Designing Self-Organisation for Evolvable Assembly Systems

Regina Frei
New University of Lisbon
Portugal
regina.frei@uninova.pt

Giovanna Di Marzo Serugendo
Birkbeck College
University of London, UK
dimarzo@dcs.bbk.ac.uk

Jose Barata
New University of Lisbon
Portugal
jab@uninova.pt

Abstract

Current solutions for industrial manufacturing assembly systems do not suit the needs of mass customization industry, which is facing low production volumes, many variants and rapidly changing conditions. This technical report explains in detail the concept of Self-Organising Evolvable Assembly Systems, where assembly system modules and product parts to be assembled self-organise and self-adapt (among others, choose their coalition partners, their location and monitor themselves) in order to easily and quickly produce a new or reconfigured assembly system each time a new product order arrives or each time a failure or weakness arises in the current assembly system. This report presents in detail the design of such a system following an architecture for self-organising and self-adaptive systems based on policies enforced at run-time on the basis of collected and updated metadata. As a case study, the assembly of a adhesive tape roller dispenser is considered.

1 Challenges in manufacturing

In the era of mass production, companies needed optimal solutions to produce large quantities of an identical product as fast and as cheap as possible; it was worth paying the big investments for custom-made installations, which would be disposed of once the product is out of production. In the best case some of the equipment could be reused but needed manual re-programming. Any change, even small, in the product design or any failure of a module implied halting the production, changing and re-programming the system, which is a work- and time-intensive as well as error-prone procedure.

Nowadays, the market tends increasingly towards mass customisation, meaning that clients like to individually compose their product from many options. Companies require high responsiveness and the ability to cope with a multitude of conditions such as small lot sizes and assembly-to-order.

Due to the high salaries in the Western Hemisphere, companies often have to off-shore manual work to low-wage countries and run the risk of not only losing jobs but also knowledge and finally the entire business. The only alternative is automation. Robotic assembly systems must become able to cope with such dynamic production conditions: frequent changes, low volumes and many variants. Systems need to be quickly reconfigurable, following a Plug&Play approach and avoiding time- and work-intensive re-programming. Evolvable Assembly Systems (EAS) and Self-Organising Evolvable Assembly Systems (SO-EAS) specifically target these challenges and go beyond.

1.1 Goal

Our goal is to develop self-organising evolvable assembly systems, i.e. given a specified product order provided in input, the system's modules spontaneously select each other (preferred partners) and their position in the assembly system layout. They also program themselves (micro-instructions for robots movements). The result of this self-organising process is a new or reconfigured assembly system that will assemble the ordered product. In the self-organising jargon, the appropriate assembly system emerges from the self-organisation process going on among the different modules in the assembly system. Using self-organisation in real engineering problems also

requires some flexibility on the definitions - the concept can not always be applied in its purest form as know in natural sciences.

Any new product order triggers the self-organising process, which will lead to a new appropriate system. The self-organisation process does not stop at the layout formation. During production time, whenever a failure or weakness occurs in one or more of the current elements of the system, the self-organisation 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 current failure, eventually degrading performance but maintaining functionality; or may decide to trigger a re-configuration leading to a repaired system. The actual decision will depend on the situation at hand and on specific production constraints (cost/speed/precision).

1.2 Organisation of this report

This report is organised as follows: Section 2 and 3 respectively introduce the notion of Evolvable Assembly Systems and an illustrative case study: the assembly of an adhesive tape roller dispenser. Section 4 briefly presents MetaSelf. Section 5 explains how MetaSelf is applied to EAS, while a design analysis is started in Section 6. Open issues are dicussed in section 7. Section 7.1 reports implementation aspects.Conclusions and an outlook are provided in Section 8.

Remark: a short version [6] of this report was published at SASO 08, the IEEE International Conference on Self-Adaptive and Self-Organising Systems.

2 Evolvable Assembly Systems and Self-Organising Evolvable Assembly Systems

An **assembly system** is an industrial installation able to receive parts and join them in a coherent way to form the final product. It consists of a set of equipment items (**modules**) such as conveyors, pallets, simple robotic axes for translation and rotation as well as more sophisticated industrial robots, grippers, sensors of various types, etc.

An **Evolvable Assembly System (EAS)** is an assembly system which can co-evolve together with the product and the processes; it can easily undergo small and big changes and seamlessly integrates new modules independently from their brand or model. Modules carry tiny controllers for local intelligence. Thanks to wrappers, every module is an agent, forming a homogeneous society with the others, despite their original heterogeneity (nature, type and vendor). Each module is carrying self-knowledge information about its physical reality, especially its workspace (the portion of the space that the module uses when in action / that is accessible by the module), its interfaces and its skills (the capabilities of the module). Several modules together can dynamically form a coalition in order to offer complex skills. EAS are based on a multi-agent control solution called CoBASA [1], which is operational at Uninova, Portugal.

A **Self-Organising Evolvable Assembly System** is an EAS with two additional characteristics: 1) modules self-organise to produce an appropriate layout for the assembly and 2) the assembly system as a whole self-adapts to production conditions.

2.1 Design time, run-time, creation time and production time

The terms *design time* and *run-time* have different meanings in computer science and in manufacturing engineering. In computer science, design time corresponds to the period when a software system is being designed (and possibly developed). During this phase the software architect thinks at the best solution (architecture/algorithm) for the project. Run-time is then the period when the software is deployed and executing. In manufacturing engineering, design time corresponds to the period when the layout of the system is (manually) created (robots are chosen, assembled and programmed), while run-time corresponds to the period when the assembly system is actually building product items. In the context of our work, we have four different periods to discern, and therefore we provide the following definitions:

- **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, etc. and their implementation.

- **Run-time:** the SO-EAS is running (executing). This encompasses creation of the layout (*creation time* below) and building of product items (*production time* below).
- **Creation time:** a phase of the run-time when an EAS layout is being built by the software.
- **Production time:** a phase of the run-time when an SO-EAS is building product items (software and mechanical modules are running).

2.2 Modules

Basic module types which are needed for executing assembly operations are as follows (see Figure 4(a) for illustration):

An **axis** is a module which can execute a movement along or around a certain direction (axis). Its workspace is thus linear or circular. If combined with other linear axis, their combined workspace can be square, or cylindrical in case of a rotational axis. More complex combinations / industrial robots like a Delta, ABB or Scara robot obviously have more complex workspaces.

A **gripper** is a device which is mounted on an axis and allows to grab a part, either with its fingers (mostly 2-3 of them), by aspirating it or by activating an electric magnet.

A **feeder** is a device which receives the parts to be assembled and puts them at disposal of the respective modules which will treat them. For instance, tape rolls are contained in a tube and pushed upwards, where a robot can grip them. In case of screws, they are put into a vibrating bowl (Fig 4(a) shows one of them, feeding screws), which - due to the vibrations - delivers them well-aligned on a rail, where a robot can pick them.

A **conveyor** is a typically linear transportation device consisting of several modules which can be arranged to move work-piece carriers, or sometimes loose parts. Other instances of conveyor modules are corner units and T-junctions.

The modules, even though actively transporting parts while assembling product parts, cannot physically move themselves without human assistance from one **location** to another across the assembly system. In SO-EAS, modules ask a human operator to move them. In this sense we mention *assisted self-assembly*.

2.3 Product, process and systems

Figure 1 shows the mutual interrelations between product, processes and systems. Every **product** has its own specific characteristics (size, form, etc) but also integrates into a product class, which stands for a certain type of product (e.g. mechanical luxury watches). Typically, products belonging to a certain product class require a certain limited set of processes, where a **process** means a coherent suite of assembly operations which lead to the desired result (i.e. the finished product). The equipment which will execute these processes is called the assembly **system**. It is composed of modules, which have certain capabilities, which are called skills.

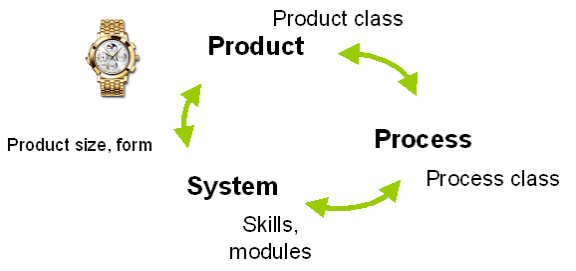


Figure 1. Product, Process and System

The main point is that 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 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. This is one of

the bases of Evolvable Assembly Systems (EAS) [8]: the assembly systems must be made to co-evolve with the processes used and the products to be made.

2.4 EAS and self-organisation

Existing EAS are not yet fully self-organising; the agents still require some manual configuration, and the user creates the layout. Applying an architecture for self-organisation and self-adaptation is a step towards realizing true self-organising evolvable assembly systems.

MetaSelf [5] specifically addresses systems working under dynamically changing conditions. It encompasses coordination of work among autonomous components, seamless addition or removal of components at any time, and coping with failures of various types, while still maintaining a predictable level of dependability. These are exactly the characteristics manufacturing systems need to cope with in dynamic production situations.

2.5 Agents

An EAS / SO-EAS is composed of a set of agents. Each of them can be instantiated as often as required and according to the actual item to be represented.

- **Order Agent, OA**: A product order comes into the system as an OA, asking for a certain number of instances of a specific product to be assembled within a certain deadline. An OA carries the **Generic Assembly Plan, GAP** (see section 2.6), specifying in a general way how to assemble which parts. The GAP stays fixed, even in case of changes in the layout; only changes in the product design itself lead to changes in the GAP.
- **Product Agent, PA**: PAs represent the instances of the product to be made. They are launched by an OA and carry the **Layout-Specific Assembly Instructions, LSAI** (see section 2.7). Each PA exists until its product is finished. As the product assembly progresses along the assembly system, the PA requests the appropriate services from the various MRAs in order to perform the different steps on the assembly plan. PAs are associated with the RFID on the work-piece carriers.
- **Manufacturing Resource Agent, MRA**: An MRA is an agentified module that provides simple skills and has the capability to participate in coalitions. Besides being robotic modules, MRAs can also incorporate the following items:
  - **Conveyor Agent, ConvA**, which transports material between locations in the system, mostly with the help of Work-piece Carrier Agents (see below).
  - **Feeder Agent, FA**, which feeds parts into the system using any kind of device (vibration feeding, tubes, rails, pallets or even manual delivery).
  - **Palette Storage Agent, PSA**, which is used if there are feeder pallets which can circulate in the transport system.
  - **Buffer Agent, BufA**, which provides storage facilities for carriers.
- **Dynamic Coalition Agent, DCA**: MRAs can form coalitions to provide composite skills. Each coalition is represented by a DCA. Notice that a provided simple or composite skill is called a *service*.
- **Work-Piece Carrier Agent, WPCA**: Each WPCA carries a product on the conveyors, along the assembly system, and allows it to reach the MRAs executing the desired operations.
- **Part agents, PartA**: Product parts, represented by PartAs, are delivered by the feeders and need to travel to their target position in the final assembly. There is one part agent per part type (not one per part). PartAs collaborate with FAs to organise the delivery of the parts.
- The **Ontology Agent, OntA**: The OntA provides controlled access to the ontology, offering the capability of filtering the list of possible agents that have a certain skill by the use of specific criteria.

2.6 Generic Assembly Plan

The Generic Assembly Plan (GAP) specifies the way a product is to be assembled: it includes the assembly order of the different parts and the way they must be joined.

An example of a GAP is: *Pick part 1 and place it on the work-piece carrier, then pick part 2 and insert it into the hole in part 1 by applying a force x.*

The generic assembly plan does not provide information about what module to use and what movement to make. It only provides information about how the different product parts must be assembled and in which order. In other words, the GAP says *what* to do but not *how* and is thus independent from any layout.

2.7 Layout-Specific Assembly Instructions

For realising the assembly, the GAP needs to be transformed into **Layout-Specific Assembly Instructions** (LSAI). This is done in collaboration between the OAs and the MRAs; as far as it already exists at this moment, the Dynamic Map (see below) can serve as a support; otherwise, the Dynamic Map is created in parallel with the LSAI, in an interlinked self-organised process (see remark in the next section). The LSAI consist of executable programs for the robotic modules. We also refer to these instructions as "micro-instructions". See Figure 2 for illustration.

For the GAP given above, the LSAI could look like this: *Robot R1 with gripper G1 is at position P1 = (x1/y1/z1) and moves to position P2 (above pallet), opens the gripper (if it was closed before), moves down to P3, closes the gripper, then moves to P4 (location on the work-piece carrier), opens the gripper. Then R1 moves on to P5 (feeder), closes the gripper, moves to P6 (location of insertion) and moves down towards P7 with a force x, then opens the gripper and moves back to P1 and closes the gripper.*

2.8 Directory Facilitator, Dynamic Map, Virtual Reality and Ontology

The **Directory Facilitator** (DF) is a registry where all the agents (including those not selected to be part of the layout) register and may retrieve services from other agents.

The **Dynamic Map** (DMap) is also a kind of registry, but only modules currently being used in the layout register there, and they continuously update their availability. The DMap is thus, as its name says, dynamic and order-specific; each time the layout changes or a module becomes unavailable, the DMap needs to be updated (the corresponding modules are unregistered from the DMap). Visual representation is not mandatory.

Virtual Reality (VR) is a 3D visualisation of the entire layout. It also includes every module's micro-instructions, workspaces and movements simulations. The VR closely follows the DMap status. The interest of the VR is that it provides human operators with a visualisation of the system (especially during creation time) and possibly to interact with it. The VR is to be realised once the other functionalities are implemented.

The **Ontology** includes DF functionality, but goes much further concerning the relations between skills, modules, interfaces, etc. It contains basically all the necessary assembly knowledge which could ever matter.

Remark. The DMap is incrementally built according to a self-organising process (see section 5.9). It is not clear yet whether the micro-instructions are issued simultaneously with the DMap (within the same self-organising process) or if they are issued in a subsequent step, once the DMap (i.e. the layout) is fully completed. See Figure 2 for illustration of this open issue.

2.9 Global view

Figure 3 shows the system as a whole. At the core of the system at creation time is a self-organised incremental and iterative process, controlled by creation time policies and supported by the ontology. The creation time process receives as input:

- the GAP,
- all the available modules as registered in the DF, and
- user preferences.

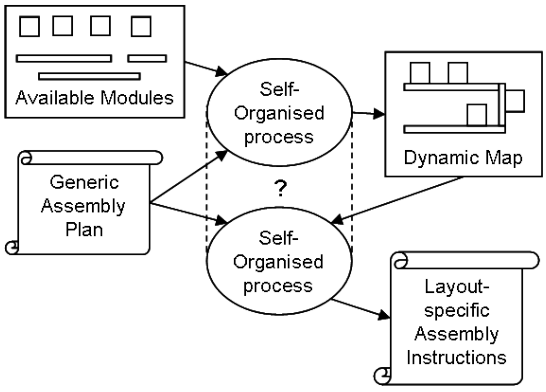


Figure 2. Overview: creation of the layout and generation of the micro-instructions

The output of this process is:

- the DMap as well as the real layout (to be built by the user according to the specifications given by the self-organised process),
- possibly a representation of the layout in virtual reality, and
- the LSAI, which allows the execution of the assembly in reality.

The output of the creation time process is the input of the self-organised and self-adaptive incremental process at production time. This process is controlled by production time policies, and leads to the finished products at the end. If necessary, it also triggers system reconfiguration, which restarts the creation time process, taking as additional input the information about the current state of the system and its problems / failures.

3 The adhesive tape roller dispenser assembly system

To illustrate the concept os SO-EAS, we have chosen a simple product: an adhesive tape roller dispenser (see Figure 4(b)) consisting of two body parts (Parts 1 and 3) locked by a screw (Part 4) and the tape roll (Part 2). In the remainder of this paper we will refer to this assembly system as the *tape roller dispenser*. The assembly is made on top of a work-piece carrier circulating on the conveyors.

For reasons of simplicity, the choice of system modules, as illustrated in Figure 4(a), will for now be very limited: a Z-axis moving in a vertical direction; an X-axis working horizontally; a feeder receiving screws in bulk. In reality, modules which are available at other places in the system, in the storage or even in the system supplier's catalogue can be considered for joining the coalitions on request. Figure 5(a) shows a combination of the two robotic axes introduced in Figure 4(a), with additionally a 2-finger gripper mounted on the Z-axis. This configuration can be used for executing all the movements required to assemble the tape roller dispenser. Figure 5(b) shows a robot placing part 1 on the conveyor passing inside its workspace.

The self-organisation process could produce many different layouts and choose one depending on user preferences and available modules. For this case study, we consider that a layout with a circle such as shown on Figure 6(b) is produced, with all the robots being identical and as shown on Figure 5(a).

This layout has some interesting characteristics: thanks to its circular structure, work-piece carriers re-visit Robot 1, which executes two different work-steps. In this example we assume that Robot 1 works faster than Robots 2 and 4, and thus assembles Part1 as well as Part3. The letters A, B, C and D on Figure 6(b) refer to the respective feeding of the parts 1, 2, 3 and 4, while *IN* represents the entry of the empty work-piece carriers and *OUT* means that the carrier with the finished product leaves the system. The screw (part 4) is fed by a regular feeder such as on Figure 4(a) and picked up as well as transported by a magnetic screw driver. The tape roll is fed from a tube and the gripper takes each roll from its inside part.

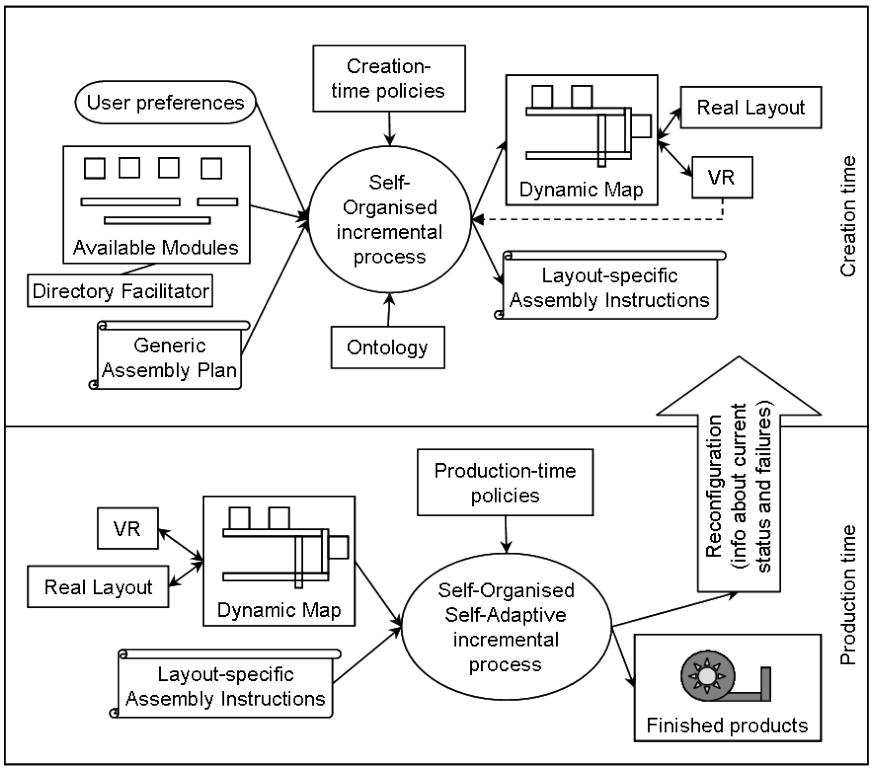


Figure 3. An overview of the system including most relevant components

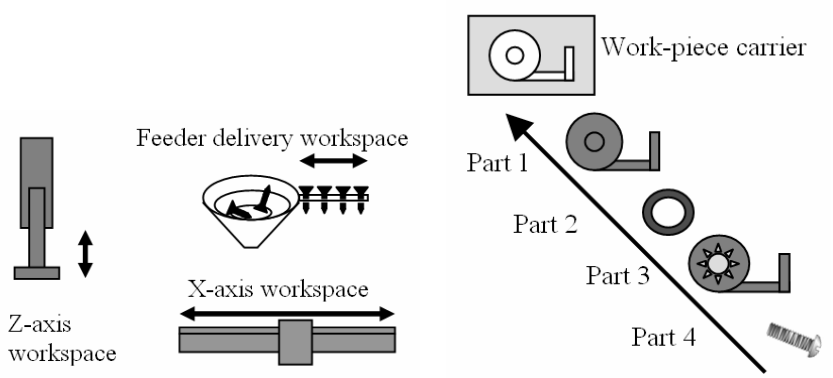
Figure 6(a) is an illustration of how the same gripper can be used in two different ways to cope with different part conditioning. This is a simple modification of the way a gripper is applied; changes in the way parts are fed into the system do not always necessarily lead to changes in the layout - sometimes a little modification in the programming is enough. The agents can do this autonomously by triggering a reconfiguration process with a minimal-effort preference.

The agents in this case study are:

- An *OA*, which carries the GAP for the tape roller dispenser.
- *PAs*, which carry the LSAI.
- *MRAs*:
  - 5 linear conveyor units, 2 corner units and 1 T-junction unit,
  - Robots R1, R2 and R4: each needs 1 x-axis, 1 z-axis and 1 gripper (G1, G2, G4) with 2 fingers,
  - 1 stick or tube feeder (tape rolls - B),
  - 1 vibrating bowl feeder (screws -),
  - 2 pallet feeders (1st and 2nd body part - A, C),
  - work-piece carriers,
- *PartAs*, which represent Part 1 to Part 4.

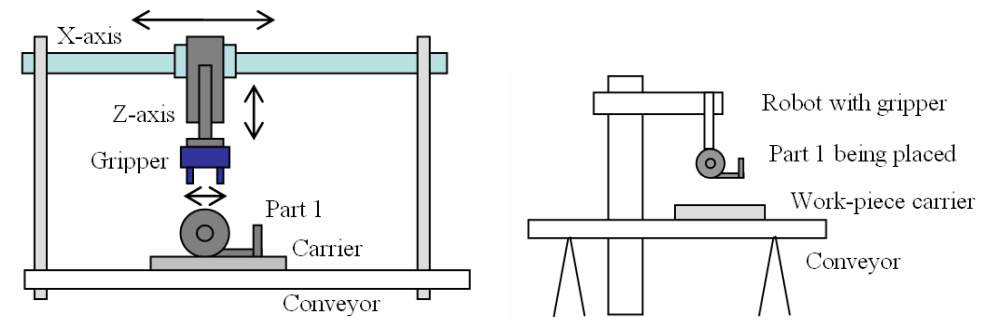
3.1 Scenarios

We will consider the following scenarios to trigger a self-organisation process in our assembly system:



(a) Selection of modules (b) The adhesive tape roller dispenser

Figure 4



(a) A simple robot made of 2 axes and 1 gripper and part 1 placed on a work-piece carrier circulating on the conveyor (b) A robot placing part 1 on a work-piece carrier circulating on the conveyor which passes inside its workspace

Figure 5

New product order: carried by an OA, it triggers the building of the corresponding layout as a result of a self-organising process. (User preferences considered: circular system, morphologically identical robots with one being faster than the others.)

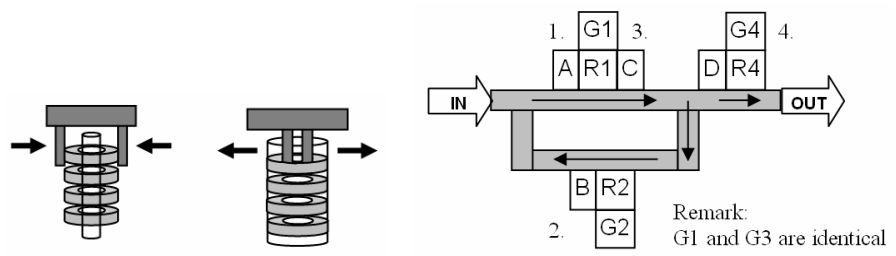
Resilience during production: failure of some modules and re-organisation of the remaining modules for building a repaired or alternative system. (Concrete incident considered: R2 fails; R1 can take over until R2 has been replaced.)

Small change in tape conditioning: sometimes little changes in conditioning or design do not need physical reconfigurations but only small modifications of the micro-instructions. (Concretely: instead of being delivered in a tube, the tape rollers are delivered on a stick. G3 will thus grab the tape rollers from outside instead of inside.)

Small change in product design followed by the reconfiguration of the layout: some modifications have bigger consequences and effect resource attribution. (Change considered: the screw will be eliminated and replaced by a snap-fit mechanism integrated on Part 3; as a consequence, R4 is redundant and R1 (=R3) needs a new gripper which is able to apply a force F in the center of Part 3.)

4 MetaSelf

MetaSelf is a service-oriented architecture for self-organisation and self-adaptation, where the services are provided by components or agents. It exploits *metadata* to support decision-making and adaptation based on the



(a) The same gripper grabbing tape rollers, once fed from a stick (left) and later also once fed from a tube (right).

(b) Circular layout where Robot 1 treats part 1 and part 3

Figure 6

dynamic enforcement of explicitly expressed *policies*. Metadata and policies are themselves managed by appropriate services. The components, the metadata and the policies are all *decoupled* from each other and dynamically updated (or changed).

4.1 Design time activities

Figure 7 represents the design time activities: During the analysis phase, properties of the overall system are identified, and driven by these properties, the designer selects the architectural patterns and adaptation mechanisms that the system will adhere to. Afterwards, the chosen patterns are instantiated for the specific application, architecture and policies. The individual components are designed, and the necessary metadata is selected and described.

4.2 Run-time infrastructure

Figure 8 shows a generic architecture for the run-time infrastructure.

Components are the different agents of the system. In the case of SO-EAS, the components are PAs, OAs, MRAs, PartAs, etc.

Metadata is data about functionality and performance characteristics, as opposed to data which is treated directly by component. Metadata is stored, published and updated at run-time by the run-time infrastructure (monitoring activities) or by the components themselves (sensing/acting). Different types of metadata are available: self-description of the components (possibly including interfaces information and formal specifications), environment related metadata (possibly supporting coordination, e.g. through stigmergy), self-* properties metadata refer to resilience and non-functional metadata related to either individual components or to groups of components (such as level of *fatigue* of a component of the assembly system).

Policies are also available at run-time to both the run-time infrastructure and the components themselves. Policies are subject to dynamic changes if necessary. Policies come in different categories, and apply to different levels (system-level policies vs. component level policies).

Guiding policies stand for both high-level goals the system as a whole has to reach and for individual components goals. Coordination policies refer to rules which components have to adhere to when coordinating their different tasks (for instance scheduling of tasks or overlapping of work spaces).

Bounding policies are intended to prevent the system going beyond its limit: these are limits set by the designer to avoid the system going out of control (too many failures), as well as limits imposed by the environment on the system (e.g. a particular type of modules cannot be placed on the lower left corner of the assembly plan).

Finally, **sensing/monitoring policies** are lower level policies, where individual components react to on-going activities in the system. There is no restriction on the policies, they can be action-, goal- or utility-based policies [9].

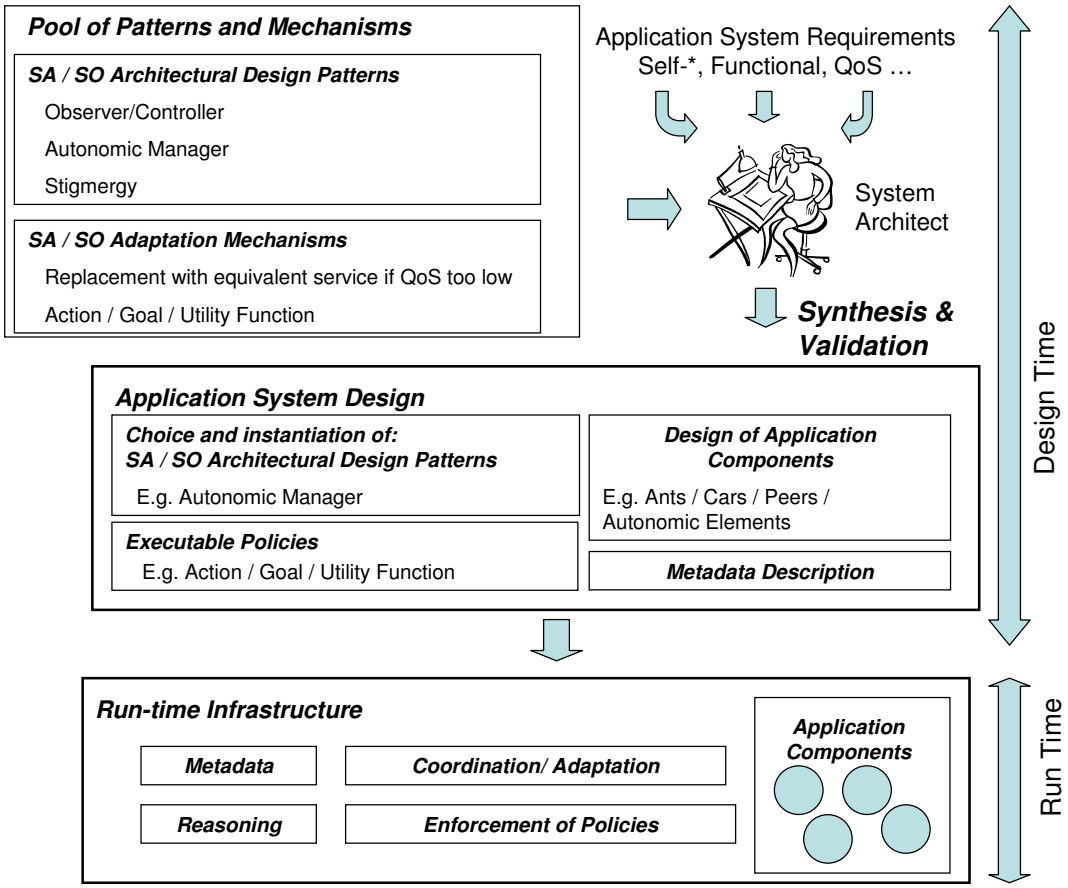


Figure 7. Design time activities and overview of run-time infrastructure

4.3 Enforcement of policies

Low-level policies attached to specific components trigger a reaction in those components based on the corresponding metadata value (for instance, monitoring metadata about position allows to take corrective measures if necessary). The run-time infrastructure itself is equipped with specific services responsible for the enforcement of the policies (given the current metadata values) by directly acting on the components (e.g. replacing, reconfiguring), the metadata values or the policies. It provides different tasks related to the processing of metadata stored in the metadata registry, such as comparison/matching of metadata, determination of equivalent metadata information, composition of metadata; it encompasses automated reasoning over the policies and the metadata.

4.4 Coordination and adaptation.

At design time, the coordination and adaptation service implements the self-adaptive and/or self-organisation pattern chosen for the particular application. For instance, it is responsible to support digital pheromone [2] (concentration, volatility) if this is the chosen adaptation mechanism.

The run-time infrastructure is preferably not centralized. The different services providing access to the description of components, or monitoring and acquisition of metadata can reside at different locations and work autonomously. Metadata and policies have either a local or global scope, and can be locally attached to a compo-

nent. The actual implementation depends on the application. As said before, the main point is that components are decoupled from each other, and that metadata and policies are decoupled from the components code so that anyone of them can be dynamically updated or changed.

Generic services necessary to build such a 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-* related metadata; a registry that handles the policies; a reasoning tool that matches metadata values and policies, and enforces the policies on the basis of metadata.

Metadata is either directly modified by components or indirectly updated through monitoring. Together with the policies, metadata causes the reasoning tool to determine whether or not an action must be taken. The policy enforcement services act on both components and metadata, impacting components both directly and indirectly.

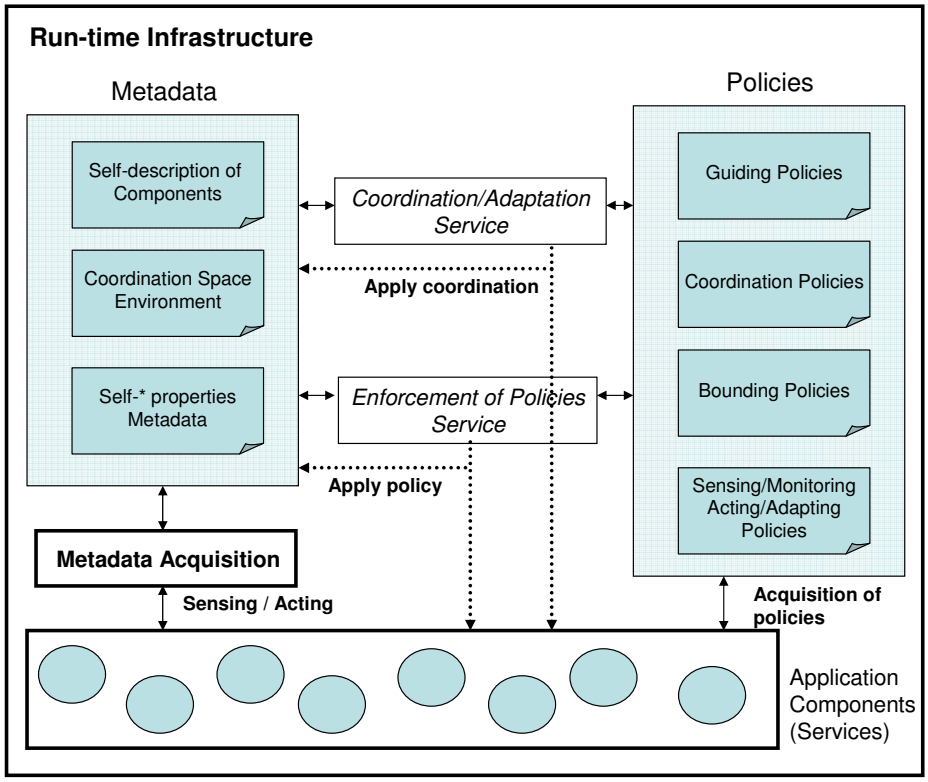


Figure 8. Run-time infrastructure

4.5 Control

Control of the system happens in the following different ways. First, components have to conform to their policies based on the current values of metadata. Each action is subject to control of the policy and the corresponding metadata. Second, the Reasoning and Enforcement of Policies services directly act on components by dynamically reconfiguring them, allocating more components, removing faulty ones, etc. Third, an indirect action is performed by modifying the metadata used by the components to sense their environment. This is a technique used for (externally) controlling self-organising system working with stigmergy. Fourth, an additional way of controlling the system consists of modifying the policies used by the components for driving their behavior on-the-fly. Policies

are decoupled from the components even if they are locally attached to them: changing the policies will immediately affect the corresponding component. Even though the control shown on Figure 9 is internal to the system, external control is applied in the same way, by acting directly on the components, metadata or policies.

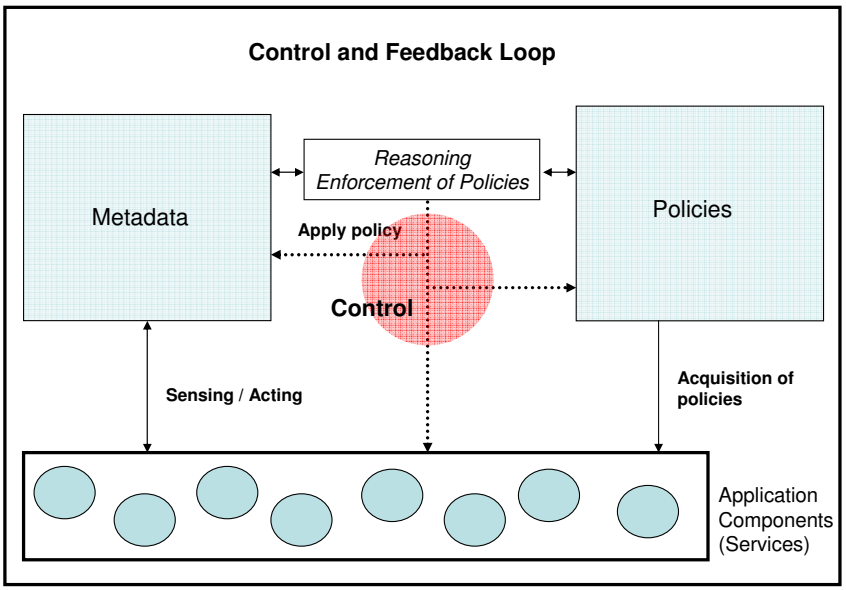


Figure 9. Control

4.6 Self-organisation and self-adaptation in the case of EAS

All the components register their services, skills and constraints *(self-description metadata)*. They have access to global or local *coordination metadata* (e.g. assembly status of current product item) and *resilience metadata* (e.g. current level of precision or speed of a module itself or of a partner module). Policies that the system as a whole has to adhere to are global to all components. For instance:

- Fulfil the generic assembly plan.
- No MRA is allowed to move outside the allowed global workspace.
- The user favours a circular layout.

Examples of policies which are locally attached to individual components are:

- Avoid collisions.
- Adapt your own speed to the speed of your partner.

Self-organisation is mainly related to the creation of a new layout (a new organisation of the resource agents) when a new GAP is given or when a failure has occurred. This happens bottom-up, following the *tiles model* [10] (detailed later).

Self-adaptation is related to production time (adaptation of speeds, tasks coordination and collision avoidance) and leads to self-organisation when it triggers a new re-configuration.

See [5] for a more detailed discussion of this architecture, in particular how it allows control to be inserted in the system, and how it unifies self-adaptation (top-down: evaluate the global system behaviour and adapt the agents' behaviour accordingly) and self-organisation (bottom-up: local rules lead to a global result) designs.

5 Application of MetaSelf to EAS

The architecture described in section 4, when applied to EAS, requires at design time the determination of components, self-* requirements, a self-organisation and/or self-adaptation mechanism and the choice of a coordination mechanism. On the basis of these choices, corresponding metadata and policies are derived. This section highlights these choices when designing a SO-EAS and illustrates them in the specific case of the tape roller dispenser.

5.1 Components

As already mentioned, the components are: order agents, product agents, manufacturing resource agents (including those in store) and product parts. See section 3 for more details about these components.

5.2 Self-* requirements

The self-* requirements of an SO-EAS are requirements which are to be achieved in an autonomic way be the agents.

5.2.1 Layout formation

Any new assembly plan triggers a self-organisation process leading to a new configuration (layout). This encompasses self-selection of modules and their partners, and establishment of the layout-specific assembly instructions (micro-instructions).

5.2.2 Task coordination at production time

- *Tasks sequencing* is done according to the LSAI. Modules coordinate their work according to the current status of the product being built.
- *Collision avoidance* between the modules is also a fundamental part of the self-organising process. Modules with overlapping workspace must maintain a minimum distance to each other while moving.

5.2.3 Self-adaptation

- *Self-healing* during production means that whenever a module failure is detected, by the module itself or one of its partners, either the modules can solve the problem by themselves (software restart, open and close a gripper, etc.) or they alert the user. A module failure can mean many types of perturbations, for instance a blocked gripper, a software problem, a lost part or anything else.
- *Self-optimization* happens when the speed of processing the product parts is adjusted to the queuing level. Specific policies can trigger other optimisation targets, such as a maximal use of the resources or minimal transportation distances.

C1	New GAP in input
C2	Production
C3	Production completed (stop)
C4	Re-configuration requested

Table 1. Configurations

5.3 Other requirements

In the end, most requirements could be formulated as self-* properties of an autonomic system. Therefore we mention examples of requirements which are somehow 'exterior':

- The system has to produce the requested products correctly, on time and in the right number, respecting all the specifications given.
- All the steps being made by the system have to be traceable in industrial context. Each agent documents its actions in a log-file.
- Safety for users has to be granted. Robots have mechanical well as software mechanisms for avoiding undue forces.
- Waste (broken parts or non-functional products) should be avoided.
- Great attention must be given to user-friendly interfaces, avoiding programming as far as possible.
- Compatibility of equipment from different suppliers and legacy equipment is primordial; granted through the use of software wrappers as described in CoBASA [1].

5.4 Mechanisms

Self-organisation mechanism (1): The overall mechanism chosen for letting the different agents perform the appropriate tasks at the different phases of the production (from receiving the product order to delivering the finished product) is based on *stigmergy by work-in-progress*, also called *qualitative stigmergy* [2]. The different configuration states (Ci in Table 1) of the system trigger different responses from the agents; they represent the different phases of production.

Self-organisation mechanism (2): The mechanism chosen for letting the different modules and product parts re-organise whenever a new GAP arrives, or when a module change is requested, is inspired by the tiles self-assembly model of crystal growth. In this model, tiles progressively attach to each other following matching rules and current configuration of the structure to build [10]. This is used in configurations C1 and C4. In [3], the tiles self-assembly model is used to calculate the result of a mathematical function; analogously, in EAS, the tiles (agents) self-assemble to provide a solution (a layout - DMap), to the given function (GAP).

Self-adaptation mechanism: Self-adaptation in C2, e.g. resilience to failures and collision avoidance, is performed by modules monitoring their own behaviour or their neighbour's behaviour.

Coordination mechanism: During production (C2), the coordination of tasks for each individual product item is done through indirect communication by storing the current advancement of the assembly in a shared place - a RFID attached to the WPCA.

Metadata and policies: We concentrate here on describing the metadata and policies related to the three categories of self-* requirements described above. These are design time policies; they need to be turned into executable policies at run-time. Figure 10 shows how the policies and metadata relate to each other and to the use cases described in subsection 3.1. Their development is further detailed in subsections 5.9 to 5.11.

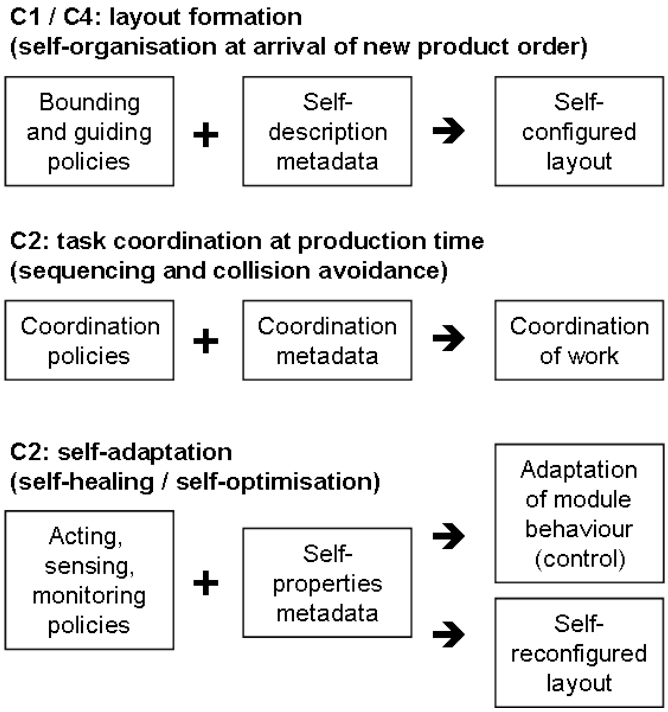


Figure 10. Relations between metadata and policies

5.5 Reconfiguration schemata

Reconfiguration schemata are a type of policy used when creating or changing the layout. They define under which conditions the layout should be built in which way. Once a prototype of the policy-system is running, the reconfiguration schemata need to be tested experimentally; according to the results of the experiments, the set of schemata will be adapted. Examples of reconfiguration schemata are:

- Use the minimal number of modules.
- Work with a module's preferred partners if possible; refuse to collaborate with modules on the *black list*.
- If the product has many variants, then create a layout with several exits (OUT).
- If the product consists of several sub-assemblies, then create a layout with several entries (IN).
- If possible respect the default locations for placing feeders and pallets; otherwise place them close but at a sufficient distance from the robot and close to the location of the product to be treated.
- If the number of requested products is small (threshold T1), then relax constraints on time / throughput.
- If the number of requested products is big (threshold T2) then do not relax constraints on time / throughput but rather adapt the layout to comply with the time constraints.
- If a failure occurs then relax constraints on time / throughput.

5.6 Policy classes and types

Policies can be categorised according to classes and types; any combination is possible, but some of them are more frequent than others. The class refers to the target of the policy, while the type describes the form in which the policy is expressed (which is often a decision of the designer, and another solution could be imagined).

5.6.1 Policy classes

1. Policies for monitoring / sensing / acting
2. Policies for self-organisation / self-adaptation mechanisms
3. Coordination policies
4. Bounding policies
5. Guiding policies: give the possibility for executing external (or internal) control, or some kind of supervision over all the local rules (global view, global results). Guiding policies can also help the system to achieve good / optimal solutions.
6. Agent level policies
7. System level policies

5.6.2 Policy types

The first three types have been adapted from Kephart [9], while the others have been added by ourselves.

1. Action-based policy: defines what to do given the current state of the system or if a certain property on the current state of the system holds ('if (P(current state)) then (...) else (...)'). Such policies are most relevant for situations which occur frequently and where the action to be taken is clear.
2. Goal-based policy: specifies conditions on future states of the system (for instance $s \leq val$) and have binary values (true or false). This type of policies applies when the system must decide what to do from a set of possible actions. An eventual failure to achieve the goal must lead to the emission of an error message.
3. Utility-function-based policy: guides the system towards maximizing/minimizing a certain function (for example $min(\mathcal{F})$); they have decimal values. Utility functions are similar to goals in the sense that they guide the system towards its target, but it is not necessarily an error if the utility-function has a low satisfaction value. The system has to find out by itself what to do to improve.
4. Constraint policy: expresses constraints which have to be respected by the agents, such as 'do not accept a black-list module as your partner' or 'update your log-file after every operation'. If an agent is unable to respect a constraint policy, it must emit a warning.
5. Preference policy: describes preferences given by the user, such as 'build a linear layout'. In case the preference shows to be obstructing, it can be ignored without further consequences.

5.7 Metadata types

Also metadata can be categorised according to different criteria and any combination of them, the list not being exhaustive. Metadata can be:

- functional or non-functional
- value-based or descriptive
- related to resilience
- related to the general functioning of the system
- related to reconfiguration

5.8 Reasoning Service

Reasoning services can be implemented in two different ways: either locally inside every agent, or as middleware.

A typical application of a reasoning service is the dealing with conflicting policies. Does the a global policy have priority over a local one? In which case is it the other way round?

Also the choice of appropriate actions in the context of goal and utility-function policies can be resolved by a resoning service.

Another potential application of reasoning is in the context of evolutionary algorithms, which might in future be used to make systems evolve.

5.9 Layout formation (C1/C4)

This section describes the policies and metadata necessary for the creation of a new layout, or the adaptation of an existing layout to new requirements.

5.9.1 Policies

Guiding policies are high-level policies which mostly take the form of a goal-based policy or a utility-function policy. The GAP acts as a high-level goal and triggers C1.

System-level policies **(P1)**

- Build a continuous path from IN to OUT.
- Favour a layout with a circle; or favour a linear layout.
- Minimize changes in the layout when re-configuring due to failure or change in product design.
- When creating the path through the layout, the workspaces of different module coalitions may not interfere as this might lead to collisions (except for conveyors); inside a coalition however, workspace interference is necessary and the respective actions must be coordinated in time and space (see C2).
- Maximise the through-put.
- Minimise the work-in-progress.
- Maximise the use of all MRAs in the layout.
- If Layout is completed, change configuration to C2.

OA goals **(P2)**

- Fulfil the GAP.
- Produce the specified number of products.
- If the production is finished, trigger configuration C3.
- Minimize the travelling from feeders to target positions on assembly.

MRA goals **(P3)**

- Find a compatible partner (matching skills, interfaces, etc.) to work with in the given GAP. For instance, a robotic axis needs to find a suitable gripper.

PartA goals

- Communicate preferential and forbidden gripping positions.

Bounding policies (P4) limit the modules' behaviour.

- Every MRA must always stay inside the allowed area, defined as the general workspace.
- Grippers cannot be placed at any location except on the axis holding it or in the tool warehouse. (This policy can be gripper-specific or general for the whole layout.)
- Respect speed and force limits.
- Execute agreed tasks; do not defect.

5.9.2 Metadata

Self-description metadata supports the specific application of policies based on module and part characteristics.

- For each MRA: all registered skills, interfaces, workspace and constraints (e.g. preferred ways of combination with other modules, list of preferred partners. For more details see [7]);
- For each product part: its location and the way it is conditioned (prepared for feeding) and grabbed by a gripper.

5.9.3 Tape roller dispenser system scenario

The scenarios help illustrate how the policies are used in reality.

- **Establishment of a circular layout (new product order):** The layout formation is progressively reached through a series of service requests including the needed 3D movements (x/y/z dimensions - micro-instructions). The first request is the one from the OA requesting to fulfil the GAP **(P2)**. Requests are progressively answered by MRAs and respective coalitions **(P3)**. Self-description metadata is matched against the requests taking into account skills, constraints and any system **(P1)** / bounding **(P4)** policy, and current achieved configuration. This corresponds to the matching rules of the tile self-assembly model. In the particular scenario of the tape roller dispenser system, the set of robots at disposal causes R1 to be selected twice for two different tasks favouring thus a circular layout. This happens to be also the primary choice from the corresponding system-level policy **(P1)**.
- **From screw to snap-fit (small design change):** The screw is not needed any more because a snap-fit is now integrated with Part 3. The minimum change policy **(P1)** in case of reconfiguration drives the selection of an additional gripper G3 mounted on R1, able to apply the appropriate force on Part 3. R4 and its gripper are not needed any more.
- **From tube to stick (small change in conditioning):** Again P1 leads to minimal effort and thus only a small change in the micro-instructions. No mechanical reconfiguration is necessary.

5.10 Coordination (C2)

This section reports the policies and metadata which help the system coordinate its actions.

5.10.1 Policies

Task sequencing policies assure the well-ordered execution of the tasks specified in the GAP. A task can start when the WPCA has reached its working position next to the robot. The WPCA can move on as soon as the robot has finished its operation.

PA goals **(P5)**

- Fully satisfy the LSAI.
- If the current operation is finished, request the execution of the next tasks and move to next position in the layout.

PartA goals **(P6)**

- Travel from the feeder to the target position in the assembly layout.

Collision avoidance policies help avoid crashes.
MRA policies **(P7)**

- The distance MRAs must always be bigger than a safety threshold (T3), unless they are members of the same coalition.
- Always stay inside the global workspace.
- If an undue object is detected inside the workspace then stop. If the object remains there for more than 3 seconds, alert the user. Remark: some other modules as well as the parts being treated are allowed inside the respective workspaces of the active modules.

5.10.2 Metadata

Task sequencing metadata supports the application of task sequencing policies.

- For each PA: current status of LSAI execution - the result of any operation is always written into the product's RFID (success, failure, problems, etc.).
- For each MRA: status / availability (idle, reserved, working, failure, in storage). A log file stores the performed operations history (for traceability purposes).
- For each WPCA: Sensors provide the exact position (approaching robot, leaving robot, etc.).
- For each resource agent:

Collision avoidance metadata supports the application of collision avoidance policies.

- For all MRAs: occupancy of its workspace by other MRAs (e.g. feeders), including a list of possible undue items (e.g. a loose screw is lying on the workspace).

5.10.3 Tape roller dispenser system scenario

- **Task sequencing:** A WPCA circulates along the layout, and the product agent notices that it passes twice next to R1. According to its LSAI it asks **(P5)** for the appropriate assembly (Part 1 or Part 3) **(P6)**. Similarly R1 accesses the RFID to read the previous task and to write the result of the current one.
- **Collision avoidance:** The workspaces of R1, R2 and R4 do not interfere in this particular example; no danger of collision between them. Whenever a collision sensor detects something irregular in the workspace (a human hand or a loose screw), the root operation is stopped until the intrusion has gone **(P7)**.

5.11 Self-Adaptation (C2)

The metadata and policies required for self-adaptation are described in this section.

5.11.1 Policies

Self-optimization policies help MRAs improve their performance.
Feeder policies **(P8)**

- Adapt the feeding speed to the part removal speed. (Feeders always need to deliver parts at their output. If the parts are taken away quickly, their feeding speed should be high.)

Conveyor policies **(P9)**

- Achieve the requested throughput (1).
- Respect the specified speed limits (2).

- (2) has priority over (1).

Resource agent policies **(P10)**

- If the queue of PAs is above the queuing threshold (T4), increase the speed of operation.
- If the quality decreases, decrease the operation speed in order to increase the quality.

Self-healing policies help assure the good health of the system. It is necessary to report the state or respectively the result of collaboration with other MRAs. Every agent monitors its own state, especially with reference to critical parameters, and alerts the user in case of problems. E.g., MRAs need to monitor their own precision, and eventually also their neighbour's precision, in order to detect the effects of fatigue or other kinds of disturbances and to take corresponding countermeasures.

MRA - axes and conveyor policies **(P11)**

- If the target position after a movement has not been reached correctly, take corrective measure (advance more / less, ask for maintenance, etc).

MRA policies (general) **(P12)**

- In case of malfunctioning, request the replacement of the failing coalition partner. If no replacement is found, ask for a re-configuration of the layout.
- If the queuing level is too high and the speed is at its allowed maximum, ask for a re-configuration of the layout.
- If a gripper is blocked then 1) open / close the gripper again, 2) restart the software, 3) reset the power, 4) replace the gripper or 5) call the user.

System-level policies (bounding policies) **(P13)**

- If one (or more) failure occur more than a certain number of times in a certain period of time, then alert the user and trigger configuration C4.

5.11.2 Metadata

Self-* metadata supports self-* properties. For each MRA:

- Maximal / optimal speed of operation
- Precision of movements on every MRA's own axes and its partners' (neighbours') axes
- Current queuing level of PAs (input to each MRA)
- Quality of assembled product (any measurable characteristic, or feedback given by the user)

5.11.3 Tape roller dispenser system scenario

- **Self-adaptation:** R2 experiences problems in reaching its target positions and asks for maintenance **(P11)**. As a temporary solution, R1 is asked to take over and the user is asked to place feeder B close to R1 **(P12)**.
- **Re-configuration triggered:** After taking over from R2, R1 experiences a high level of queuing. This leads R1 to ask for a re-configuration **(P1)** (and triggers configuration C4).

6 Design analysis and predictability of dependability properties

Policies (further refined and enforced at run-time) are also a useful tool for analysing the correctness of the software design and proving properties using temporal logic formulae. A first step is to show that a layout will be found. Other examples:

- The safety property 'collisions never happen' is guaranteed by the collision avoidance policies **(P7)** and the selected layout.
- The liveness property 'the specified number of products has eventually been assembled' is a policy by itself (goal of OA - **P2**).
- The invariant 'at all times, the quality of assembled products is above a specified threshold' is provided by the self-optimisation / self-healing policies (decrease speed / increase movement precision - **P11**).

Formal proofs of these properties need formal specifications of the system, the properties, the policies (and their potential conflicts). A mathematical model is currently being elaborated.

7 Open issues

This section discusses various issues which are still to be resolved or which are otherwise worth mentioning.

- **Self-management** is more than self-organisation in the sense that a self-managing system is independent from the user as a supervisor.
- Due to the nature of manufacturing systems and industrial conditions, resilience, predictability and traceability are fundamental. Industrial systems have to perform according to specifications and need to maintain productivity also under perturbations.
- Generally, there are two options for the implementation of mechanisms: **hardcoding mechanisms into the agents** or **using policies** which the agents have to (download and / or) consult before acting. Each option has advantages and drawbacks. Hard-coding is a more local solution, and policies are more difficult to change because it has to be done in every agent separately. Always consulting policies can be time-consuming. The local option might be more favourable for basic mechanisms such as checking the work progress to know the next step, while calculating workspace interferences is a more global issue, which depends on the actual layout; policies are therefore the right choice for the latter. Remark: policies can be generated dynamically and after each change, the agents can download them. For instance, if the layout does geometrically not allow collisions between different robots, anti-collision policies do not need to be generated.
- In a more advanced state of self-organisation, it may be interesting to give the system **more freedom for finding creative solutions**. The less we specify and constrain, the more freedom the system gets. See [4].
- **Learning** could be used in several ways. For instance in layout formation: once a solution is found, it could be remembered, and further self-organisation processes could take profit of it.
- **Norms** are generic rules that all agents should observe. **Policies** encompass both generic rules (applying to all agents, e.g. no robots can move outside the general workspace), and specific ones (applying to a class of agents or to a specific agent only, e.g. specific gripper can only with specific robot) and are thus more interesting than norms. See Table 2.
- It is easier for agents to modify a given layout by self-(re-)organisation (which is close to reality at Uninova) than to build it from scratch. Therefore **first implementations of policies** etc. should focus on the reconfiguration of existing layouts.
- **System autonomy** can allow the shop-floor do be productive during the night, when no operator is present (so-called ghost-shifts). But then the safety and dependability issues are more important.

	Application target	Application time
Norms	all	all
Rules	specific	all
Policies	specific	specific

Table 2. Norms, rules and policies

- The first level of safety is calculating the trajectories of the robots and making sure that they never intersect at the same instant. The second level are collision sensors which are being checked at operation time.
- **Optimisation** itself is of limited interest to EAS; solutions need to be quickly deployable, not optimal, as EAS are not made for Mass Production. On policy level, however, the system sometimes needs to optimise values (for instance $f(x) \rightarrow max$).
- How should **user preferences** and **user-given constraints** be expressed in policies? Preferences are rules which should be followed but can also be ignored if they lead to problems.
- The system needs policies for general behaviour and for adaptation. How to make the difference? One possibility is to make them dependent on the configuration state of the system.
- Until now, evolutionary algorithms have not been proposed within the context of EAS. They might however be an interesting option in order to make systems evolve towards better solutions.

7.1 Implementation issues

The work done at Uninova, Portugal, is based on CoBASA [1], which has been continuously growing during the last years. The currently implemented system encompasses: a directory registration service; an ontology including an OntA; dynamic coalitions of modules to provide composite skills; and easy reconfiguration in case of failures or little changes in process or product design. It does not include the self-organised layout creation from scratch yet, which is currently under construction. The following subsections describe implementations issues; for more see [6, 7].

7.1.1 Services Directory and Ontology

Modules and product parts are all wrapped as agents and register their services at a Directory Facilitator (DF), also storing relevant global knowledge such as the nature of the exchanged messages, skills and skill composition rules. The DF is an active entity incorporating an EAS Ontology, which already exists and is being refined. It responds to requests by returning the list of appropriate modules (skills, speed, etc.). The DF also filters the matching MRAs more precisely by using specifications, e.g. all pneumatic Z-axes which have a workspace of 10cm, can support a charge of 500g and have a gripper interface called *Gxy3*. It delivers also information about which partners a certain module requires, e.g. a Scara robot will need a conveyor passing through the lower part of its workspace and a gripper.

7.1.2 Dynamic coalitions

In the initial CoBASA, coalitions were formed by the user, and they were thus static. The current new approach, enabling the system to execute self-organisation, is based on dynamic coalitions which the agents form themselves and which can change at any time without the user having to reconfigure the coalitions. Knowledge about useful or problematic coalitions is kept (instead of lost at coalition resolution) and can be re-used whenever the same composite skills are required again.Coalitions are formed as an answer to unfulfilled requests from PAs.

7.1.3 Layout formation and LSAI

The self-organised layout formation will be based on the notion of dynamic coalitions. It will be realized using a series of service requests (incorporating the required moves - x/y/z values of the micro-instructions) in order to fulfill the given GAP. If the GAP cannot be fully realized, the current layout is abandoned and a new one is re-built. The layout and the LSAI are then progressively built together. These trial and error constructions of layouts are performed virtually. Modules only ask to be placed on a layout when a complete solution is reached.

8 Conclusions and outlook

The work presented in this report still needs further development and implementation to be validated, but already at this early stage, it shows a high potential for tackling the challenges of modern manufacturing. This report shows the relevance of an architecture for self-organisation and self-adaptation in a real-world industrial scenario, but does not (yet) provide a ready-to-use recipe for building self-organising assembly systems.

The current version of the designed system is self-organising and supports evolvability, but does not consider a way of measuring the quality of the obtained solution: Is it really the solution with maximum precision and minimum resources that has been selected? How long does it take to find a solution, if any? This could be further achieved by inserting more utility functions into the policies.

It also remains to be investigated what to do in case of conflicting policies - consulting the human user is always an option.

Future work encompasses: development of the run-time infrastructure (Figure 8) supporting the dynamic use of policies and metadata; creation of the layout according to the tiles self-assembly model; as well as formal study of the design and proof of dependability properties.

9 Acknowledgement

Regina Frei thanks the Portuguese Foundation for Science and Technology for the financial support (PhD grant SFRH/BD/38608/2007).

References

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

[2] E. Bonabeau, M. Dorigo, and G. Théraulaz. *Swarm intelligence.* Oxford University Press, Santa Fé Institute, NM, USA, 1999.

[3] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Minneapolis, MN, USA, 2007.

[4] J. Buchli and C.C. Santini. Complexity engineering, harnessing emergent phenomena as opportunities for engineering. Technical report, Santa Fé Institute Complex Systems Summer School, NM, USA, 2005.

[5] G. Di Marzo Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi. Metaself - a framework for designing and controlling self-adaptive and self-organising systems. Technical report, School of Computer Science and Information Systems, Birkbeck College, London, UK, 2008.

[6] 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.

[7] 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.

[8] R. Frei, L. Ribeiro, J. Barata, and D. Semere. Evolvable assembly systems: Towards user friendly manufacturing. In *IEEE Int. Symposium on Assembly and Manufacturing (ISAM)*, pages 288 – 293, Ann Harbor, MI, USA, 2007.

[9] J.O. Kephart and R. Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[10] E. Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.