA architecture minimizes the need for special services/concepts/solutions by making the basic concepts generic.

Table 2 lists examples of product agents in within application domains that qualify against the intrinsic preconditions and one 'toy' domain (gastronomy). The toy domain example illustrates how an advanced recipe leaves the order agent more options. If the delivery of eggs is delayed, 'production' can still start. In this toy example, the flexibility of the recipe is mostly irrelevant. The analogy in manufacturing and supply networks does benefit significantly from such non-deterministic process plans, particularly when the recipe is very detailed to serve for automated systems and the degree of over-specification becomes very high. Notice that a better/specific name choice for 'product agent' may be indicated for the non-manufacturing application domains.

An order agent corresponds to a task that needs to be executed. The agent is responsible for performing the assigned work correctly and on time, where it delegates all technical issues to the associated product agent. The order agent is a manager; this agent handles the logistics and the resource allocations needed to execute a valid sequence of operations to get its product instance produced. The order agent interacts with the product agent to know what valid sequences are available. The order agent manages the product state model, which captures the state of the product instance that is being produced. An order agent may represent customer orders, make-to-stock orders, prototype-making orders, rush orders, orders to maintain and repair resources, etc. In simple implementations, an order agent can be associated with work piece as it travels through the factory. Table 3 lists examples of order agents in application domains that qualify against the intrinsic preconditions (see Section 1).

2.2. PROSA agent interactions

PROSA agents use delegation to restrict their exposure. Product agents never accept to manage state information about ongoing tasks. They receive an object representing the state of a task from the corresponding order agent

Table 2
Sample product agents

Domain	Product type	Recipe or product agent description
Gastronomy	Pancake batter (primitive)	Procure the following ingredients: 4 eggs, 1/2 liter of milk, 100 grams of vanilla sugar, 400 grams of flour. Mix the flour sugar and eggs. Finally, add the milk and mix
	Pancake batter (advanced)	Procure ingredients according to the following ratios in function of the quantity required by the customer: ... Mix the ingredients in any sequence but take care that the flour gets mixed/humidified with a limited amount of liquid (egg or milk) such that it becomes a viscous mass before adding the milk in full
Manufacturing	Heat treatment of gear wheels	The product agent knows which furnaces are suited and which temperature trajectory is required for the heat treatment in each of the candidate furnaces
	Heat treatment of gear wheels (advanced)	In addition to the above, the product agent knows the set of allowed temperature trajectories, creating the opportunity to batch product instances of its type with product instances of compatible types
Supply networks	Heat treatment of gear wheels	The product agent knows which manufacturing plants offer the required processing facilities and knows how to route its product instance through the supply network such that it visits a suitable plant and receives its heat treatment. Inside the plant, the above product agent takes over. The product agent knows which containers are needed for transporting the product instance...
Traffic	Working days	The product agent supports an agenda system for working days in which the user can specify the required journeys. The product agent knows are permitted ways to execute these journeys. It knows about resource requirements (e.g., a car with sufficient room for the children, groceries, parking space, slots on road segments, etc.). An advanced agent knows about margins on the requirements
Electricity	Climate control for offices	The product agent knows how to cool or heat an office complex in function of the available electricity supply. It knows what are suitable energy sources and required capacities on the electricity transport and distribution systems to get the energy from the source to the office complex. An advanced product agent will use its own margins to account for the availability of the supply

Table 3
Sample order agents

Domain	Task	Remarks
Manufacturing	Customer order no. 332	Tracks and manages a specific customer order with the assistance of the product agent within the plant as a rush order
Supply network	Customer order no. 556	Tracks and manages a specific customer order with the assistance of the associated product agent within a supply network
Traffic	John's working week	The order agent executes the 'working days' recipe for John's agenda; perhaps using whatever has been learned about John's preferences, predictability... The order agent owns the agenda instance with John's data; the product agent only provides the agenda template from which initial empty agenda instances are created
Electricity	Climate control in the Stella Artois HQ	The order agent executes the climate control for offices recipe for a specific office complex, using available information about its current status, experience about the habits of its occupants, weather forecasts for the location...

whenever they need this information to perform their tasks. Likewise, order agents always consult product agents about possible ways to proceed with their task. A typical interaction amongst the basic agents goes as follows:

- When a new task enters the system, a new order agent is created with a link to the appropriate product agent.
- The order agent retrieves an object that represents its initial state from the product agent. This state object basically indicates that nothing has been done.
- The order agent passes his state object to the product agent as a parameter in a query about possible next processing steps. The product agent sends a list of possible next steps to the order agent.
- The order agent searches and selects a combination of a next processing step and a suitable resource agent to execute this step. Section 3 discusses in more detail how the order agent searches and selects.
- When the selected processing step is executed on the selected resource, the resource agent returns an object that represents the processing result to the order agent.
- The order agent passes its state object, selected processing step identifier object and the result object to the product agent in a query to receive an updated state object.
- Until the task is fully executed, the order agent repeats this procedure from the third bullet point on.

Note how the order agent needs no understanding of what is inside the state object or result object, and only uses processing step identifier objects to query resource agents about their ability to perform this step. If the internals of these objects change, the order agent keeps on functioning without modification. Replacing a primitive product agent (knowing a fixed list of operations) by a sophisticated aggregated product agent (comprising a team of human experts) does not require any software maintenance for the order agent. Likewise, the product agent has no notion of the execution state of the tasks or the configuration of the resources. For instance, the same product agent software performs properly in a job shop or a flow shop or in any other manufacturing topology without modification. Similarly, the resource agents are not exposed to the product types and order portfolio of the manufacturing system.

These boundaries on the exposure of the agents that compose a PROSA implementation yield high reusability of the software, high flexibility and configure-ability. Indeed, each agent performs his duty as long as nothing changes within its exposure, which is exactly what is minimized by the architecture. Concerning the performance and success criteria relevant for this paper, PROSA achieves its objectives through design.

2.3. PROSA agent structuring

PROSA adopts the structuring mechanisms from object-oriented programming: aggregation and specialization. In addition, it supports staff agents that provide advice to the basic agents without assuming final responsibility. This restriction to advice-only is the key to safeguarding performance for staff agents concerning the criteria in this paper. A further discussion of staff agents is outside the scope of this manuscript.

Aggregated agents are sets of related agents that are clustered together. They form bigger agents with their own identity. An agent may belong to several aggregations and aggregated agents can dynamically change their contents.

For instance, membership of a batch order agent may depend on the timely arrival of the prospective members. Aggregated agents may emerge out of the self-organizing interaction of agents or they may be designed up front.

The number of aggregation levels depends on the specific needs of a certain system, and is not dictated by the architecture.

Aggregation reduces exposure of the individual agents and increases software reuse. Consider a shopping list order agent and a corresponding product agent. This shopping list order agent simply creates new order agents for each item on the shopping list and manages the aggregate. The production of the individual items is delegated to the agents corresponding to the items. The shopping list agent manages however the due dates. For instance, when one item suffers a delay, the aggregated agent serves as the reference point for the item order agents and provides them with suitably adapted individual due dates. The shopping list product agent knows which combination of items can be shipped and invoiced to the customer.

Specialization allows a layered implementation of the agents, where the higher layers can be shared and, more importantly, used as interfaces to the remainder of the coordination and control system. In object-oriented programming, this is known as inheritance of subclasses from superclasses. The superclass is more abstract (polygon) whereas the subclass is more concrete (triangle).

For instance, in a manufacturing control system there will be a specialization hierarchy starting from agent along resource agent, transport resource agent, conveyor belt agent toward 'conveyor belt type XYZ model IV' agent. Obviously, each subclass implementation profits from reusing the superclass software. More importantly, any agent working with resource agents will perform its duties regardless which subclass of resource agents it is interacting with. When an order agent expects to interact with a resource agent in the abstract/superclass sense, it will work perfectly with any subclass. Specialization mainly increases the reusability of the agents interacting with the various subclasses that will be present in the control system.

2.4. Discussion

PROSA preserves the attractive properties encountered in cartography (the analogy mentioned in Section 1). PROSA agents reflect local parts of a reality that is integrated, coherent and consistent, which greatly facilitates achieving the same – integration, coherency and consistency – in the coordination and control system. Moreover, PROSA subdivides the system in software components (agents) in a manner that maximizes both the size and diversity of the potential user communities of these components. This maximizes the chances of success and survival for these software components [8–10].

Although PROSA makes no explicit contribution to the forecasting, it is an essential element in the emergent nature of the forecasting technique. An explicit and separate subsystem to generate forecasts is only desirable if it remedies the drawbacks of existing alternatives (e.g., of explicit centralized planning). The attractive properties of a PROSA design, which it shares with cartography, constitute an opportunity in this respect.

To capitalize on this opportunity, the forecasting functionality needs to be added without compromising the PROSA advantages. In other words, the scope limitations imposed on PROSA agents must be preserved whereas non-local system properties need to become available at the relevant locations. In cartography, placing road signs on a map to places outside the map, without requiring map builders to know or monitor the world beyond the immediate neighborhood reflected by their map, would be an example. To successfully reconcile these two concerns, a forecasting technique must make non-local and global information emerge. Indeed, the system-wide aspects must not be reflected within individual agents or software components but still must be available within the coordination and control system to support effective decision-making.

3. Emergent forecasting

The previous section provides the terminology to discuss the agents in the multi-agent coordination and control system. It is an application of the essential modeling approach in state-of-the-art object-oriented design. Imagine essential modeling as a generalization of the concept of developing maps when solving navigation problems. Furthermore, active agents, instead of passive objects, model the active entities in the underlying world in the control systems discussed in this paper, resulting in a better match. This section discusses how the PROSA agents are rendered more intelligent in a manner that produces short-term forecasts for the behavior of the underlying system. This section first presents the source of inspiration for the design: food foraging in ant colonies. Next, it presents the emergent forecasting design. Appendix 1 contains formal definitions of key agent behaviors and properties.

3.1. Ant colonies

Food foraging ants execute a simple procedure (the NetLogo programming tool contains a nice simulation; see http://ccl.northwestern.edu/netlogo):

- In absence of any signs in the environment, ants perform a randomized search for food.
- When an ant discovers a food source, it drops a smelling substance – a pheromone – on its way back to the nest while carrying some of the food. This pheromone trail evaporates if no other ant deposits fresh pheromone.
- When an ant senses a pheromone trail it will be urged by its instinct to follow this trail to the food source. When the ant finds the food source, it will return with food, while depositing pheromone itself. When the ant discovers that the food source is exhausted, it starts a randomized search for food and the trail disappears because of the evaporation.

These simple behavior patterns result in an emergent behavior of the ant colony that is highly ordered and effective at foraging food while being robust against the uncertainty and the complexity of the environment. An important capability of this type of stigmergy is illustrated: global information (where to find food in a remote location) is made available locally (in which direction must the ant move to get to this food). Also, the complexity of the environment is handled by making the environment part of the solution, effectively shielding the ant colony solution from this complexity.

For the design and development of coordination and control systems, the following principles are recognized:

- Make the environment part of the solution to handle a complex environment without being exposed to its complexity. This complies with the essential modeling approach of object-oriented design.
- Place relevant information as signs in this environment ensuring that locally available data informs about remote system properties, supporting system-wide coordination.
- Limit the lifetime of this information (evaporation) and refresh the information as long as it remains valid. This allows the system to cope with changes and disturbances.

These engineering principles are used to design an emergent forecasting multi-agent schema for manufacturing control systems and other application areas complying with the preconditions stated in the introduction of this paper.

Note that these principles must also be applied elsewhere in the control system. Recall the shopping list order agent and the item order agents. Item agents normally query the list agent about their due date. When an item agent performs a calculation in which this querying would be prohibitively expensive, the item agent remembers the due date provided by the list agent and reuses this stored value. In such cases, the stored value must have a limited lifespan and be refreshed when necessary. Forget-and-refresh is a basic mechanism to handle dynamics in systems, which must comply with the first precondition for applicability for it to be effective.

3.2. Environment augmentation

The basic PROSA design needs to be augmented in three ways to support agents that implement ant colony inspired coordination. These agents are called ant agents in the remainder of this paper. First, ant agents must be able to navigate virtually through the underlying system. Second, the ant agents must be able to deposit, observe and modify information in this virtual environment reflecting the underlying system. Third, the ant agents must be able to interact in a what-if mode with the other agents in the coordination and control system. Each of these features is discussed below.

3.2.1. Navigation

The ant agents cannot navigate through the underlying system that is being controlled, in contrast to related work on ant colony coordination in telecommunications [11]. Therefore, the multi-agent control system provides a reflection of the underlying system by means of the resource agents. In current implementations, the resource agents implement a navigation graph on which the ant agents can travel virtually through the underlying system. Because this happens in cyberspace, ant agents can travel up and down the underlying system much faster than in reality.

Each resource agent has a finite set of exits and entries, where entries are connected to exits and vice versa. Thus each resource agent only has a local view on the topology of the underlying system. Nonetheless, this suffices for the ant agents to navigate virtually through the entire system. A graph model is only one of many options. For other coordination and control tasks or underlying systems, geometrical two-dimensional or three-dimensional self-modeling functionality may be necessary. The graph model has been more than adequate for the ongoing research.

Practically, each resource agent is able to provide its visiting agents with the list of its entries or exits, where each entry knows the exit to which it is connected and each exit knows the connected entry. Each exit or entry also knows its resource. This suffices to virtually travel through the underlying system. In addition, resource agents normally possess further self-knowledge such as traveling times or processing times, and they keep their status information synchronized with the actual resource state (e.g., when it breaks down, they immediately reflect this).

3.2.2. Information spaces

PROSA agents and in particular resource agents provide the ant agents with information spaces. These spaces are like blackboards attached to the (resource) agents or entries and exits of the resource agents. Ant agents can drop, observe, modify and delete any serialize-able software object in these spaces. The objects in such information space have a limited lifetime and disappear after a finite amount of time (note: lazy evaluation keeps the implementation efficient).

The ant agents must refresh information at a frequency above the evaporation rate if they want to keep it available. Note that higher refresh and evaporation rates ensure shorter delays between changes in the system and observation of these changes by the agents. Higher frequencies require more communication and computing power however. Ongoing developments did not discover the need for a signal strength parameter as in the insect world. Information simply is either valid or out-dated and inaccessible.

3.2.3. What-if mode

The PROSA systems have an important edge over the ant colonies: their resource agents can be equipped with intelligence whereas the environment of the real ants (soil, vegetation) is passive and non-cooperative. Indeed, a conveyor belt agent can be designed to be knowledgeable on how this belt will behave in a given situation without the need to actually bring the conveyor belt in that situation. The resource agent can be given the ability to operate in a what-if mode (in addition to and in parallel with the normal operating mode). Note that this mode still only requires self-knowledge, although at a higher level of intelligence, about the corresponding entity in the underlying system.

In the PROSA systems discussed in this paper, a conveyor belt agent not only knows its status (switched on, speed setting, identity of the connected resources, identity of items on top, etc.), but the agent also can give informed answers about an anticipated but still fictitious future. For instance, the resource agent can answer questions like "if a pallet were to arrive in half an hour, at what time will it reach the other side of the belt?" Details on how the resource agent is able to give an informed estimate are given below. In short, ant agents inform the resource agent about the intentions of prospective users, from which the resource agent computes its expected workload that is used to compute the estimates answering these what-if queries.

3.3. Exploring ant agents

Every order agent generates explorer ant agents at a given frequency. These explorer agents are scouts that each search for an attractive route through the underlying system that accomplishes the given task (Fig. 1). Depending on the performance criterion, these explorer agents search forward from the current state of the task on (e.g., lead time minimization) or backwards from the final delivery point (e.g., due date accuracy). Note that different order agents can have different performance concerns; rush orders, normal orders, low priority orders, maintenance orders have different objectives. The objective of a given order may even suddenly change (e.g., when a work piece gets damaged and needs a speedy replacement).

These scouts use the same method as the order agent, managing the actual execution of the task, to ensure that a proper sequence of processing steps gets executed, but virtually. How such feasibility concern is handled is outside the scope of the present paper. For the present discussion, it suffices to know that another kind of ant agent – feasibility ant agents – deposits information on the information spaces attached to entries and exits of the resources that allows the product agents to discern valid and invalid local routings. This information also evaporates and is refreshed to account for changes in the underlying production system.

The search strategy employed by the explorer agents is a parameter or plug-in of the control system. Not every explorer agent uses the same strategy. Typically, some percentage looks for the promising routes whilst other ant agents look for solutions that aim to avoid critical resources. The key point is that the emergent forecasting does not rely on which strategy is employed by these scouting agents.

On its exploration journey, the ant agent delegates the information processing to the product agent and resource agents. The product agent provides the set of feasible routing options that are open to the scout at each routing point. It makes sure that the product recipe is obeyed. The resource agents provide the necessary performance estimates.

Consider an explorer agent moving forward from the current position of a work piece on a conveyor belt. The scout queries the conveyor belt agent about the estimated remaining traveling time on the belt. The explorer agent than virtually travels to the exit of the belt and adjust its internal clock indicating the expected arrival time. At this point, the scout selects an entry connected to the exit of the conveyor belt and continues his virtual journey. When

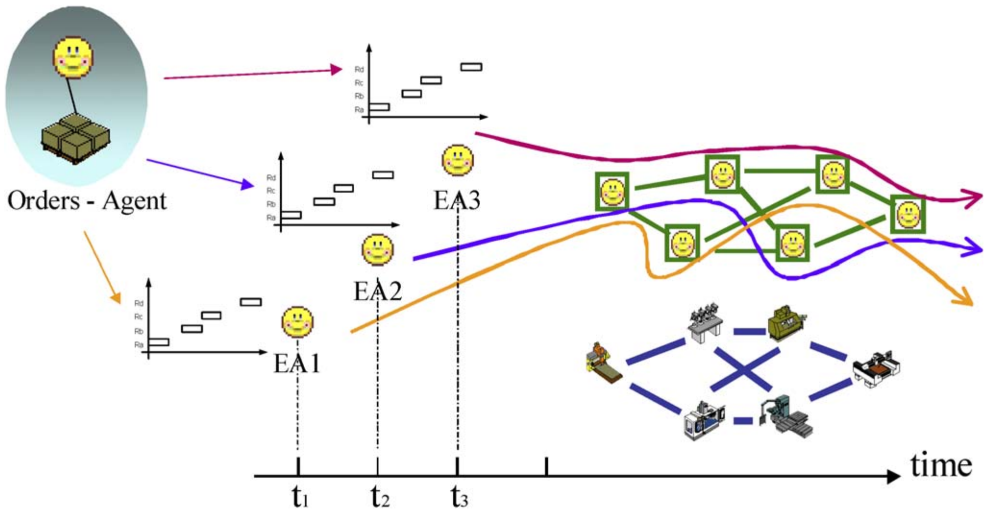


Fig. 1. Ant agents explore possible routes.

the ant agent arrives at a processing unit (e.g., furnace), the product agent indicates the possible processing steps and the explorer agent selects a processing step. The explorer agent queries the resource agent of the processing unit about estimated queuing and processing times as well as processing results. In this manner, the virtual journey continues until the final delivery point is reached. Note that the resource agents will virtually execute their own execution strategies. The emergent forecasting mechanism does not require any specific strategy from these resources. Again, a strategy is a plug-in for the control system.

When an explorer ant agent has virtually executed the task, it reports back to the order agent. The report includes the journey and the performance estimates of that journey. Based on the results of its exploring agents, the order agent keeps a set of candidate routes. These candidates get refreshed regularly, either explicitly by specialized exploring ants that simply follow a given route, or by ensuring that the normal exploring ants will rediscover these currently attractive candidates with a high probability. The set of candidate routes is selected based on the performance estimates and on their complementary nature (i.e., limit the number of candidates that have very similar routings). The set of candidates evaporates candidates that have become too old. Basically, the exploring ants implement a distributed heuristic search for good production schedules and adapt the solution continuously to account for changes and disturbances. The optimization heuristic itself is not the focus of the research.

3.4. Intention ant agents

The above exploration requires the resource agents to possess an adequate estimate of their future workload. The order agents generate intention ant agents, at a given frequency, to serve this purpose. When a suitable set of candidate solutions has been constructed (see above) and the estimated starting time for the processing of the product instance(s) approaches, the order agent selects one of the candidate solutions to become its intention. Then, the order agent generates intention ant agents to notify the agents of the affected resources of its intentions (Fig. 2).

The intention notification service operates as follows:

- The intention ants virtually execute the routing and processing of their selected candidate solution. On their virtual journey, the intention ants acquire travel, queuing, and processing times from the resource agents on their path. Any changes, which occurred since the exploration, immediately become visible when these resource agents provide the information.
- In contrast to the exploring ants, intention ants inform the resource agent that their order agent is likely to visit them at the estimated time. In this way, intention agents make a (evaporating) booking on the resource, and the resource agent adapts its load forecast (local schedule of the resource) to account for the visit of which it is informed by an intention ant. In other words, the intention ants enable an emergent forecasting of the resource utilization. As a consequence, resource agents are able to predict performance more accurately to their visitors, the exploring and the intention ants.
- The intention information at the resource agent evaporates. Order agents must create intention agents at a refresh frequency that is sufficiently high to maintain their bookings at the resources.
- While refreshing, the order agents observe the evolution of the expected performance of their current intentions (through intention ants reporting their estimated performance). This performance estimate is compared to the estimates of the candidate solutions that are found and refreshed by the exploring agents.

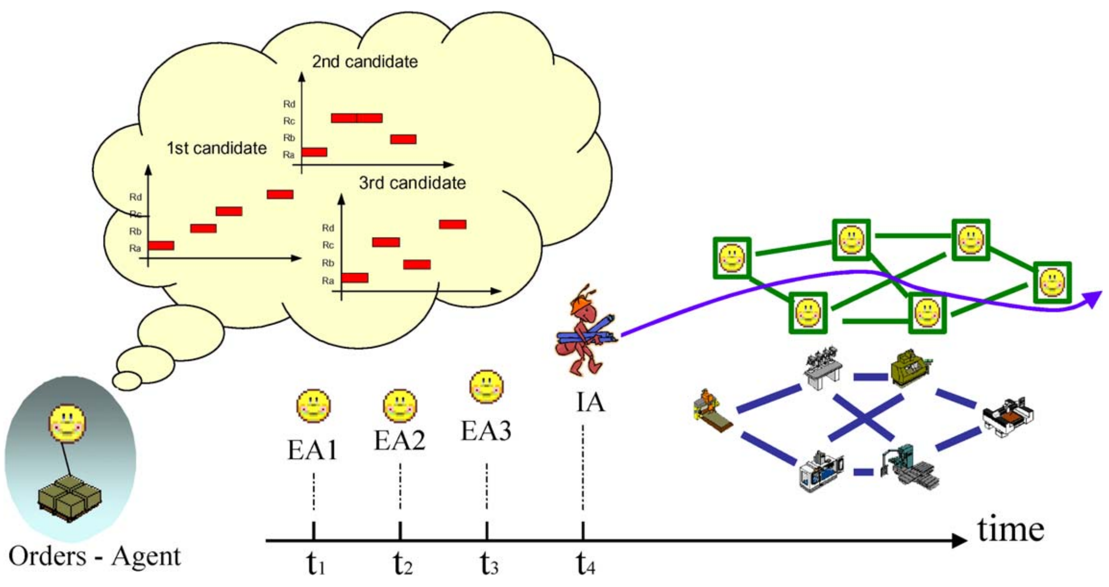


Fig. 2. Ant agent propagates the order intentions.

- When the estimated performance of the current intentions drops significantly below the estimated performance of some candidate solutions, an order agent may change its intentions.
- When the product instance(s) of an order agent reaches the point where a decision needs to be executed, the order agent triggers the action in the underlying system in accordance with the intentions. If an event occurred that makes this impossible or highly unattractive, the order agent delays the action shortly such that the above procedure may find a better solution.

The above enables the coordination and control system to generate short-term forecasts of the system behavior both for resources and orders. It thus suffices to combine software agents that are developed on the basis of local knowledge only, including a what-if self-model, to generate these forecasts that reflect how the system is likely to behave as a whole. There is however one caveat: the order agents must behave in a socially acceptable manner as is discussed below.

3.5. Responsiveness versus forecast reliability

To have useful forecasts, the control system designer must ensure that the order agents have a 'socially acceptable behavior'. The proper design and tuning of suitable social control mechanisms is addressed in [12] and is subject of ongoing research. Such socially acceptable behavior means that agents do not change their intentions too easily; otherwise, minor disturbances like a very short breakdown of a resource may create an avalanche of orders that shift to alternative resources.

Thus, order agents refrain from changing their intentions to easily and too frequently. The perceived improvement must be higher than a threshold value before the current intention is replaced by the more-promising alternative. In addition, order agents limit the frequency at which they may change their intentions. Furthermore, a typical control system design favors the changing of intentions for actions that are further in the future over last-minute changes, as is the case in socially acceptable human behavior. Note that different order agents may have different thresholds and frequency limits (rush order agents are more responsive/nervous than unsold make-to-stock order agents).

Moreover, a proper distributed design randomizes the behavior of the agents therein. The threshold values will be randomized; the frequency limits will be randomized, etc. This avoids that several agents, independently, decide to switch to the same resource/route/etc. Likewise, order agents change their intentions probabilistically such that within a single refresh cycle only a small fraction of order agents may change their intentions, and the other agents are able to observe the consequences before changing their intentions as well. With such a design, order agents will gradually shift toward alternative routes when a disturbance occurs until a new equilibrium is reached. Note how this last mechanism only works well if there are sufficient refresh cycles between events in the physical underlying system – recall the first precondition for applicability.

3.6. Remarks

The above describes how order agents create two types of ant agents that, together with the basic PROSA agents, enable the construction of short-term forecasts that account for changes and disturbance as they occur. As a result, today's unpredictable and turbulent environment, characterized by increasingly frequent changes and disturbances, is the normal operating condition for this control system design.

This paper describes the basic design, which normally incorporates additional mechanisms. For instance, two order agents may try to swap resources when changing intentions to reduce the disturbance to the remainder of the system. Likewise, ant colony inspired mechanisms can be added to facilitate self-organizing batch building or minimize changeovers. The basis for such functionality is to have resource agents (with batch processes or sequence-dependent processing times) create their own kind of intention ants. These ant agents propagate the resource utilization forecast information through the virtual factory while collecting travel time information en route. This information creates a 'large target' for exploring ants to discover opportunities to join a suitable batch or to get processed without triggering a changeover. A detailed discussion is outside the scope of this paper.

More generically, the ant agents may spread probabilistic information in addition to the forecasting of the most expected behavior. Furthermore, planners, serving as staff agents in the PROSA architecture, can easily enhance the basic design. These planners may guide the explorer agents: routes advised by the planners may be explored first and/or more frequently. Likewise, the route/trajectory advised by the planner may become the order agent's first intention and benefits from the agent's tendency to stick to its intentions.

4. Conclusion

The above describes a design that emerged after a number of iterations. These are discussed concisely. Next, the achievements of the current design are discussed. Finally, future research activities are addressed.

4.1. Development history

The coordination and control technology presented in this paper has been developed in a number of iterations. The first design consisted of a feedback design, in which actual order agents were creating ant agents to travel backward reporting their success/failure in actual production. For instance, an order agent in a congested situation would discourage (similar) orders from following the same route. This first design was not anticipating and was only reporting congestions after they already occurred. This avoids congestions from propagating too far backward, but does not prevent them from occurring. It also creates a tuning issue to make sure that the system discovers sufficiently fast that the congestion has disappeared.

The second design already had the intention-propagating ant agents allowing resource agents to build a local short-term forecast [13]. These local forecasts were propagated backward through the system, during which the estimated travel time to the resource was accumulated. The intention ants used this information to select optimized routes. This works fine in the immediate neighborhood of the resource, but the estimation of the travel time becomes problematic when back-propagating the resource load forecasts through storage systems and processing units. This design was abandoned in favor of the more elegant solution described in this paper. It remained however a valid mechanism for augmenting the performance of the explorer ant agents, where the load information is only propagated as far as the travel time estimate can be calculated accurately.

Indeed, this mechanism – called resource intention propagation as opposed to order intention propagation discussed above – already resurfaced to support exploring ants in discovering batching opportunities. When exploring ants virtually enter the neighborhood covered by the resource intention propagation (performed by another kind of ant agents), they locally perceive the existence or absence batching opportunities. In other words, the target for the scouting agents becomes a lot bigger. Similarly, opportunities to avoid changeovers and setups become visible in a much wider area.

4.2. Current design

The current design represents a stable point in the iterative development described above. The individual agents are able to delegate any information processing from which they need to be shielded, and the need for a common understanding of complicated data structures is minimized.

The PROSA architecture allows the design to be composed of agents that reflect parts of the underlying reality. Since these parts fit together in that reality, the integration issues will be restricted to cosmetic problems.

The ant colony inspired design succeeds in preserving this exposure-limiting design one step further. The system is able to produce short-term forecasts, both for resources and orders, without exposing the individual agents to system characteristics outside their own local scope.

The actual decision-taking mechanisms cannot be included in this design. Since this is an NP-hard problem, any known efficient solution is an approximation whose performance depends on the prevailing situation; if conditions change, the approximated solution typically needs retuning. Exposure is unavoidable short of a technology breakthrough that solves NP-hard problems over a wider operating range.

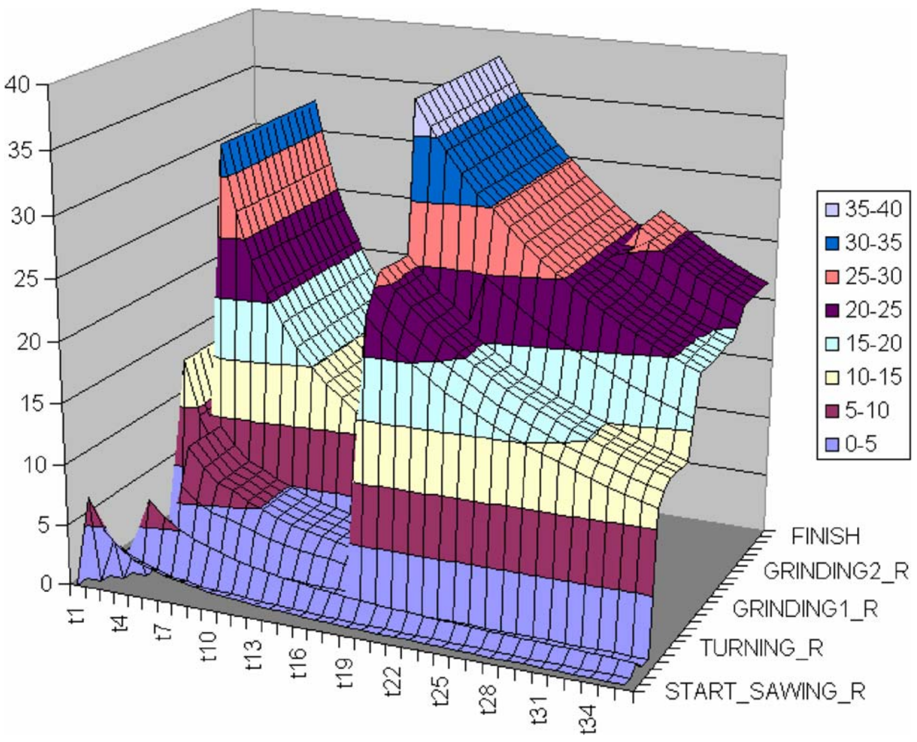


Fig. 3. Error plot of intention data of an order.

4.3. Future work

Ongoing developments address the implementation, the responsiveness issue and the concept of emergence and self-organization itself.

The implementation efforts aim to make the experimental platform widely available. It is targeted to become the engine behind the web-based benchmarking service (www.mech.kuleuven.be/benchmarking) of the IMS Network of Excellence (www.ims-noe-org). The main goal of this activity is to establish a benchmarking service for manufacturing control systems that goes beyond the usual toy case testing in current research. Appendix 2 presents results of a sample implementation.

The responsiveness versus forecast accuracy issue is a main research focus. Fig. 3 shows some preliminary results on the forecasting accuracy. The vertical axis represents the forecasting error. The horizontal axes represent time and processing steps, respectively. The short-term accuracy is high until a serious disturbance occurs (machine breakdown or rush order arrival). After a disturbance, a number of refresh cycles is required to restore the accuracy. In these early results, the forecasting horizon is also limited by the disturbance created from new order releases into the system. Since these new orders are known much longer on beforehand, releasing their intentions earlier will improve the forecast accuracy further.

Finally, the research team addresses the issue of what constitutes emergence and/or self-organization and its contribution to the development of better coordination and control systems. The research focuses on understanding how self-organization and emergence support limiting the exposure of software components/agents.

Acknowledgements

This paper presents work funded by the Research Fund of the K.U.Leuven (GOA – AgCo$_2$) and the European Commission.

Appendix 1. Formal agent models

This section presents formal definitions for key agents in the emergent forecasting design. The models use pseudo-code (i.e., they are so-called operational specifications) reflecting the fact that the design consists of communicating sequential processes. Note that, since the coordination system handles a going concern, no closed-formula objective function exists and it is impossible to treat the design as an optimization problem. Furthermore, work on Petri-net based specifications is ongoing but produces models that are less self-explaining due to the low-level nature of Petri-nets. The developments aim to parameterize the agents by Petri-nets (www.renew.de).

The exploring and intention ant agents, created by order agents, are the most prominent players in the emergent forecasting mechanism. This section start from these ant agent types and elaborates other agents as they interact with this principal agent type. The agent specifications only address the generic and reusable parts of the forecasting mechanism. The add-ons are not addressed for obvious reasons (i.e., the DecisionMaker and ResourceScheduler classes). The remainder of the PROSA system is out of scope. The pseudo-code is based on Java and adds basic mathematical concepts such as sets where indicated. The pseudo-code makes abstraction of visibility scopes (such as indicated by public, protected, and private in Java).

```
agentClass ExploringAnt extends AntAgent {
  // Attributes
  ResourceAgent resource; DecisionMaker decider;
  OrderAgent creator; ProductAgent product;
  PartialSolution solution;
    // solution contains history until virtual current time
    // including all locations, states, operations... done so far.
    // lastResource, lastState, lastTime attributes represent
    // the virtual situation of the explorer ant.


  // Constructor
  ExploringAnt (DecisionMaker d, PartialSolution s) {
    decider = d; solution = s; resource = solution.lastResource;
    creator = solution.order; product = creator.product;
    // agent behaviors become active at this point
  }

  // Active behaviors (threads)
  agentBehavior explore() {
```

```
  resource.virtualExecute(solution); resource = solution.lastResource;
    // Resource agent virtually performs ongoing operation
    // which may be processing, transport, no-op, etc.
    // solution is updated as if operation was performed.
  while (product.incomplete(solution)) {
  ActionSet options = product.possibleNextOps(solution);
    // product agent calculates the set of possible
    // next actions, including performing processing operations
    // at the current location, moving to a neighboring
    // location/resource, and waiting. Notice how exploring ants
    // dont distinguish productive actions (processing) from
    // storage, handling, transporting... It suffices that the
    // decision maker knows this distinction.

  if options.empty() { // no solution possible
    solution.failed(); // update solution to reflect failure
    break; // jump out of while loop
  }

  decider.selectAction (solution, options);
    // decider selects an action, member from options
    // and opdates solution to reflect this choice
  resource.virtualExecute (solution);
    // resource agent virtually performs selected operation
  if solution.failed() break;
    // ResourceScheduler failed to find a valid solution

  resource = solution.lastResource;
    // if action was transport, the virtual location of the
    // exploring ant will be changed.

  } // while
    creator.report(solution); // inform order agent of solution found
    decider.wrapUp(solution); // decision maker may use solution for ...
    this.terminate(); // explorer agent ceases to exist
  }
}

agentClass IntentionAnt extends AntAgent {
  // Attributes
  ResourceAgent resource; PartialSolution solution;
  ActionSequence intention; Action nextOp;
  OrderAgent creator; ProductAgent product;

  // Constructor
  IntentionAnt (PartialSolution s, ActionSequence i) {
    solution = s; intention = i;
    resource = solution.lastResource;
    creator = solution.order; product = creator.product;
    // agent behaviors become active at this point
  }
  // Active behaviors (threads)
  agentBehavior propagateIntention() {
    Iterator i = intention.getIterator();
    resource.virtualExecute(solution);
      // Resource agent virtually performs ongoing operation
      // which may be processing, transport, no-op, etc.
      // solution is updated as if operation was performed.
  while (product.incomplete(solution)) {
    nextOp = i.next();
    if product.notPossibleNextOp(nextOp, solution) {
      solution.failed(); // update solution to reflect failure
      break; // jump out of while loop
    }
    solution.selectNextAction(nextOp); // update solution
    resource.virtualExecuteAndReserveCapacity(solution);
      // resource agent virtually performs selected operation
      // and reserves the required processing capacity
    if solution.failed() break;
      // ResourceScheduler failed to find a valid solution
    resource = solution.lastResource;
  } // while
    creator.report(solution); // intention performance update
    this.terminate(); // intention agent ceases to exist
  }
}

agentClass OrderAgent extends ProsaAgent {
  // Attributes
  ResourceAgent resource; PartialSolution state;
  ActionSequence intention = emptySequence;
  DecisionMakerFactory dmf; IntentionManager im;
  ProductAgent product;


  // Constructor
  OrderAgent (ProductAgent p, DecisionMakerFactory d) {
    product = p; dmf = d; im = dmf.getIntentionManager();
    state = product.initialState();
    resource = state.lastResource;
    ...
    // agent behaviors become active at this point
  }


  // Active behaviors (threads)


  // exploring ants are created throughout the orders lifetime
  agentBehavior createExplorers() {
    while (product.incomplete(state)) {
      new ExploringAnt (dmf.next(), state);
      stayInactive (dmf.exploreRefreshInterval());
    } // while
    this.terminate();
  }
```

```
// intention ants are created when selected throughout order lifetime
agentBehavior createIntentionPropagators() {
  while (product.incomplete(state)) {
    if (intention.notEmpty()) { new IntentionAnt (state, intention); }
    stayInactive(dmf.intentionRefreshInterval());
  } // while
  this.terminate();
}

// order agent handles actual execution of operations
agentBehavior executeOperations() {
  ... // execute intention when real time matches intention time
  ... // out of scope for this paper
}

// Actions
void report(PartialSolution s) {
  im.updateCandidateSolutions(s);
  im.updateIntentions();
    // intention manager im inserts solution s in the collection
    // of candidate solutions if sufficiently attractive.
    // intention manager im evaluates whether current intention
    // needs to be changed (implements social control mechanism).
  }
}

agentClass ProductAgent extends ProsaAgent {
  // Attributes

  ActionSet possibleActions;

  // Actions
  boolean incomplete(PartialSolution s) {
    // check whether the solution s performs all required operations
    return ...;
  }

  ActionSet possibleNextOps(PartialSolution s) {
    possibleActions = emptySet;
    // 1. Collect information on capabilities of s.lastResource
    // and insert all operations allowed on s.lastState in
    // the possibleActions set.
    ...
    // 2. Retrieve information attached to s.lastResource exits
    // and check whether all required remaining operations
    // are available beyond the repsective exits. If so,
    // insert leaveTruExit for this suitable exit in the
    // possibleActions set.
    //
    // The actual implementation is done by subclasses !
    //
```

```
      ...
      return possibleActions;
    }

    boolean notPossibleNextOp(Action Op, PartialSolution s) {
      return not(Op elementOf this.possibleNextOps(s));
    }


    PartialSolution initialState() {
      ... // create and return data structure corresponding
      return ... // to an order as it enters the production system
    }
  }

  agentClass ResourceAgent extends ProsaAgent {
    // Attributes
    ResourceScheduler rs;
    PhysicalResource; // link to device control
    EntrySet entries; // Set of resource entry points
    ExitSet exits; // Set of resource exit points
      // during configuration and reconfiguration
      // exits and entries are connected to permits
      // virtual navigation.



  // Constructors
    ResourceAgent (ResourceScheduler r) {
      Rs = r;
      ...
      // agent behaviors become active at this point
      }
  // Active behaviors (threads)
    agentBehavior evaporate() {
      // CapacityReservations that fail to be refreshed reconfirmed
      // by intention ants are removed; rs determines evaporation.
      for (;;) { rs.evaporate(); stayInactive(rs.refreshDelay());
    }
  // Actions
    abstract void virtualExecute(PartialSolution s) {
      ... // s defines executable but not-yet-executed operation op
      ... // modify s such that op is executed with expected result
      ... // and expected duration.
      ... // ResourceScheduler rs ensures that the load forecast is
      ... // accounted for.
      ... // Note that this method will be implemented by the subclasses!
    }


    abstract void virtualExecuteAndReserveCapacity(PartialSolution s) {
      ... // similar to virtualExecute but the required capacity
      ... // is reserved by rs to account for the commitment
      ... // expressed by the visiting intention ant.
      ... // Note that this method will be implemented by the subclasses
    }
  }
```

Appendix 2. Sample results

This section presents results from a sample system implementation. Note that the performance data reported in this example depends on the add-ons and is irrelevant for the focus of this paper. Nevertheless, the results are included because many researchers only appreciate this dimension and because industrial take-up critically depends on it.

The example does not include sophisticated decision making mechanisms. Instead, it is designed to benefit from the widespread availability of an off-line planning in industrial environment. These plans may originate from a finite capacity planner (e.g., from I2 Technologies) or an ERP system (i.e., an infinite capacity planner). Most often, the plans simply reflect how production management handled similar situations in the past. The research team opted for this approach because it mirrors industrial practice when people handle plan execution, which is the most common case.

The decision-making add-on considers the externally supplied plan as the initial intention, and the intention selection mechanism strongly prefers solutions according to the plan. The exploring agents ignore the plan. The results reveal an ability to stick to the plan if it remains feasible. When the plan cannot be executed, the system discovers solutions deviating from the plan with equal probability as alternatives with minor deviations from the plan. Future developments intend to have exploring ants prefer

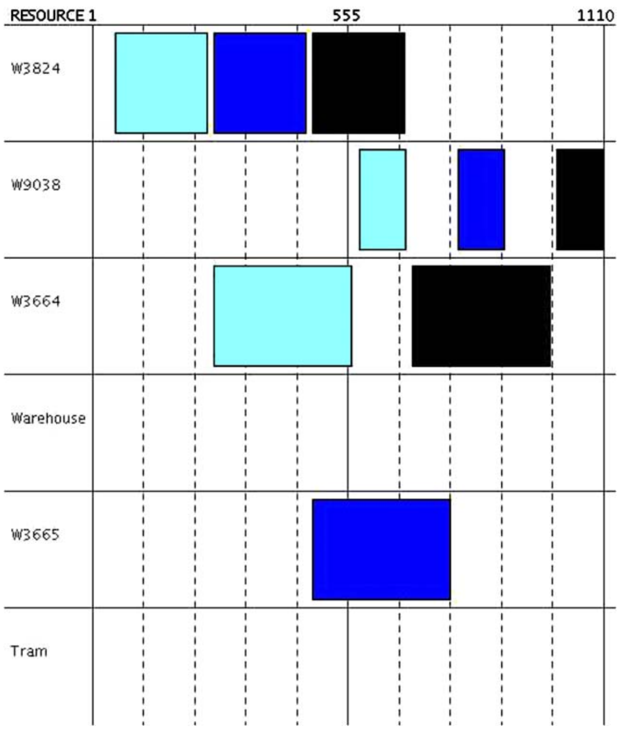
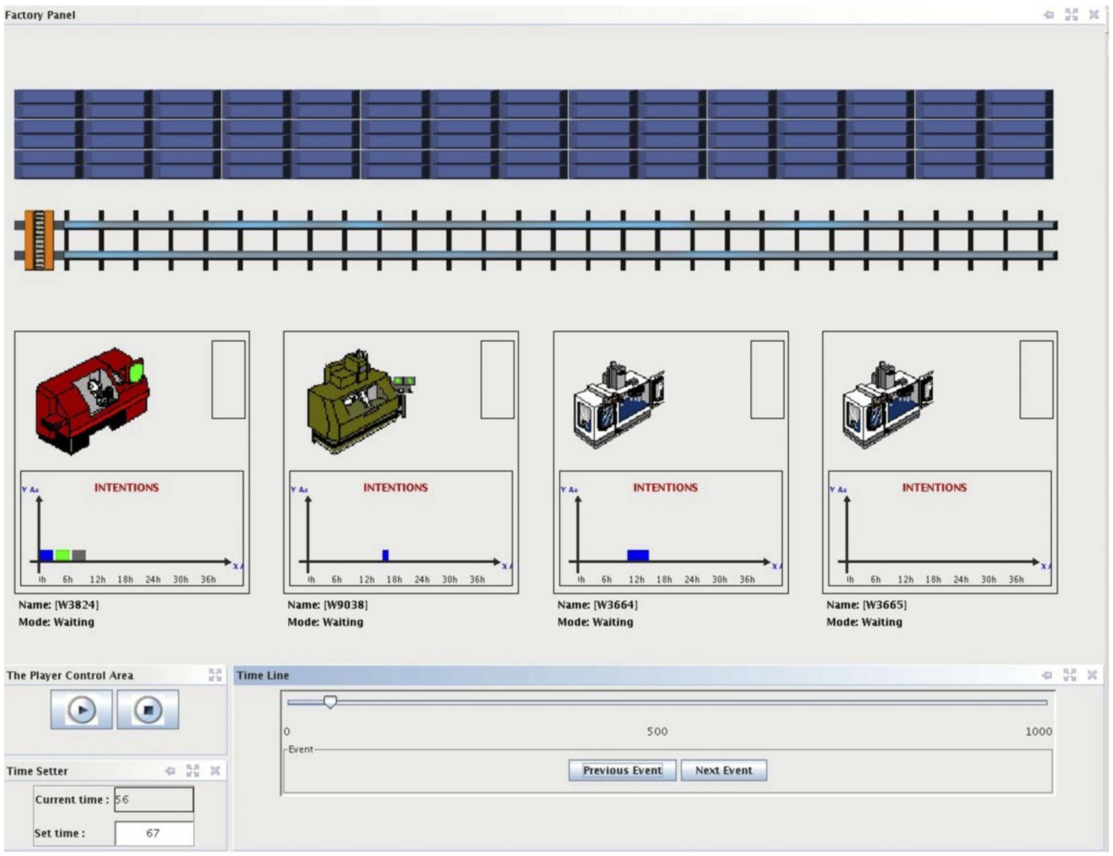


Fig. 5. The externally supplied plan.



Name: [W3824]
Mode: Waiting

Name: [W9038]
Mode: Waiting

Name: [W3664]
Mode: Waiting

Name: [W3665]
Mode: Waiting

Current time : 56
Set time : 67

Fig. 4. The manufacturing system.

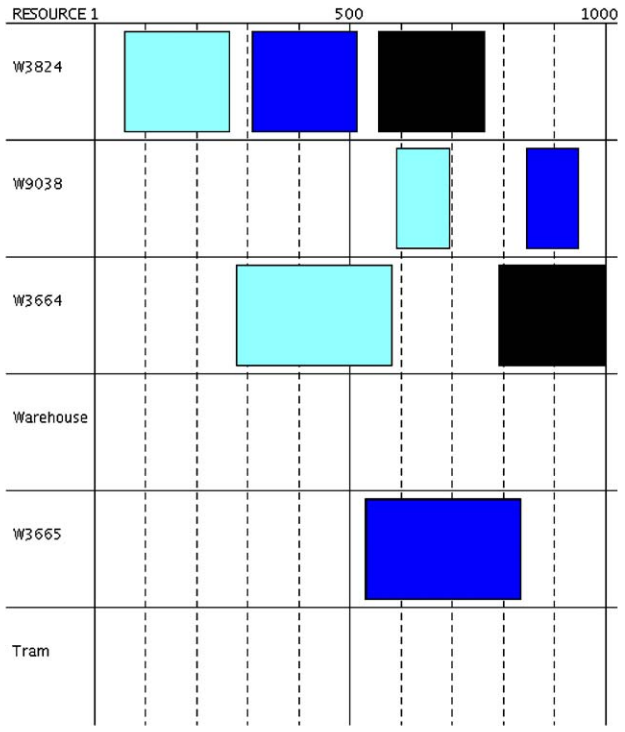


Fig. 6. Manufacturing execution follows the plan.

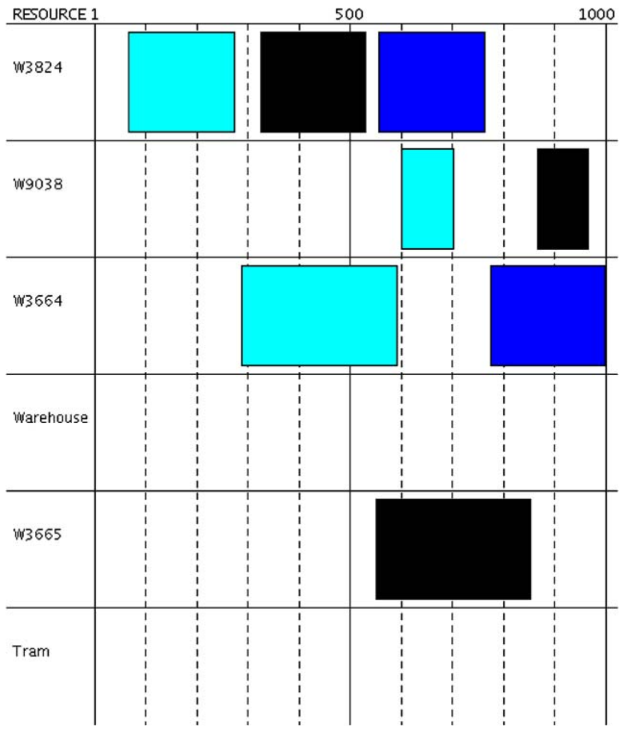


Fig. 7. Manufacturing execution deviates from the plan.

solutions respecting the plan. Research issues include determining what 'close to the plan' means as well as detecting when the plan is no longer a valid target to aim for.

Fig. 4 shows the configuration of the manufacturing system. Fig. 5 depicts the externally supplied schedule for three jobs (one color for each job) that is unfeasible because transport operations are ignored. Fig. 6 shows how the manufacturing control follows this schedule while adapting it to render it feasible. Fig. 7 shows an execution deviating from the schedule. In the current implementation, both are equally likely to occur (as in industrial practice). Further discussion is outside the scope of this paper.

References

[1] N. Duffie, Synthesis of heterarchical manufacturing systems, Computers in Industry 14 (1990) 167–174.

[2] J. Hatvany, Intelligence and cooperation in heterarchic manufacturing systems, Robotics and Computer-Integrated Manufacturing 2 (2) (1985) 101–104.

[3] S. Bussmann, K. Schild, Self-organizing manufacturing control: an industrial application of agent technology, in: Proc. 4th Int. Conf. on Multi-Agent Systems, Boston, USA, 2000, pp. 87–94.

[4] H.V.D. Parunak, A.D. Baker, S.J. Clark, The AARIA agent architecture: From manufacturing requirements to agent-based system design, Integrated Computer-Aided Engineering 8 (1) (2001) 45–58.

[5] L. Monostori, B.C. Csáji, B. Kádár, Adaptation and learning in distributed production control, Annals of the CIRP 53 (1) (2004) 349–352.

[6] P. Valckenaers, H. Van Brussel, Hadeli, O. Bochmann, B. Saint Germain, C. Zamfirescu, On the design of emergent systems: an investigation of integration and interoperability issues, Engineering Applications of Artificial Intelligence 16 (2003) 377–393.

[7] H. Van Brussel, J. Wyns, P. Valckenaers, L. Bongaerts, P. Peeters, Reference architecture for holonic manufacturing systems: PROSA, Computers in Industry 37 (1998) 255–274.

[8] H.A. Simon, The Sciences of the Artificial, MIT Press, Cambridge, MA, 1990.

[9] M. Waldrop, Complexity, the Emerging Science at the Edge of Order and Chaos, VIKING, Penguin group, London, 1992.

[10] P. Valckenaers, H. Van Brussel, Fundamental insights into holonic systems design, in: Proc. of HoloMAS'05, Copenhagen, 2005.

[11] G. Di Caro, M. Dorigo, AntNet: Distributed stigmergetic control for communications networks, Journal of Artificial Intelligence Research 9 (1998) 317–365.

[12] Hadeli, P. Valckenaers, P. Verstraete, B. Saint Germain, H. Van Brussel, Emergent Forecasting Using a Stigmergy Approach in Manufacturing Coordination and Control, Lecture Notes in Computer Science, vol. 3464, 2005, pp. 210–226.

[13] P. Valckenaers, H. Van Brussel, M. Kollingbaum, O. Bochmann, Multi-agent Coordination and Control using Stigmergy Applied to Manufacturing Control, Lecture Notes in Artificial Intelligence, vol. 2086, 2001, pp. 317–334.