Implementing Self-Organisation and Self-Management in Evolvable Assembly Systems

Regina Frei, Giovanna Di Marzo Serugendo
Department of Computer Science and Information Systems
Birkbeck College, University of London
London WC1E 7HX, United Kingdom
Email: work@reginafrei.ch, dimarzo@dcs.bbk.ac.uk

Nuno Pereira, José Belo, José Barata
Departamento de Engenharia Electrotecnica
Faculdade de Sciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
Email: nunoenpereira@gmail.com,
jnb555@sapo.pt, jab@uninova.pt

Abstract—**Evolvable assembly systems (EAS) are intended to tackle the challenges of agile manufacturing: high responsiveness, the ability to cope with ever-changing requirements, many variants and small lot sizes. This article discusses *self-organising evolvable assembly systems (SO-EAS)*, which are a research direction of EAS focusing on *self-organisation* and *self-management*. SO-EAS are composed of modules with local intelligence and self-knowledge, able to self-organise to form a suitable shop-floor layout which fulfils a generic assembly plan received in input. During production, the modules self-manage while executing the assembly tasks. This article reports on the latest implementation advances, such as the integration of the agent platform Jade with the reasoning engine Jess.**

I. Introduction

Nowadays, robotics and automation play an important role in many fields such as search and rescue, medicine, warfare, space exploration as well as *manufacturing* and *assembly*. Robots can collaborate with humans and replace them in the execution of repetitive, monotonous and dangerous tasks. Nevertheless, also in highly automated systems, the human user plays important roles, especially in system configuration, robot programming, supervision / monitoring, error recovery and system reconfiguration. These tasks are often work-intensive and error-prone procedures.

With systems becoming increasingly complex, there is a need for systems that can take care of themselves (exhibit *self-* properties*) and play a proactive role [9]. Concretely, in *self-organising evolvable assembly systems (SO-EAS)*, modules self-organise to create the shopfloor layout and self-manage when assembling small mechatronic products.

Organisation of this article: The software architecture and infrastructure is detailed in section II, and section III explains the implementation issues so far addressed. Discussion, conclusion and further work follow in section IV.

A. Evolvability in assembly systems

Evolvability [21] refers to a system's ability to continuously and dynamically undergo modifications of varying importance: from small adaptations to big changes. Products, processes and systems are intrinsically related to each other. Each *product* belongs to a certain product class [26] which requires a certain limited set of processes, where a *process* means a coherent suite of assembly operations which lead to the finished product. Production processes are intimately linked to product design and to system module capabilities: any change in the product design has an impact on the processes to apply and on the actual assembly system to use. Similarly, any change in a joining process (for instance replacing a rivet by a screw) may imply a change in the product design, and certainly has an impact on the assembly system to use.

B. Contributions in the area of EAS

Evolvable assembly systems (EAS) [20], [1] refer to modular assembly systems that seamlessly integrate new modules thanks to an agent-based control approach; that evolve / adapt to changes in products, processes and assembly systems; and that make automation more accessible to SMEs.

The research activities in the area of EAS have gradually increased over the last years, and we identify five main research areas:

Agent-based technologies: A multi-agent shopfloor control system has been developed with all modules of the shop-floor agentified and coordinating their work during production [1]. The use of agent-based architectures for EAS was compared with the use of service-oriented architectures, modules offer their skills under the form of services [23]. Dynamic coalitions [13] have been defined that allow the modules to spontaneously create and change coalitions, as well as to request other modules to join them in order to satisfy all requirements of a task to be fulfilled.

Ontologies and specifications: EAS ontologies were developed for products, processes and systems [15]; for a consumer electronics industrial test case [17], and for controlling production [24].

Roadmaps and the EUPASS project: Directions for further research were formulated [5]. Within the EUPASS project[1], numerous academic and industrial partners made EAS advance [25]. A roadmap for adaptive assembly technology [22] was written.

Diagnosis: Systems for the diagnosis of EAS [4] have been investigated, as well as the suitability of service-oriented architectures for EAS diagnosis [3].

Self- properties and emergence:* The possibility of emergent behaviour in EAS was investigated [2], a proposal for

[1] http://www.hitech-projects.com/euprojects/eupass/index.htm

self-reconfigurabililty of the assembly system has been proposed [18]. SO-EAS belong to this direction of research, and will be detailed below.

Proactive assembly systems [7] are conceived on the same mind-set as evolvable assembly systems, but the focus is on the operator, who is part of the assembly system and an important factor. The human must be a highly specialised expert which is very active (that is, proactive) in all areas of an assembly system life cycle. In SO-EAS, we go a step further: it is the system which is proactive and provides innovative services to the user.

C. Self-organising evolvable assembly systems

SO-EAS are a specific case of EAS, where: (1) Assembly system modules **self-organise**, that is, modules select suitable partners to form dynamic coalitions which have the skills to fulfill the assembly plan of incoming product orders, and coalitions choose their position in the shopfloor layout; self-organisation includes also self-reconfiguration in case of changes in the product design or the assembly processes; (2) Assembly system modules **self-manage** during production, that is, they monitor themselves as well as their coalition partners and adapt their behaviour accordingly; if necessary, an assembly system reconfiguration process is triggered.

The design of SO-EAS has been explained in [11], and certain aspects have been implemented: dynamic coalitions [13], [14] allow the modules to spontaneously create and change coalitions, as well as to request other modules to join them in order to satisfy all requirements of a task to be fulfilled.

This article reports on the current status of the implementation, which consists of: (1) adapting ontologies, (2) representing the *Generic Assembly Plan* and the *Layout-Specific Assembly Instructions* as workflows, (3) defining agents in Jade, with generic core behaviour, sensitive to Jess rules, and (4) implementing agent reactions based on communication, which forms the basis for further full implementation of the CHAM design (*Chemical Abstract Machine* [6]).

II. Architecture and infrastructure

SO-EAS are composed of assembly system modules. A software agent is associated with every module and equipped with thorough self-knowledge [10]. An SO-EAS receives four main inputs from the user:

a) A set of ontology files containing the description of the available assembly system modules.
b) A *generic assembly plan* (GAP, described in section II-A), containing a sequence of operations needed for the product to be assembled.
c) The number of products to be assembled.
d) A list of rules for self-organisation and policies for self-management to guide the system and keep it under control.

In a self-organising process guided by rules, modules find suitable partners to form *coalitions* to fulfill the tasks specified in the GAP; the coalitions then arrange themselves in the shopfloor layout. Once the layout has been built, the *layout-specific assembly instructions* (LSAI, described in section II-C) are derived, and executed by the modules on request of the products to be assembled. During production, the agents monitor themselves and their peers to make sure that they are working according to the policies for self-management. If a policy is broken, appropriate measures are taken, again as specified by corresponding policies. For instance, in case a module does not answer as expected, it may be requested to restart its software. If the problem persists, the other coalition partners may search for a replacement module.

A. The Generic Assembly Plan (GAP)

The GAP specifies the way a product is to be assembled: it includes the assembly sequence of the different parts and the way they must be joined. Tasks are defined in the form of generic *skills*, that is module capabilities. The GAP does not provide information about what module to use and what movement to make. In other words, the GAP says *what* to do but not *how* and is thus independent from any concrete layout. Figure 1 shows the example of a GAP represented as a workflow and written in XML. The four simple illustrated tasks each have an operation type (Op), an object to be handled (Obj), a start point (StPt) where the part is grabbed, an end point (EndPt) where the part is released, as well as a start orientation (StOr) and an end orientation (EndOr). This GAP specifies that a carrier is loaded from the $Storage$ to $Conveyor_1$, then $Part_1$ is picked from $Feeder_1$ and placed on the $Carrier$, then $Part_2$ is picked from $Feeder_2$ and placed on top of $Part_1$, and finally, the $Carrier$ with the assembled product is unloaded to the $Storage$.

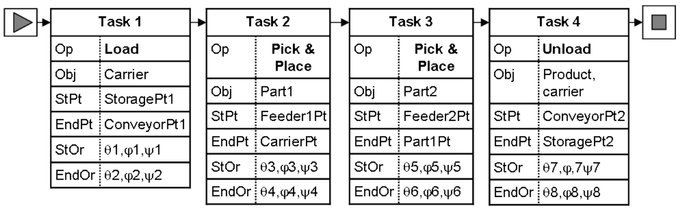


Fig. 1. Example of a GAP written as a workflow

B. Layout creation

The layout is incrementally built according to a process which follows a self-organising mechanism inspired by chemical reactions as illustrated in Figure 2. Modules self-assemble to form coalitions according to a process of reactions, modelled according to chemical reactions. Coalitions are built to progressively match with the tasks defined in the GAP. In this case, 'chemical' reactions occur: (1) when modules agree to collaborate with others (i.e. they form coalitions), and (2) when a task 'bonds' with the skills offered by a coalition (the coalition offers its skills).

When a GAP appears in the system, all the modules immediately look for tasks which they can fulfil alone. If they find a suitable task, they offer their services. Otherwise they search for suitable partner modules, according to their

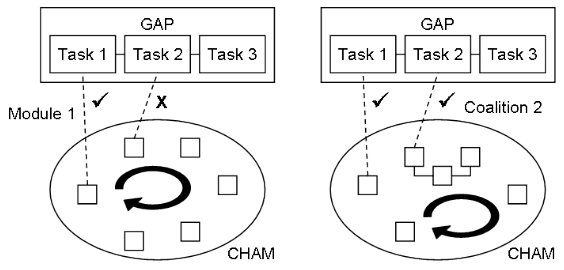


Fig. 2. Self-assembly of coalitions, following the CHAM paradigm

own physical requirements as well as the corresponding rules for module composition. For instance, a gripper will look for a robot to hold it. Once they have found all their partners and formed a *complete* coalition (that is, the coalition can provide all requested skills and does not have any open requirements), they look again for suitable tasks in the GAP and, if found, offer their services. If the coalition does not have the required composite skills yet, additional or alternative partners are searched and asked to join the coalition. This procedure continues until all the tasks have received at least one service offer, or until no more modules are available. In this case, the task will emit a user alert after a certain time because no satisfactory solution was found.

An *incomplete* coalition which, in the process of forming itself, cannot find any suitable partners in the module pool, will be discarded. If a task gets more than one service offer, there are several possibilities for choosing a coalition: (1) The user makes a choice; (2) The first suitable coalition making an offer automatically gets the job; (3) The choice follows an optimisation rule, selecting the coalition with the lowest number of modules, or the newest modules, or the one with the shortest estimated processing time for the task at hand.

A coalition which has successfully offered its services and has been accepted (that is, assigned the task) will then determine its position within the layout. The first coalition may choose at random (or start at a position defined by the user), and the following coalitions will then join them, leaving suitable distances between them and asking for the necessary conveyor paths to be laid. Additional chemical reaction rules apply for the layout. A full specification in Maude of the chemical rules is available on request from the authors. For more details about CHAM and Maude, see [12].

C. The Layout-Specific Assembly Instructions (LSAI)

For actually assembling the product, the GAP needs to be transformed into *Layout-Specific Assembly Instructions (LSAI)*. This is done in collaboration between the *order agents* (agents which represent the orders to be fulfilled) and the module agents and based on the created layout. For transforming the GAP into the LSAI, a rewriting mechanism [19] is applied.

The LSAI consist of executable instructions for the modules. The instructions are generated for a certain layout; if the layout is modified, these instructions must be changed. Figure 3 shows the example of an LSAI represented as a workflow, based on the GAP in Figure 1. $Coalition_1$ is composed of $robot_1$ and $gripper_1$. In addition to the information contained in the GAP, the LSAI also specifies the module(s) which execute the tasks (actor). Additional tasks A and B assure the transport between the loading / unloading and the assembly (tasks 1, 2, 3 and 4). Tasks 2-1 and 3-1 provide the delivery of the parts to be assembled.

At production time, the assembly of a product will result from the execution of the LSAI by the agents/modules according to the workflow, taking into account the policies for self-management.

Both the GAP and the LSAI (see section II-C) must support sequences of activities, parallel activities, free changes of state from one activity to another, and changes of state controlled by a condition. The *workflow* notation is currently used and has already been tested without conditions controlling the change of states. Workflows have been successfully used to control an educational shopfloor kit (MOFA, see [13]). The test was done by the means of a small agent program using the Jade platform. An agent behaviour was created solely with the purpose of executing the workflow. This behaviour was inside every agent in the platform and every robotic module was represented by an agent running its own workflow. Aside from the modules agents, there were agents representing the parts being processed, also with a workflow behaviour inside, and their function was to run the workflow: to discover what action was needed next and to send the corresponding part of the workflow to be executed by a module agent. This approach, with some minor changes, is used here at a first stage and will be refined later. The changes will result from the conditions that can exist between states, and from the possible attribution of the part agent's role to other agents.

D. Architecture

Figure 4(a) illustrates an abstract view of the architecture for self-organisation and self-management. It is an agent-based approach, where modules are agentified and provide skills in the form of services. It has been adapted from MetaSelf [8] and exploits *metadata* to support decision-making and adaptation based on the dynamic enforcement of explicitly expressed *policies for self-management* and *rules for self-organisation*. Figures 4(b) and 4(c) illustrate the actual decentralised software architecture at module and coalition level. Each module / coalition agent is equipped with metadata (information about itself), a reasoning engine and local policies, and has access to the whole ontology. The components, metadata, rules and policies are all *decoupled* from each other and dynamically updated (or changed).

Additional services to build the run-time infrastructure encompass: a registry/broker that handles the service descriptions and services requests supporting dynamic binding; an acquisition and monitoring service for the self-* property related metadata (e.g. performance); a registry that handles rules and policies; a reasoning tool that matches metadata values, rules and policies, and enforces the rules and policies on the basis of metadata. Metadata is either directly modified

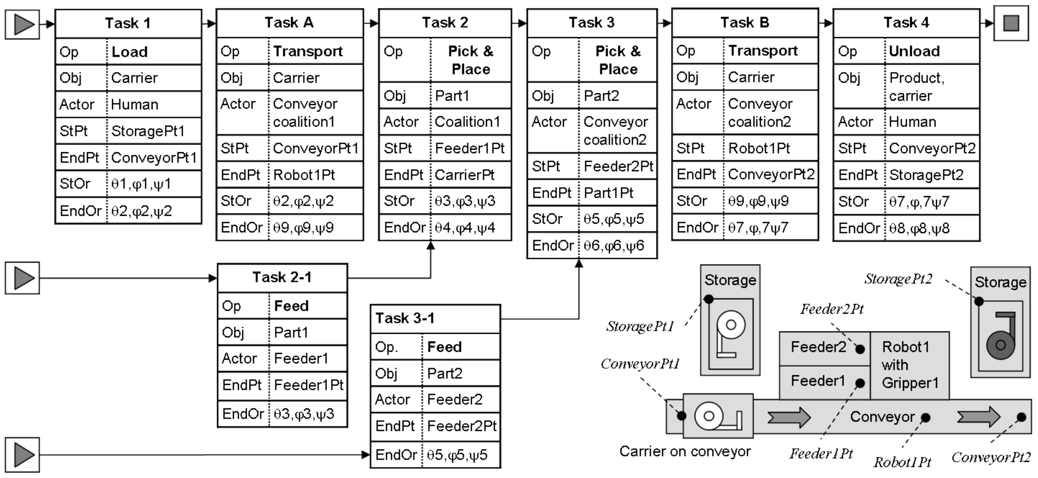
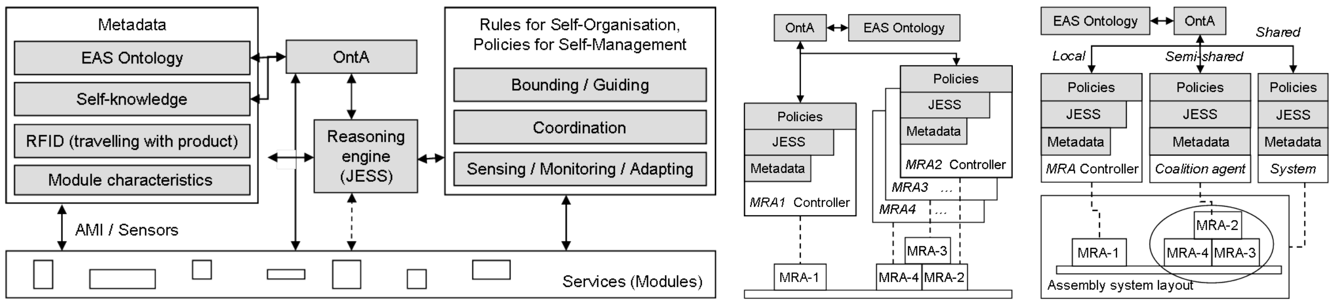


Fig. 3. Example of an LSAI written as a workflow



(a) SO-EAS software architecture (b) Module level (c) Coalition level

Fig. 4.

by components or indirectly updated through monitoring. Metadata and policies cause the reasoning tool to determine whether or not an action must be taken.

III. Implementation

The implementation described in this section follows the architecture detailed in section II.

A. The Jess rule engine

Jess[2] is a rule engine written in Java which allows a Java application to reason through a given set of rules and facts. The Jess rule engine uses an enhanced version of the Rete algorithm[3], which is a very efficient pattern-matching algorithm for implementing production rule systems.

In practical terms, this engine has to receive facts into a so-called *working memory*. Facts are a way to know the status of the outside world, and, according to these facts, the rules (in our case: rules for self-organisation or policies for self-management) fire if their conditions are met. Conflicting rules are treated according to their priorities.

[2]http://www.jessrules.com
[3]http://www.jessrules.com/jess/docs/52/rete.html

B. The ontology

The ontology used for describing each of the modules was not done from scratch; instead, extracts from the EUPASS ontology [17] were adapted. The program being used to add elements to the ontology is Protégé[4], which creates OWL[5] files used as an input for the application.

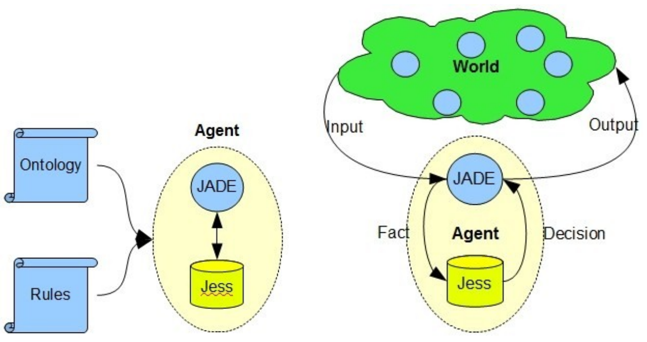
We added some new classes and properties to the ontology primarily because many of the real characteristics of robotic modules had not been included in the ontology before but are now important for the rules. The addition of the missing elements to the Eupass ontology was done according to robotic module manufacturer websites and the modules' data-sheets. For the the ontology classes to be easier to use inside the agents, they were converted to Java classes using a Protégé capability do convert OWL to Java.

C. Agent architecture

To comply with the architecture (loosely coupled policies and components), the agents representing manufacturing components are generic: they have a common core behaviour which may be adapted according to ontologies and rules

[4]http://protege.stanford.edu
[5]http://www.w3.org/TR/owl-features

loaded at runtime as shown in Figure 5(a). Notice that in Jess terms, *rules* refer to both rules and policies.



(a) Ontologies and rules are loaded at run-time. (b) Execution of rules in a JADE agent, using JESS.

Fig. 5.

The agents themselves are programmed using JADE API. Agents read ontologies, gather facts for the rule engine and perform the actions the rule engine needs.

The ontologies are loaded using the Protégé generated Java classes, giving an individual agent a set of properties during runtime. This gives the agent the possibility to be reconfigured each time a new ontology is loaded. The objective is to have agents whose behaviours are guided by policies and rules. The chosen agent architecture has been adapted from [16] and is shown in Figure 5(b). Each agent has its own inference engine in Jess that is in charge of its decisions according to the facts that were inserted in it. This makes the agent changeable at run-time instead of design time, which is a considerable advantage.

Figure 5(b) illustrates the input the agent receives from the outside world and its conversion into a fact to insert in the Jess rule engine. This new fact may, on its own or in combination with other facts, already present in the engine's working memory, cause the firing of a rule that was previously loaded. In this case, the engine may emit an order to the agent to execute an action on itself or on the outside world.

D. Internal functioning of the agents

In the current implementation, an agent is able to load an ontology describing the module to which it is associated, and a set of Jess rules. The description of the module is fed to the agent in an OWL file. The agent reads the given OWL file and creates the needed object with the help of the converted ontology classes. This object is then inserted in the agent's working memory, so that inferences can be made for it in the rule engine. An agent is also able to react with other agents to create coalitions according to the rules that were loaded.

The current implementation works with Jade and Jess to create reactions as described in section II-B. Reaction rules are implemented with communication among the agents, according to a FIPA compatible protocol. Each agent has its own Jess rule engine to which are loaded a set of simple rules and a set of simple facts. The currently supported **facts** are:

ThisAgent: Each created agent has one such fact to describe itself in the rule engine. It is added to the agent's working memory during the initialisation.

NeighbourAgent: A single agent can have several facts of this type to describe the surrounding agents. Each time an agent is registered in the system, the other agents create a fact of this type describing it.

ACLMessageExt: This is an extension to the ACLMessage[6] class used in Jade. Each time an agent receives an ACLMessage, it converts it to this type of fact and inserts it in the working memory [16].

When the agents are launched in the agent platform, they react with each other according to the rules for coalition formation. A newly formed coalition is represented by a newly created agent. Coalitions also register themselves in Jade's Directory Facilitator (DF). This is necessary because the agents need to know of the existence of neighbour agents in order to communicate; before they start reacting, they ask the DF about other agents.

As previously stated, the rule engine may fire a rule when a fact is added to its working memory. At this time, there are only four simple **rules** in the rule engine. These rules will make the agents react with each other, creating a new agent for the coalition from this reaction. The communications follow the FIPA protocol:

Propose rule: A single agent waits for a fact describing a neighbour agent to be inserted in its working memory. When this happens, this agent proposes a reaction to the other agent.

Accept Reaction rule: A single agent waits for a fact of the type of ACLMessageExt that describes a reaction proposal. If the agent is compatible with the reaction proposal (matching skill or physical compatibility), this rule is fired, the agent accepts the proposal and a new agent is created out of the reaction of the two original agents.

Reject Reaction rule: A single agent waits for a fact of the type of ACLMessageExt that describes a reaction proposal. If the agent is not compatible with the reaction proposal, this rule is fired, the agent rejects the proposal and no new agent is created.

React rule: The proposing agent waits for a fact of the type ACLMessageExt that describes a reaction acceptance. If this rule is fired, the agent's rule engine tells the agent itself to create another coalition agent with the properties of both the reacting parties.

The mechanism works as follows: one agent asks the DF to know all the registered agents. The DF then adds their descriptions (NeighbourAgent objects) to the rule engine. The rule engine then checks to see if any rule can be triggered by this. In our implementation this should fire the Propose rule. This rule makes the agent send reaction proposals to all other agents. When the other agents receive the message, they convert it to an ACLMessageExt object for Jess compatibility purposes and add it to their own engine. This should fire the AcceptReaction rule or the RejectReaction rule, according to

[6]Agent Communication Language

their compatibility. Both the rules cause the agents to respond with either an Accept Proposal or a Reject Proposal. The initiator agent receives the response and, if it is an Accept Proposal, tries to create the coalition agent.

IV. Discussion, conclusion and further work

This article presented the current implementation advances for the concrete realisation of self-organising evolvable assembly systems. An ontology was adapted for the current needs and now contains relevant data about real instances of grippers, robotic axes and feeders. The generic assembly plan (GAP) and the layout-specific assembly instructions (LSAI) were represented as workflows and written in XML, which is an agent-readable format. Jade was integrated with Jess and showed promising results. Agents with generic core behaviour, which are sensitive to Jess rules, were defined in Jade. To create coalitions, agent react with each other based on FIPA-compatible communication. This is a first basis for further implementation of the CHAM design.

The next steps include: adding data-sheet information about more types of modules to the ontology; enabling the writing of rules directly on the application interface; implementing the functions needed to define the system layout; showing the defined layout to the user; implementing the agent behaviours that will execute the LSAI; deriving the LSAI from the GAP. *Future steps include:* 2D or 3D visualisation of the modules in a virtual room; acceptance of higher level policies such as goal based policies and utility-function based policies.

Acknowledgments

The authors thank Mauro Onori from KTH Stockholm as well as Niels Lohse and Pedro Ferreira from Nottingham University for their comments on this paper and fruitful discussions about the topic.

Regina Frei is supported by the Portuguese Foundation for Science and Technology (PhD grant SFRH / BD / 38608 / 2007).

The authors also thank the EU-funded coordination action Perada[7] for financially supporting travel exchange between Uninova and Birkbeck College.

References

[1] J. Barata. *Coalition based approach for shopfloor agility*. Edições Orion, Amadora - Lisboa, 2005.

[2] J. Barata and M. Onori. Evolvable assembly and exploiting emergent behaviour. In *IEEE Int. Symp. on Industrial Electronics (ISIE)*, Montreal, Canada, 2006.

[3] J. Barata, L. Ribeiro, and A.W. Colombo. Diagnosis using service oriented architectures (soa). In *5th IEEE Int. Conf. on Industrial Informatics (INDIN)*, volume 2, pages 1203–1208, Vienna, Austria, 2007.

[4] J. Barata, L. Ribeiro, and M. Onori. Diagnosis on evolvable assembly systems. In *IEEE Int. Symp. on Industrial Electronics (ISIE)*, pages 3221–3226, Vigo, Spain, 2007.

[5] J. Barata, P. Santana, and M. Onori. Evolvable assembly systems: A development roadmap. In *IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, St Etienne, France, 2006.

[6] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1998.

[7] K. Dencker, J. Stahre, P. Groendahl, L. Martensson, T. Lundholm, and C. Johansson. An approach to proactive assembly systems. In *IEEE Int. Symp. on Assembly and Manufacturing*, pages 294–299, Ann Arbor, MI, USA, 2007.

[8] G. Di Marzo Serugendo, J. Fitzgerald, and A. Romanovsky. Metaself - an architecture and development method for dependable self-* systems. In *25th Symp. on Applied Computing (SAC)*, pages 1–1, Sion, Switzerland, 2010.

[9] R. Frei and M. Barata, J.and Onori. Evolvable production systems, context and implications. In *IEEE Int. Symp. on Industrial Electronics (ISIE)*, pages 3233–3238, Vigo, Spain, 2007.

[10] R. Frei, G. Di Marzo Serugendo, and J. Barata. Designing self-organization for evolvable assembly systems. Technical report, BBKCS-09-04, School of Computer Science and Information Systems, Birkbeck College, London, UK, 2006.

[11] R. Frei, G. Di Marzo Serugendo, and J. Barata. Designing self-organization for evolvable assembly systems. In *IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 97–106, Venice, Italy, 2008.

[12] R. Frei, G. Di Marzo Serugendo, and T.F. Serbanuta. Ambient intelligence in self-organising assembly systems using the chemical reaction model. *Submitted*, 2010.

[13] R. Frei, B. Ferreira, and J. Barata. Dynamic coalitions for self-organizing manufacturing systems. In *CIRP Int. Conf. on Intelligent Computation in Manufacturing Engineering (ICME)*, Naples, Italy, 2008.

[14] R. Frei, B. Ferreira, G. Di Marzo Serugendo, and J. Barata. An architecture for self-managing evolvable assembly systems. In *IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC)*, San Antonio, TX, USA, 2009.

[15] N. Lohse. *Towards an ontology framework for the integrated design of modular assembly systems*. PhD thesis, School of Mechanical Materials and Manufacturing Engineering, Faculty of Engineering, University of Nottingham, Nottingham, UK, 2007.

[16] H. Lopes Cardoso. Integrating jade and jess. http://jade.tilab.com/doc/tutorials/jade-jess/jade_jess.html, 2007.

[17] A. Maffai and T. Rossi. *Development of an Ontology to support the EAS: ELECTROLUX Test case*. M.sc. thesis, University of Pisa, 2007.

[18] A. Maffei, K. Dencker, and M. Bjelkemyr. From flexibility to evolvability: Ways to achieve self-reconfigurability and full autonomy. In *Int. IFAC Symp. on Robot Control (SYROCO)*, Gifu, Japan, 2009.

[19] J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR*, pages 384–400, 1990.

[20] M. Onori. Evolvable assembly systems - a new paradigm? In *33rd Int. Symposium on Robotics (ISR)*, pages 617–621, Stockholm, Sweden, 2002.

[21] M. Onori, J. Barata, and R. Frei. Evolvable assembly systems basic principles. In *IFIP Int. Conf. on Information Technology for Balanced Automation Systems (BASYS)*, pages 317–328, Niagara Falls, Canada, 2006.

[22] M. Onori, C. Hanisch, J. Barata, and T. Maraldo. Adaptive assembly technology roadmap 2015, project report-public, document 1.5f, nmp-2-ct-2004-507978, 2008.

[23] L. Ribeiro, J. Barata, and P. Mendes. Mas and soa: Complementary automation paradigms. In A. Azevedo, editor, *Innovation in Manufacturing Networks*, volume 266, pages 259–268. Springer Boston, 2008.

[24] L. Ribeiro, J. Barata, M. Onori, and A. Amado. Owl ontology to support evolvable assembly systems. In *9th IFAC Workshop on Intelligent Manufacturing Systems (IFAC-IMS)*, Szczecin, Poland, 2008.

[25] D. Semere, J. Barata, and M. Onori. Evolvable systems: Developments and advance. In *IEEE Int. Symposium on Assembly and Manufacturing (ISAM)*, pages 288 – 293, Ann Harbor, MI, USA, 2007.

[26] D. Semere, M. Onori, A. Maffei, and R. Adamietz. Evolvable assembly systems: coping with variations through evolution. *Assembly Automation*, 28(2):126–133, 2008.

[7]http://www.perada.org/