Experience Report in Developing and Applying a Method for Self-Organisation to Agile Manufacturing

DI MARZO SERUGENDO, Giovanna, FREI, Regina

Abstract

The design and implementation of distributed, self-organising and self-adaptive systems are challenging. This article details our experience gained during the development of self-organising assembly systems, which provide solutions for user-friendly agile manufacturing systems. More specifically, we describe how both a development method for self-organising systems, called MetaSelf, and the above particular application were progressively shaped, each influencing the other.

Reference


DOI : 10.1109/SASO.2010.24

Available at: http://archive-ouverte.unige.ch/unige:48317

Disclaimer: layout of this document may differ from the published version.
Experience report in developing and applying a method for self-organisation to agile manufacturing

Giovanna Di Marzo Serugendo  
School of Computer Science and Information Systems  
Birkbeck College, University of London  
London, UK  
dimarzo@dcs.bbk.ac.uk

Regina Frei  
Department of Electrotechnical Engineering  
New University of Lisbon  
Caparica, Portugal  
regina.frei@uninova.pt

Abstract—The design and implementation of distributed, self-organising and self-adaptive systems are challenging. In this article we report on our experience gained during the development of self-organising evolvable assembly systems, which are systems providing solutions for user-friendly agile manufacturing systems. More specifically, this paper describes how both a development method for self-organising systems and the above particular application have been progressively shaped, each influencing the other.

I. INTRODUCTION

This article describes our experience with the creation of Self-Organising Evolvable Assembly Systems (SO-EAS) [1]. The initial idea was to add self-organising and self-managing capabilities to assembly systems. We identified a suitable architecture for self-organising and self-managing systems. The first step consisted in designing and developing SO-EAS according to that architecture. As the research work progressed, experience gained with the design and development of the SO-EAS provided feedback and insight into the actual development of self-organising systems. As a result, an initial development method, called MetaSelf, for self-organising and self-managing systems was devised, mainly based around the original architecture. Gradually, improvements in the development method caused changes in the design and development of the SO-EAS, which were revisited and in turn provided additional feedback for improving the method. This paper reports on the different progression stages of both the SO-EAS and the development method and how each one influenced the other.

II. MANUFACTURING SCENARIO

The production and assembly of small mechanic and electronic products such as mobile phones, computer mice or washing machine handles is nowadays mostly automated; robots in the industrial shopfloor assemble the product parts according to the customers’ orders. Manufacturing systems of the future have to be agile, distributed, user-friendly and increasingly autonomous. They need to cope with frequently changing requirements, low production volumes, many product variants, as well as perturbations and failures. Mechanical system reconfigurations are facilitated by modular hardware, but (re-)programming manufacturing systems remains a tedious, work-intensive and error-prone procedure. Therefore system designers take inspiration from natural complex systems to build agile manufacturing systems [2], [3], [4], with additional influence from Autonomic [5], Pervasive\(^1\) / Ubiquitous\(^2\) and Organic [6] computing.

A manufacturing system can be considered as a multi-agent system, which needs to fulfil specific tasks. Indeed, manufacturing system modules are agentified thanks to software wrappers. Similarly, product orders and parts are represented by agents. The following introductory subsections report on one of the approaches currently being developed, which focuses on evolvability and self-organisation.

A. Evolvable Assembly Systems (EAS)

Evolvable Assembly Systems [7] consist of robotic modules of varying granularity. A module is either an entire industrial robot with several skills (i.e. screwing, rotating and linearly moving) or a simpler module such as a robotic axis, a gripper, a feeder, or a conveyor having a single

\(^1\)http://www.perada.eu  
\(^2\)http://sandbox.xerox.com/ubicomp

Figure 1. EAS modules
Every module is an embodied agent with thorough self-knowledge (about its skills and physical characteristics) as well as social abilities (to coordinate its work with other modules). Complex or simple modules engage into coalitions (see Figure 2), in order to provide composite skills necessary to assemble the product at hand. For instance, a gripper able to seize and release parts forms a coalition with a rotating robot in order to provide a screwing skill.

Evolvability stands for the system’s ability to continuously and dynamically undergo modifications of varying importance: from small adaptations to big changes.

Coalitions in EAS are statically created (off-line) by an engineer. Modifying a coalition implies redesigning and re-programming the whole assembly system.

B. Self-Organising Evolvable Assembly Systems (SO-EAS)

Self-Organizing Evolvable Assembly Systems extend EAS in the following way: given a specified product order provided in input, the modules spontaneously select each other and choose their position in the assembly system layout. They also program themselves (micro-instructions for robots movements). The result of this self-organizing process is a new or reconfigured assembly system that will assemble the ordered product. This means that the appropriate assembly system emerges from a self-organization process.

Any new product order (seen as a global goal) triggers the self-organizing process, which eventually leads to a new appropriate system - there is no central entity, modules progressively aggregate to each other in order to fulfil the product order. This automated process does not stop at the layout formation. During production time, whenever a failure occurs in one or more of the currently used modules of the system, the process may lead to two different outcomes: the current modules adapt their behaviour (change speed, force, task distribution, etc) in order to cope with the failure, possibly degrading performance but maintaining functionality; or may decide to trigger a re-configuration leading to a repaired system. The decision will depend on the situation at hand and on specific production constraints (cost/speed/precision).

An SO-EAS is thus an EAS with two additional characteristics: 1) modules self-organize to produce an appropriate layout for the assembly and 2) the assembly system as a whole self-adapts to production conditions.

III. EXPERIENCE

This section explains and Figure 3 illustrates how the stepwise design of the SO-EAS architecture went in conjunction with the evolution of the design method. The different steps are detailed in the following subsections. We revisited several times the design of SO-EAS, in particular the self-organising mechanisms, as can be seen in Figure 4. Figure 5 shows the evolution of the development method with its consecutive steps.

A. Misty clouds (0)

In the beginning, we had only a vague idea of the exact functionality provided by Self-Organising EAS. This stage included a literature review to gain theoretical knowledge about self-organisation in natural systems, specific self-organising mechanisms, to identify and understand concepts such as complexity, emergent phenomena and so on.

1) Method: At this early stage, there was no method applied, it was typical trial-and-error: ideas were first introduced, partially implemented, and then discarded.
2) **Application:** We started from the concept of EAS, where system modules are agentified and called *manufacturing resource agents (MRAs)*. MRAs need to form coalitions with suitable other MRAs to provide specific *composite skills*, which correspond to the combined capabilities of the MRAs in a coalition. Within the EAS concept, the composition of coalitions is manual and static. A first extension consisted in providing dynamic coalitions.

3) **Output:** We introduced (Figure 4 (0)) *dynamic coalitions* formed and modified by the agents themselves. The design abstraction we used was the autonomous service composition, also called self-assembly of service chains [8]. One MRA starts a chain and then, according to its own requirements, calls a partner. This MRA in turn calls the partners it requires, and so on. For instance, if a screw needs to be positioned and screwed within another part, a gripper is first needed to seize the screw, the gripper in turns needs a rotating robot that will hold the gripper and provide the rotating movement that will actually fix the screw in the specified part. That robot in turn needs a linear axis that holds the rotating robot and provides linear movement. Altogether they can move and seize the screw from its feeder, then move the seized screw on top of the insertion point and then rotate it.

At the same time, *routing tables* (similar to those used in telecommunications) support the *product agents (PAs)* to find their way through the shopfloor layout: from the beginning (loading of the first part) to the end (unloading of the finished product). Thanks to these routing tables, the MRAs know where to send the products after having worked on them.

At this stage, we considered only *production time* (i.e., when the system is up and running and producing products). [9] reports on this state of our work.

**B. MetaSelf architecture (1)**

At this stage, we decided to use the MetaSelf architecture for its internal and external control capabilities, and for its focus on both self-organising and self-managing issues. From then on, our approach became also more structured. Our main purpose was to develop SO-EAS according to the MetaSelf architecture, therefore we didn’t consider development methods for self-organising systems such as Adelfe [10].

MetaSelf [11] is an architecture for self-organisation and self-adaptation. It involves loosely coupled autonomous components, repositories of metadata and executable policies, and reasoning services which dynamically enforce the policies on the basis of metadata values. Metadata are of different types (self-description of components, coordination and performance related metadata). Policies are both used as rules supporting a self-organising mechanism among the components or punctual policies to recover from faults supporting self-managing activities.

1) **Method:** We followed a very generic method broken down into Analysis, Design and Implementation, keeping in mind the intention to develop SO-EAS according to the MetaSelf architecture. We focused on the analysis and requirements of SO-EAS.

2) **Application:** We informally identified policies and metadata on the basis of a series of scenarios. Metadata comes from diverse sources: self-description metadata is given by the module supplier and describes physical characteristics and skills the module provides, coordination metadata is built up at run-time by the modules, whereas performance metadata comes from sensors operating with the module.

Preliminary policies are identified, some specific to a certain module, or a type of module, others related to module
coalitions, or to the entire system. For more details about policies for SO-EAS see [12].

Besides adapting the MetaSelf architecture to the needs of SO-EAS, as well as defining policies and metadata, we extended the design abstraction for the formation of MRA coalitions. We considered the tiles model of crystal growth [13], where tiles progressively self-assemble according to the specifications of their edges, in order to calculate the solution of a mathematical function. In the SO-EAS analogy, the tiles are MRAs and the mathematical function corresponds to the assembly requirements and constraints of the product.

3) Output: We gained a better understanding of the SO-EAS, identified the self-* requirements, and defined four phases in the life of an SO-EAS (Figure 4 (1)):

- **Design time**: the SO-EAS architecture is being designed and developed by a software architect. This encompasses the selection of self-organising algorithms, determination of policies, metadata and their implementation.
- **Run-time**: the SO-EAS is running (executing). This encompasses creation of the shopfloor layout (see creation time below) and building of product items (see production time below).
- **Creation time**: a phase of the run-time when an EAS layout is being built. A layout specifies where each module, taking part in the assembly system, is physically positioned. We identified the following self-* requirement: the layout is produced as a result of a self-organising process among the MRAs.
- **Production time**: a phase of the run-time when an SO-EAS is building product items (software and mechanical modules are running). We identified the following self-* requirements in this phase: decentralised task coordination among MRAs and self-adaptation at production conditions, i.e. employing self-managing policies allowing the assembly system to overcome faults or to continue production possibly with degraded performances.

This state of our work is presented in [1], [12].

As a consequence of this work, the MetaSelf architecture evolved into a development method. The MetaSelf method proposes a development process in three phases (Figure 5 (1)):

- **Requirement and analysis**: identify the functionality of the system along with self-* requirements specifying where and when self-organisation is needed or desired.
- **Design** in two steps: (a) the Patterns and Self-* mechanisms decision step: choice of architectural patterns (e.g. autonomic manager or observer/controller architecture [14]) and adaptation mechanism (e.g. trust, gossip, or stigmergy). (b) the System design step: instantiate the chosen patterns for the specific application, architecture and policies, design the individual components (agents), select and describe the necessary metadata.
- **Implementation**: produce the run-time infrastructure supporting metadata, policies and agents.

The MetaSelf architecture and method are reported in [15].

C. Detailed design of layout creation (2)

MetaSelf guides the designer to first analyse the required functionality, and identify what functionality should follow a self-* approach (self-* requirements). In a first design phase, the self-organisation mechanism and the architectural pattern are chosen, and in a second design phase, models including agents, metadata and policies are developed. The implementation of the run-time infrastructure follows.

1) Method: At this stage of our work, we selected a self-* requirement: the self-organising process at creation time for producing the layout. We focused exclusively on the design of self-organising rules supporting this process. The arrival of a product order triggers a self-organising process among MRAs. The self-organising process has to provide (converge towards) a solution where identified MRAs are positioned in a coherent order on a layout so as to satisfy the product order.

2) Application: In order to obtain a truly bottom-up approach where modules spontaneously assemble to fulfill the product order given in input, we decided to replace the tiles mechanism with a design following the Chemical Abstract Machine (CHAM) paradigm [16]. CHAM is a design abstraction consisting of a ‘molecule solution’ and chemical reaction rules. The molecules spontaneously react with each other according to the rules; each time a rule fires, the molecules involved in the reaction are replaced (or ‘rewritten’) by their new composition (the outcome of the reaction). Afterwards, other rules may apply, and the solution is rewritten again, and so forth, until no rule can be applied any more, and the system has converged to a stable state. The rules are applied in a concurrent and distributed way; in other words, the molecules self-assemble or self-organise.

In the case of SO-EAS, the molecule solution is the set of all MRAs, rules are physical (possible) combinations of modules and their provided (simple or composite) skills matching tasks specified in the product order. The insertion of a product order into the solution triggers the reaction rules. As a result, MRAs form coalitions, according to their compatibility rules and composition pattern (compatible sizes and shapes, combination of simple skills providing composite skills), and annonce themselves for fulfilling some task specified in the product order. This design is more general than the previous design with tiles.

3) Output: We now have a clear design of the system’s behaviour at creation time, we identified the corresponding self-organisation rules (chemical reactions as described
above). Additionally, we also intend to exploit the rewrite process provided by CHAM in two ways: 1. for proving that the assembly system obtained as a result of this process is actually able to fulfil the product order provided in input; 2. as a result of the rewriting process we also obtain a detailed movement of the modules (i.e. micro-instruction of each module).

In order to formally establish these rules, a simulation of this design with K-maude\(^3\) is under construction (Figure 4 (2)).

Regarding the MetaSelf method, this work showed that for the self-* mechanisms to be correctly designed, simulations of these mechanisms should be integrated in this phase as well. This allows the correct rules to be defined at an abstract level before the implementation starts. A simulation phase within the concrete design (Design 2) now complements the method (Figure 5 (2)).

### D. Detailed design of production time activities (3)

This stage is concerned with the self-* requirements at production time, keeping in mind industrial standards such as traceability and transparency.

1) **Method:** Following the MetaSelf method, at this stage (Design 2) we focus on the design of the remaining self-* requirements identified: decentralised coordination of tasks and self-adaptation at production time.

2) **Application:** We developed an ontology describing the different modules characteristics and their relations, we also provided workflows specifying product orders. Self-describing metadata (provided by industrial catalogues) is integrated into the ontology. Metadata supporting decentralised coordination of tasks is provided by an RFID tag attached to each product being assembled. It describes the current status of the product. Modules take advantage of this information when a product reaches their workspace in order to decide what action to perform on the product. Performance metadata is provided by sensors; MRAs monitor themselves as well as their neighbours in the shopfloor layout. This requires many sensors, and their readings need to be interpreted by a reasoning service.

Decentralised coordination occurs through information stored in the RFID attached to each product being assembled. The RFID is also useful for traceability purposes. Self-adaptation to production conditions is obtained through a series of if-then-else policies applying to individual modules, dynamic coalitions or the whole system. We still need to solve issues related to priority and conflicts among policies (Figure 4 (3)).

3) **Output:** The main outputs for the SO-EAS are the establishment of the ontology, the workflows product orders and the if-then-else policies.

\(^3\)http://fsl.cs.uiuc.edu/index.php/K: A_Rewriting-Based_Framework_for_Computations_-_Preliminary_version

<table>
<thead>
<tr>
<th>Norms</th>
<th>Time</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRAs</td>
<td>always</td>
<td>self-organisation</td>
</tr>
<tr>
<td>Policies</td>
<td>production time punctual</td>
<td>self-management</td>
</tr>
</tbody>
</table>

Table I

NORMS, RULES AND POLICIES

Additionally, we understood the distinction between norms, rules and policies, as summarised in Table I.

- Norms are generic, and all entities should observe them. Norms lay the foundations for a self-* system to exist. For instance, it is a norm that agents must play fair and may not defect.
- Generally, rules are used for creation time activities such as formation of the layout. MRAs continuously follow the rules in order to aggregate to each other.
- Policies can be both universal (applying to all agents, e.g. no robots can move outside the general workspace), or individual (applying to a class of agents or to a specific agent only, e.g. specific gripper can only with specific robot). Policies are mainly active at production time and apply punctually: as long as everything goes fine, no actions need to be taken, but when something goes wrong, policies define further steps.

Finally, the impact on the MetaSelf method is as follows: policies are now characterised more finely. The policies related to self-organisation and applying continuously are now called rules, while those related to self-management, applying punctually (to recover from faults) are called policies (Figure 5 (3)).

### E. Implementation (4)

At this stage we are developing the run-time infrastructure.

1) **Method:** According to the MetaSelf method, we are now in the implementation phase. We need to develop/identify the necessary tools and software for handling metadata, for programming the MRAs and for converting rules and policies designed in the previous stages into executable ones.

2) **Application:** As said above, we are developing the MRAs, more precisely the software agent part of the modules; translating the CHAM reactions rules into an executable program involving the agents; implementing the self-managing policies and providing a working prototype with real modules.

We are also working on a reasoning engine, running at different levels (single MRA, coalition, whole system) that enforces rules and policies on the basis of actual metadata values.
3) Output: We are still in the implementation phase. A complete prototype will include the above described elements (ontology, reasoning engines, implemented agents, etc.) (Figure 4 (4)). The MetaSelf method is further refined and now includes a verification step (Figure 5 (4)). It consists in the identification of potential problems in a similar way as in the Failure Mode and Effects Analysis\(^4\) (FMEA), which is a well known method in mechanical and electrical engineering. A self-organising or self-adaptive system is considered composed of active autonomous agents evolving into an environment, handling passive artefacts, and working according to some self-* mechanism. In order to determine potential faults in the system, the verification step consists in identifying faults that can arise in each of these elements (e.g. an error in the design of the self-organising rules, a malicious agent, or a faulty environment) and determine how they have an impact on the system as a whole and how the system overcomes (or not) these faults. A discussion on this topic is reported in [17].

IV. LESSONS LEARNED

We have been learning as we progress. Our experience helped gain insight on three levels: better understanding of the application we want to develop and of different design possibilities; better understanding of a development method for self-organising and self-managing systems; and finally, trade-off between design and implementation issues.

A. Understanding the system

Working on the analysis and requirements of SO-EAS helped us understand our system much better. In particular, it helped identifying the different phases (design time, run time, creation time, production time). This is a direct result of the need to align the different jargons from two different disciplines (manufacturing and computer science), where same terms have different meanings.

B. Design versus implementation and prototypes

Sometimes the concepts for how a system shall work are relatively simple, but the tools at hand bring complications. For instance, SO-EAS require a simple reasoning service to go through the policies and apply those which are suitable. Finding a tool which can do this is not easy; JESS for instance can do it but leads to many other issues, such as the sequential operation of JESS. Policies, on the other hand, are concurrent and may be contradictory.

Another aspect is that languages and tools, such as policy specification languages for instance, are often weakly documented, not updated, or not supported any more. This means that instead of choosing the best suitable language, designers may be forced to let secondary criteria lead their choice.

When working with EAS, we always make the assumption that the mechanical side has been taken care of, i.e. the modules are available and running. However, when trying to create real prototypes with industrial modules to try out the self-organisation, the lack of readily available modules has proven to be a serious limitation. Full or partial simulation (that is, working with real modules and simulate production as well as module failures) is an alternative, but also implies a certain additional effort (e.g. buying simulation tools for industrial robotics, setting up simulations with many parameters, etc).

C. Reasoning engine: theory and practice

In theory, it seems clear what kind of reasoning engine we need. In practice, many aspects which nobody has thought of before become apparent: does the reasoning engine run through all the policies serially? Will there be one central engine for all the policies and agents, or will each agent run its own engine? How to deal with policies applying to individual modules part of a coalition?

D. Distributed or not?

A distributed / decentralised concept can be implemented in a non-distributed / centralised way, and vice-versa. For instance the concept of policies is intrinsically distributed: many agents consider many policies for guidance in their behaviour. Some policies are very specific and individual, whereas others are of more global or general scope. When it comes to their implementation, however, the policy enforcement engine may have to be centralised - simply because distributing the engines would lead to too many additional issues (among others performance). Having an engine running for each agent, each coalition, each relevant subsystem, and the system as a whole, means a considerable computational effort. But the even more important issue is probably the concurrence and conflicts among the policies: what if one enforcement engine tells an agent to do something, while exactly at the same moment, another engine says the contrary? Who wins? Defining priorities may solve the issue, but how do we predict all possible conflicts?

E. Micro versus macro

Self-organising phenomena are mostly created out of local interactions at the micro level, whereas the created phenomena appear at the macro level. As this transition is hardly fully traceable, it is very difficult to prove that the macro level system satisfies the requirements. This means that the designer needs to show that something good will eventually happen (e.g. liveness properties), that bad things never occur (e.g. safety properties), and that some characteristics are always assured (invariants). Until now, a generally applicable method to prove such qualities does not exist.

\(^4\)http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis
F. Industrial acceptance of self-* properties

For self-* systems to be eventually accepted by industry, they must show sufficient evidence of dependability and safety. In a first step, this means that we must assure that the self-* rules work correctly at the abstract level (e.g. through simulations). In a second step, it is necessary to assure that the rules and policies also function correctly at run-time (verification).

V. Conclusion

This article reported the experiences gained from the realisation of self-organising evolvable assembly systems, which are distributed complex systems. The robotic modules collectively achieve a goal which the individual modules would never have been able to do.

While developing SO-EAS and applying the MetaSelf architecture, a design method to accompany the MetaSelf architecture was elaborated and step-by-step further refined. As we are still working on developing SO-EAS, we expect further changes in the system. Similarly, the MetaSelf architecture and design method will also gain further sophistication.

It has become clear that the abstract design of such systems is much easier than the actual implementation. Tools and languages bring additional complications which have to be dealt with, and it is rare that one finds a tool or language which exactly suits the needs. Especially the more 'exotic' tools and languages suffer from a lack of documentation, updates and support. This may lead to the designer choosing a less suitable but better serviced tool or language.

The issue of creating local interactions and using local mechanisms for creating global phenomena still requires a lot of investigation. Formal methods to prove the systems’ dependability are required, as the experimental approach is rarely sufficient and viable with complex distributed systems.

Our current and future work includes the creation of the mechanisms for self-adaptation at production time, ensuring that SO-EAS can run production as autonomously as possible. We will also further elaborate on a formal model, which will then allow us to prove system properties (liveness, safety and invariants). A policy specification language is currently being developed, and the actual policies are being refined. One of our next projects will focus on the reasoning engine, which is responsible for treating contradictory and conflicting policies.

Acknowledgments

One author receives a PhD grant from a National Foundation for Science and Technology. The authors also thank the EU-funded coordination action Perada for financially supporting travel exchange between authors’ institutions.

References


