J Ambient Intell Human Comput
DOI 10.1007/s12652-012-0159-2

ORIGINAL RESEARCH

Self-organising assembly systems formally specified in Maude

**Regina Frei · Traian Florin Şerbănuţă ·
Giovanna Di Marzo Serugendo**

Received: 21 May 2012 / Accepted: 13 July 2012

Abstract This article recapitulates on the research done in self-organising assembly systems (SOAS) and presents the completed formal specifications and their simulation in Maude. SOAS are assembly systems that (1) participate in their own design by spontaneously organising themselves in the shop floor layout in response to the arrival of a product order and (2) manage themselves during production. The self-organising process for SOAS to design themselves follows the Chemical Abstract Machine (CHAM) paradigm: industrial robots self-select and self-arrange according to specific chemical rules in response to a product order with generic assembly instructions (GAP). This article presents an additional set of rules describing how the GAP is transformed into layout-specific assembly instructions, which is a kind of recipe for how the self-organising robots assemble the product.

Keywords Self-organisation · Maude · Agile manufacturing · Formal specifications · Assembly systems

R. Frei (✉)
EPSRC Centre for Innovative Manufacturing in Through-life Engineering Services, Cranfield University, Bedfordshire MK43 0AL, UK
e-mail: work@reginafrei.ch

T. F. Şerbănuţă
Formal Systems Laboratory Department of Computer Science, University of Illinois at Urbana-Champaign, 2111A Siebel Center, 201 N., Goodwin Urbana, IL 61801, USA
e-mail: tserban2@illinois.edu

G. D. Marzo Serugendo
Institute of Services Science Faculty of Social and Economics Science, University of Geneva, 1227 Carouge, Switzerland
e-mail: giovanna.dimarzo@unige.ch

1 Introduction

Self-organising assembly systems (SOAS) represent an approach which gives agile manufacturing systems more intrinsic intelligence to serve the user in a more autonomous way. This is a very valuable approach in a time where technological systems are becoming increasingly complex, difficult to manage and laborious to change. In response, a development towards systems with more autonomy can be observed in many different areas of engineering and technology. This includes autonomic computing (Kephart and Chess 2003), complexity engineering (Frei and Di Marzo Serugendo 2011a; Frei and Di Marzo Serugendo 2011b; Frei and Di Marzo Serugendo 2012), Emergent Engineering (Ulieru and Doursat 2011), many types of self-healing technologies (Frei et al. 2012), self-assembly at various scales (Boncheva et al. 2003; Gross and Dorigo 2008; Phili and Stoddart 1996) and many other self-* approaches.

Additionally, manufacturing needs to become more agile and responsive to change in a highly competitive world. Production lot sizes tend towards very small numbers, whereas the product variants and options are getting more diverse. Manufacturing and assembly systems therefore need to be agile, highly responsive to changing requirements, and able to function with minimal downtime for reconfiguration and maintenance. These objectives are addressed in the paradigm of evolvable assembly systems (EAS) (Barata 2005; Onori 2002; Onori et al. 2008, 2011). SOAS (Frei 2010; Frei and Di Marzo Serugendo 2011c) go a step further by assigning the system modules a proactive role in system design and assembly execution.

This article presents the now completed work done based on what was previously published: This includes the design (Frei et al. 2008a, 2010b; Di Marzo Serugendo and

Frei 2010) and the architecture (Frei et al. 2009) of SOAS (Frei 2010; Frei and Di Marzo Serugendo 2011c), as well as the development of a specific ontology and 'on-the-fly' creation of coalitions (Frei et al. 2008b). A brief summary is in (Di Marzo Serugendo and Frei 2012). In Frei et al. (2010a) we explained the various relevant concepts, which are briefly reviewed in this article, clarified the relation between *Ambient Intelligence* and SOAS, and presented the first part of the formal specifications. This work has now been completed and serves as a proof of concept.

SOAS and ambient intelligence: ambient intelligence (ISTAG 2001, 2003) refers to electronic systems that are sensitive and responsive to the presence of people. They are usually embedded in the everyday environment, context-aware, personalised, adaptive or anticipatory to changes in the environment.

In the area of manufacturing and assembly, a *product designer* traditionally provides a design of the product, this is an assembly sequence technically describing how to join the products parts and in which order. A team of engineers then builds an appropriate assembly system, which consists of a series of modules or complete robots, properly arranged and connected in the shop-floor layout, and programmed to execute the assembly movements.

Traditionally, the resulting assembly system is dedicated to building the specified product and runs under central control. A *human operator* monitors the system, its performance and the quality of products. The system is equipped with security features that stop the system in case of critical failures. The operator must then find out what went wrong and how to fix it.

Evolvable and self-organising assembly systems revisit this way of building systems and demonstrate Ambient Intelligence features and responsiveness to people along the following three lines (Frei et al. 2010a):

- *Product requirements and link with the product designer.* A new assembly sequence, provided by the product designer, triggers a self-organising process among the different modules available either from the storage or already positioned on the shop-floor. The modules spontaneously organise into appropriate coalitions (groups of modules) to fulfill the tasks specified in the assembly sequence.
  The assembly processes and the assembly system technology can even iteratively influence the product design (Onori 2002). The product designer and the assembly system collaborate in producing the final product design.
- *Layout design and link with the engineer building the assembly system.* The modules autonomously search for suitable coalition partners to compose the skills required to fulfill the tasks of the assembly sequence. Additionally, the coalitions arrange themselves in the shop-floor layout: they choose a position taking into account overlapping workspaces. The engineer may also collaborate to this process and suggest some specific module or preferred position. The obtained layout and final choice of modules is validated by the engineer. The assembly system resulting from the collaboration between the engineer and the self-organising modules is then able to execute the assembly sequence.
- *Production and link with the operator.* During production, the assembly system performs monitoring tasks. The individual modules participating in the assembly system monitor themselves, their neighbours, the quality of the produced products, any unexpected item (human hand, dropped part). Some of these tasks involve high precision, mini and/or micro-movements, checking of performances that go beyond human capabilities. In collaboration with the operator, who can stop/reset the system at any time, the assembly system runs the production at the best possible performance given the current production conditions.

The technology involved encompasses: RFID tags attached to individual products being assembled reporting on assembly tasks performed so far, sensors attached to the different modules reporting on performances such as precision or speed, and autonomous software agents acting as wrappers around the physical modules enabling them to become reactive and adaptive.

Organisation of this article: Sect. 2 briefly introduces rewriting logic, the chemical abstract machine and the maude software. Section 3 details the case study used in this paper. Section 4 recalls the previously published specifications and indicates differences between the previous and the new work. Section 5 presents the completed specification and simulation results. Finally, Sect. 6 concludes this article.

2 Background and previous work

This section is intentionally kept short. For more details, please consult Frei et al. (2010a).

The **Gamma chemical reaction model** (Banâtre et al. 2000) was introduced as an alternative to sequential models of programs. Gamma is built around the idea of a chemical reaction metaphor. The main data structure is the *multiset* seen as a chemical solution. A program is then a pair $(ReactionCondition, Action)$. The execution consists of removing from the set the elements that appear in the *ReactionCondition* part and replacing them with the product of the *Action*. The program stops and reaches a stable state when no more reactions can take place.

The **Chemical Abstract Machine (CHAM)** (Berry and Boudol 1998) is an extension of the Gamma model allowing modular structures: chemical reactions may occur within membranes and stay local to the membrane; and the opposite operation, the airlock, allows the extraction of a molecule from a membrane.

Rewriting logic was introduced by Meseguer (1990, 1992) as a fundamental logic for concurrency, modelling transitions between equationally defined congruence classes of terms. Rewriting logic extends equational logic with rewrite rules, allowing one to derive both equations and rewrites (or transitions). Deduction remains the same for equations (i.e., replacing equals by equals), but the symmetry rule is dropped for rewrite rules. Formally, a rewrite theory is a triple (Σ, E, R), where (Σ, E) is an equational specification and R is a set of rewrite rules. Rewriting logic is a framework for true concurrency: the locality of rules, given by their context-insensitiveness, allows multiple rules to apply at the same time provided their patterns do not overlap.

Maude (Clavel et al. 2007) is a rewrite engine offering full execution and analysis support for rewriting logic specifications.[1] Maude provides an execution and debugging platform, a breadth-first search (BFS) state-space exploration, and a linear temporal logic (LTL) model checker (Eker et al. 2003), as well as an inductive theorem prover (Clavel et al. 2006) for rewrite logic theories; these translate immediately into corresponding BFS reachability analysis, LTL model checking tools, and theorem provers for the defined models. For example, these generic tools were used to derive a competitive model checker (Farzan et al. 2004), and a Hoare logic verification tool (Sasse and Meseguer 2007) for the Java programming language.

CHAM within rewriting logic: The chemical abstract machine can be viewed as a particular definitional style within rewriting logic (Meseguer 1992; Şerbănuţă et al. 2009). That is, every CHAM is a specific rewrite theory in rewriting logic, and CHAM computation is precisely concurrent rewriting computation.

3 Case study example of SOAS

For illustration purposes, we assume the following simple product to be assembled: an adhesive tape roller dispenser, consisting of two body case parts ($part_1$ and $part_3$), a tape roll ($part_2$) and a screw ($part_4$), assembled on top of a carrier, as shown in Fig. 1. For more details see Frei (2010).

The **Generic assembly plan (GAP)** specifies the way a product is to be assembled: it includes the assembly

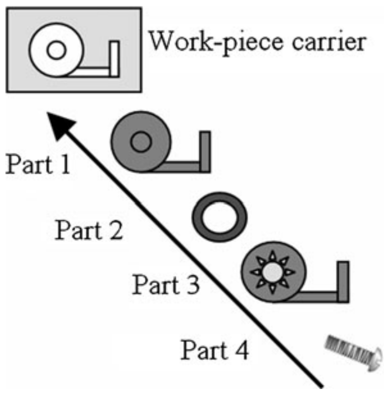


Fig. 1 The adhesive tape roller dispenser

sequence of the different parts and the way they must be joined. Tasks are defined in the form of generic operations (equivalent to skills). The GAP does not provide information about what module[2] to use and what movement to make. In other words, the GAP says *what* to do (assemble 10 tape rollers by joining $part_1$ with $part_2$, $part_3$ and $part_4$) but not *how* (which robotic modules handle which parts and execute which movements at which instant) and is thus independent from any layout. Figure 2 shows the example of a GAP represented as a workflow and written in XML.

The six tasks illustrated in Fig. 2 each have an operation type (Op), an object to be handled (Obj), a start point (StPt), an end point (EndPt), as well as a start orientation (StOr) and an end orientation (EndOr), referring to the parts to be treated. We assume this to be sufficient information at this stage of implementation. This GAP specifies that a carrier is loaded from the storage to *conveyor*, then $part_1$ is picked from $feeder_1$ and placed on the carrier, then $part_2$ is picked from $feeder_2$ and placed on top of $part_1$. The same procedure follows for $part_3$ that is placed on top of $part_2$, and $part_4$ that is screwed into $part_3$. Finally, the *carrier* with the assembled product is unloaded to the storage. The flash in the rectangle on the left hand side of the GAP represents the beginning (*IN*), and the square on the right hand side stands for the end (*OUT*).

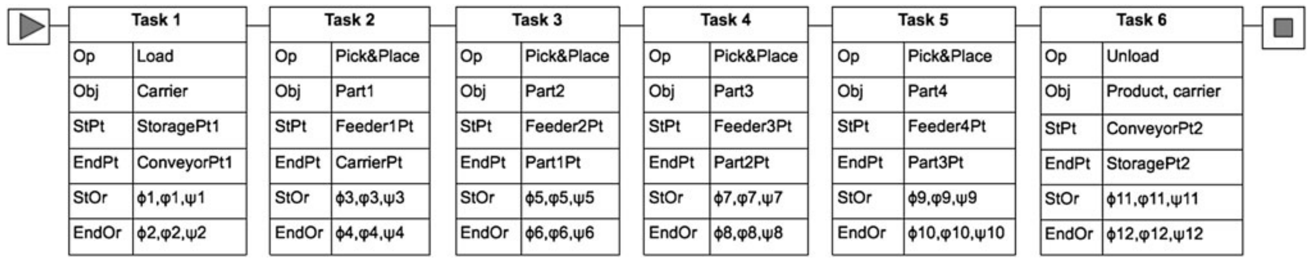
The **layout** is incrementally built according to a self-organising process illustrated in Fig. 3: modules self-assemble to form coalitions according to a process of reactions and rewriting. Coalitions are built to progressively match with the tasks defined in the GAP.

The details of the modelling of this self-organising process leading to such a layout can be found in Frei et al. (2010a); a summary is here above in Sect. 2.

To assemble the product, the GAP needs to be transformed into layout-specific assembly instructions (LSAI) as shown in Fig. 4. This transformation takes into account

[1] This work was implemented in Maude version 2.4.

[2] with the exception of the feeders, which are part-specific



	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
Op	Load	Pick&Place	Pick&Place	Pick&Place	Pick&Place	Unload
Obj	Carrier	Part1	Part2	Part3	Part4	Product, carrier
StPt	StoragePt1	Feeder1Pt	Feeder2Pt	Feeder3Pt	Feeder4Pt	ConveyorPt2
EndPt	ConveyorPt1	CarrierPt	Part1Pt	Part2Pt	Part3Pt	StoragePt2
StOr	ϕ1,φ1,ψ1	ϕ3,φ3,ψ3	ϕ5,φ5,ψ5	ϕ7,φ7,ψ7	ϕ9,φ9,ψ9	ϕ11,φ11,ψ11
EndOr	ϕ2,φ2,ψ2	ϕ4,φ4,ψ4	ϕ6,φ6,ψ6	ϕ8,φ8,ψ8	ϕ10,φ10,ψ10	ϕ12,φ12,ψ12

Fig. 2 Example of a GAP written as a workflow

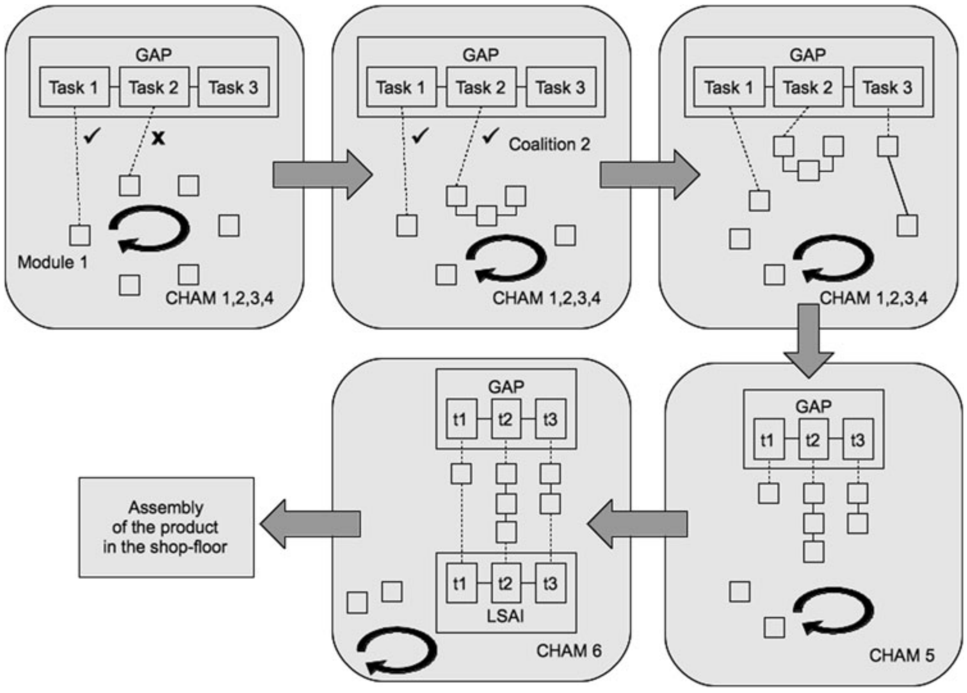


Fig. 3 Self-assembly of coalitions in CHAM, illustrating the process phases dominated by rules 1–4 as well as 5 and 6.

the actual modules, the tasks and the parts. The LSAI consists of executable programs for each of the robotic modules in the coalitions, based on their requested skills. The instructions are generated for a certain layout; if the layout is modified, these instructions must be changed.

At production time, the assembly of a product will result from the execution of the LSAI by the agents/modules according to the workflow. Any change requiring a layout reconfiguration restarts the self-organising layout design process.

Figure 6 illustrates the assembly procedure on a hypothetical linear layout.

4 Previously published specifications

The specifications recalled in this section were published in Frei et al. (2010a) and are necessary for understanding the new work published in Sect. 5. The first four of the following six rules (Frei and Di Marzo Serugendo 2011c; Frei 2010) have been specified previously; the last two are specified in this article.

1. Interface compatibility
2. Composition patterns
3. Creation of composite skills
4. Task coalition matching
5. **Layout design and transport linking**
6. **Transforming the GAP into the LSAI**

Note that Maude works with so-called *modules*, written as mod(...)endm. This is an unfortunate coincidence with the typical use of the word module in this article, which refers to a manufacturing resource agent (MRA).

Similarly, note that the term *agent* used in this context is not to be confused with software agents mentioned elsewhere.

The entire specification will not be discussed in detail here; it can be downloaded from: http://code.google.com/p/soas-maude.

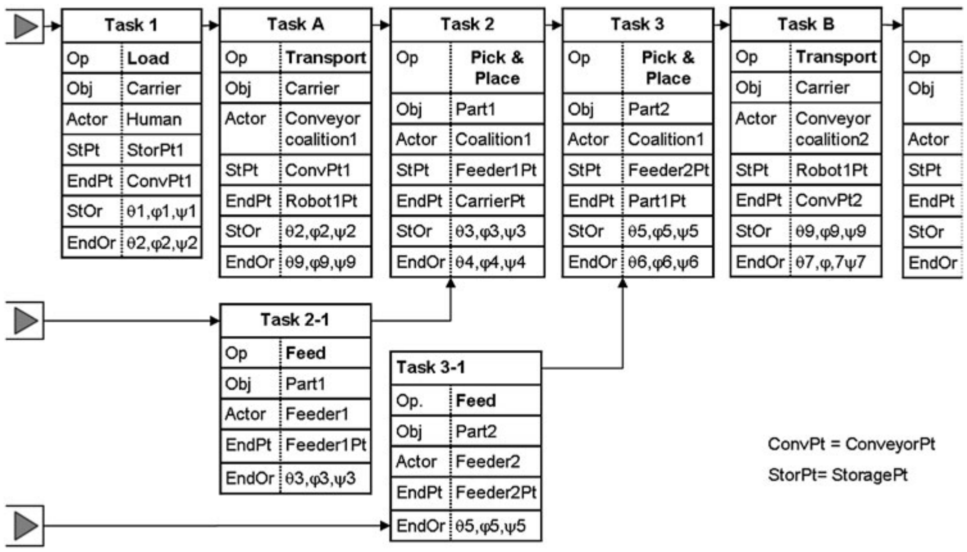


Fig. 4 Example of an LSAI written as a workflow (incomplete view)

4.1 Structure of the model

The language of our model consists of nested "cells", which are named CHAM-like molecules. Cells contain either concrete values, like Integers, Floats, Names, and enumerations, or a "soup" of other cells, representing CHAM-like configuration molecules.

The *top configuration* cell, shown in Fig. 5, contains three mandatory cells:

- *gaps*, specifying the general assembly plans
- *mras*, specifying the manufacturing resource agents
- and *parts*, specifying the parts
- as well as an optional cell, which appears during the model simulation, which contains the *agents*, (partial) coalitions of modules aiming to solve a certain task.

For details about other items, please refer to Frei et al. (2010a).

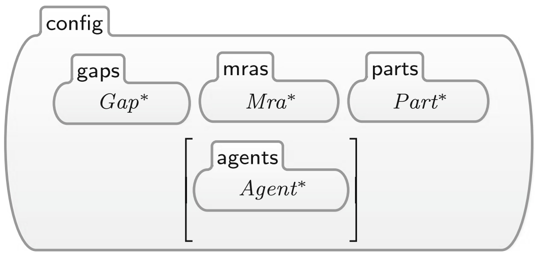


Fig. 5 The top configuration cell

4.2 Differences from the previously published specifications

The original specifications have been refined and improved; they are now richer, more detailed, and more realistic.

In the previous version (Frei et al. 2010a), the case study only included two parts, namely a body case and the adhesive tape roller. In this extended version, it consists of four parts, with an additional body case part and a screw which holds them together. Moreover, in the previous version, only four of the six types of rules to determine the process of self-organisation were formally specified, whereas now, all six are included. The current version is complete and allows the assembly system to fully design itself from the arrival of a product order (under the form of a GAP) to a complete assembly system ready to work on the pieces and assemble products.

The specifications now include simple rules for layout formation, which guide the robots towards arranging themselves in a serpentine form, with the feeder always being on the left of the robot. This was an initial choice we took for experimenting the concept, and only serves as an example at this stage. Any number of other configuration rules could be specified, such as, for instance, "if the base axis is pneumatically actuated, the feeder places itself on the right of the robot base, with a distance of 10 cm".

As a perspective for practical use of these specifications, a template for different layout creation rules/LSAI derivation rules could be provided, so that the user could give their wishes as input. The system would then be guided to design itself according to the requirements of specific production facilities and orders. There could be many different patterns for combining feeders, robots and robotic modules, grippers and conveyors, and they could arrange themselves in any specified shop-floor layout.

The previously published version of our work specified the following:

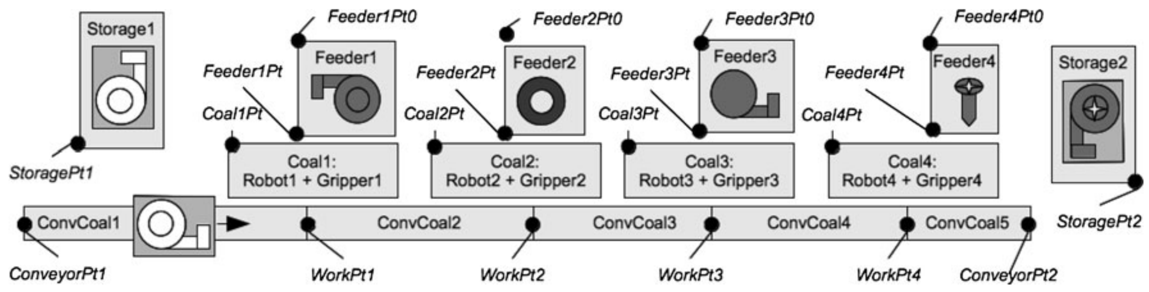


Fig. 6 Concept for a hypothetical linear assembly line

- MRA (manufacturing resource agent)
- Skills
- Positions, angles
- Task
- GAP (generic assembly plan)
- LSAI (layout-specific assembly instructions)
- Interfaces (+/-) and diverse types
- Parts to be assembled
- Reaction rules:

1. Interface compatibility: describing the physical compatibility of the modules.
2. Typical composition patterns: certain modules are typically combined with certain other modules, e.g. a gripper is always held by a robotic axis (either an individual robotic module or a complete robot).
3. Creation of composite skills: the modules which form a coalition contribute their individual skills and form composite skills, e.g. a gripper can open and close, an axis can move, and together they form a composite pick&place skill.
4. Task–coalition matching (considering only the type of skills required): a task which requires a certain skill will associate with a coalition that offers those skills.

New features in the additional rules detailed in this article:

Additional reaction rule: 5) Layout creation and transport linking: the chosen modules and coalitions are arranged to form a shop floor layout and connected through conveyors (or potentially other means of transportation).

- Form the layout (robots and feeders choose positions in the layout).
- Establish their geographical arrangement; link robots with conveyors.
- If impossible, adjust robot positions.
- IN and OUT to mark the inputs and outputs of a layout.
- Limit the search space by taking into account user preferences or other constraints.
- Adapt / reconfigure the layout in case of problems; how to determine a solution with minimal changing effort?

Additional reaction rule: 6) Derivation of the LSAI from the GAP: the concrete assembly movements need to be made explicit for the chosen layout.

Interface verification: Verifying that the interfaces which are about to connect have opposite polarities (either '+/-' or '-/+') $\rightarrow$ extending rule 1).

Gripper type: Matching the gripper-type, subtype and range with the material and size of the part to be moved $\rightarrow$ extending rule 4).

Indirect requirements: Taking into account the requirements of the modules (for instance, a middle axis will need a base axis) $\rightarrow$ extending rule 2) and 3).

Coalition selection: Rules for deciding which coalition will be associated with a task, if there are more than one possibilities (till now, the first solution was taken) $\rightarrow$ extending rule 4).

The CHAM-based part of the process ends once suitable modules have associated with tasks to be fulfilled. The layout creation and then the LSAI derivation are based on simple rules, which are being defined using functional Maude equations.

5 Completed specification and simulation results

5.1 Available modules

The following modules were specified and are thus available in the hypothetical shopfloor repository to build a layout:

- 3 industrial robots *abb-robot*, *cartesian-robot*, and *scara-robot*;
- Base axes a_1, a_2, a_3 (having only one interface, the other connection being the ground), middle axes a_4, a_5, a_6, and top-axes a_7, a_8, a_9 (able to hold a gripper);
- Conveyors (generic);
- Endings of the layout *ending-in* and *ending-out*;

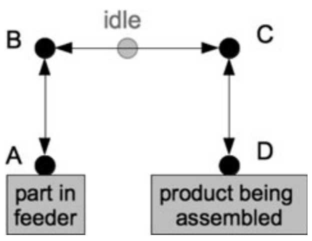


Fig. 7 Positions *A*, *B*, *C*, *D*, and *idle*

– Feeders f_1, f_2, f_3 and f_4;
– Grippers g_1, g_2, g_3, g_4, and g_5;
– Positioning devices pd_1, pd_2, pd_3 and pd_4 (allowing the conveyors to place the carrier correctly under the robots);
– Humans (generic); technically, the human is wrapped by an MRA with the skills *load* and *unload*.

These modules were chosen because they are suitable to execute the tasks specified in the GAP. Like the product chosen for this case study, the modules only represent an example; any other set of modules could be specified according to the user's needs and the existing equipment. Not all available modules are necessarily used in the layout. For efficiency reasons, humans and conveyors were abstracted; there are as many of them available as required.

5.2 Pick&place operation

Figure 8 illustrates the possible trajectories of a robot executing a pick&place operation. The movement sequence is normally as follows, with the positions illustrated in Fig. 7:

– Preparatory phase : from idle position or any other position, move to B. If gripper closed: open gripper.
– Phase 1: From B move to A, close gripper, move to back to B.
– Phase 2: From B move to C, or the other way round.
– Phase 3: From C move to B, open gripper, move to back to C. If no other task to execute: move to idle position and close gripper.

Furthermore, Fig. 8 indicates the robotic skills which could execute the required movements. For Phase 2, for instance a robot capable of moving linearly in the horizontal plane (lin x, lin y) would be suitable. Alternatively, also a robotic module with the vertical z axis as a rotational axis may be used. For reasons of simplicity, different solutions with other modules or other module combinations are not considered here. For Phases 1 and 3, an axis that moves linearly along the z axis or one that rotates along a horizontal axis (no matter if x or y) may be used. Again,

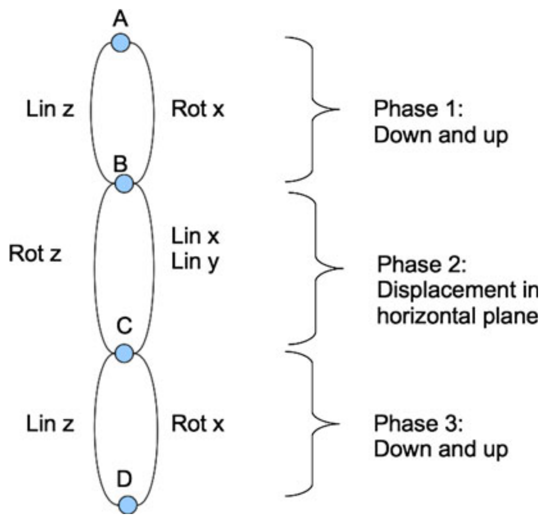


Fig. 8 Robot trajectories in a 'pick&place' operation and suitable movements; abstract representation showing the combinatoric possibilities to combine robot movements

this choice is not exhaustive and motivated by the intention to simplify the model.

In this sense, the generation of composite skill 'pick&place' is currently hardcoded in the rules. This is suitable because the combination of gripper and axes follows a typical composition pattern. For less common module combinations - for instance, combining a full industrial robot with an additional axis, other rules could be introduced; there is even some space for the emergence of new opportunistic combinations.

The Maude rules defining the LSAI generation for the pick&place operation are presented in Sect. 5.4.

5.3 Rules for layout design and transport linking

Figure 9 illustrates the result of the specification execution, based on a set of simple rules, as follows:

– Start at (0,0,0) + offset, going eastwards (rule in Fig. 10)
– Add a coalition towards the direction of advancement if there is enough space (e.g., rule in Fig. 12)
– If end of floor is reached, add a conveyor corner, then a conveyor going north, another conveyor corner, and change the direction (e.g., rule in Fig. 13)
– Stop when all coalitions have been placed

Any other strategy or rules for building a layout could be specified according to the user's preferences and the actual shop floor constraints.

In our current Maude implementation, the layout derivation begins at the end of the process of assigning tasks to the formed MRA coalitions. Once a set of agents covering

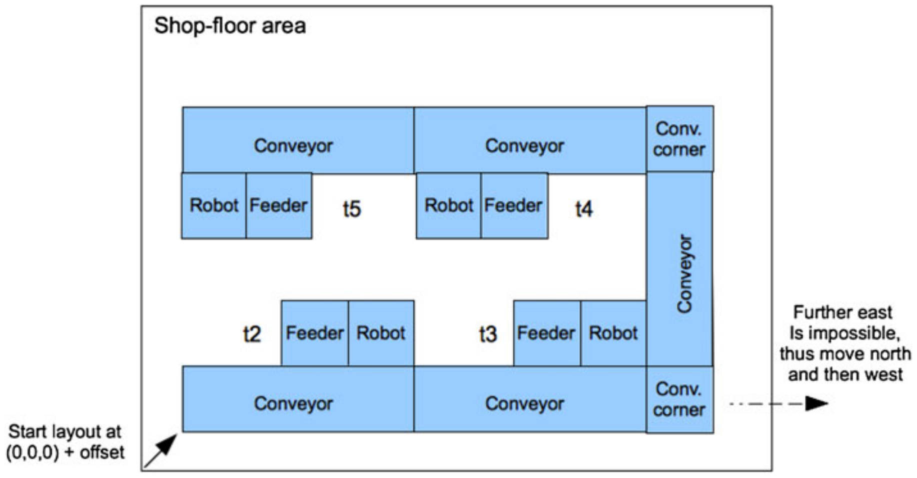


Fig. 9 Layout generated with a set of simple rules inspired by chemical abstract machine, representing an example of a more general case. The first task t1 (loading) and the last task t6 (unloading) are executed by humans and therefore not shown. To maintain the generic nature of the illustration, the robots and grippers are not given specific identifications. The form of the layout was determined by layout formation rules that were chosen for no particular reason; they could be easily replaced by any other layout formation rules and are classified as 'user preferences.

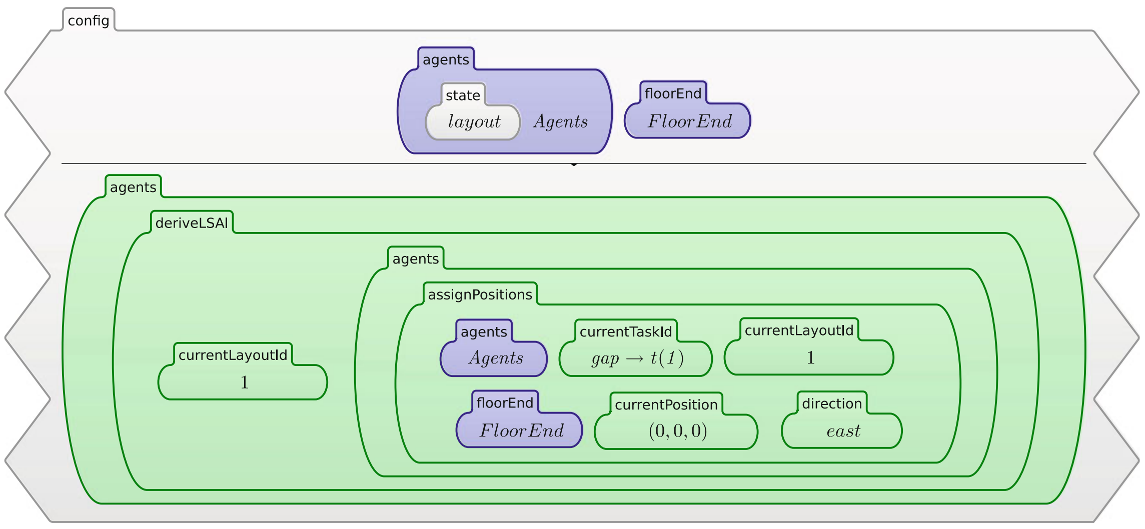


Fig. 10 Rule for starting the layout and LSAI plan generation

all the tasks was successfully chosen, the system transitions into the *layout* state. Once this happens, the rule in Fig. 10 is triggered, setting the environment for the layout and LSAI generation phases.

This rule sets up a nested structure of cells which will trigger the layout and LSAI specific rules. For example, the layout generation cell is initialized with the agents resulted after the coalition formation phase, as well as with cells containing necessary metadata information such as the beginning position set to the south-west corner or the direction of advancing set to `east`. These cells are conceived to dissolve once all the agents have been processed

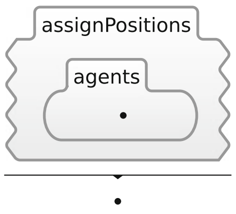


Fig. 11 Rule for concluding the layout generation phase.

in that stage; for example, Fig. 11 presents the rule concluding the layout generation phase.

A typical rule for setting the position of an agent is presented in Fig. 12. The rule first checks in the side condition whether there is enough room on the shop-floor

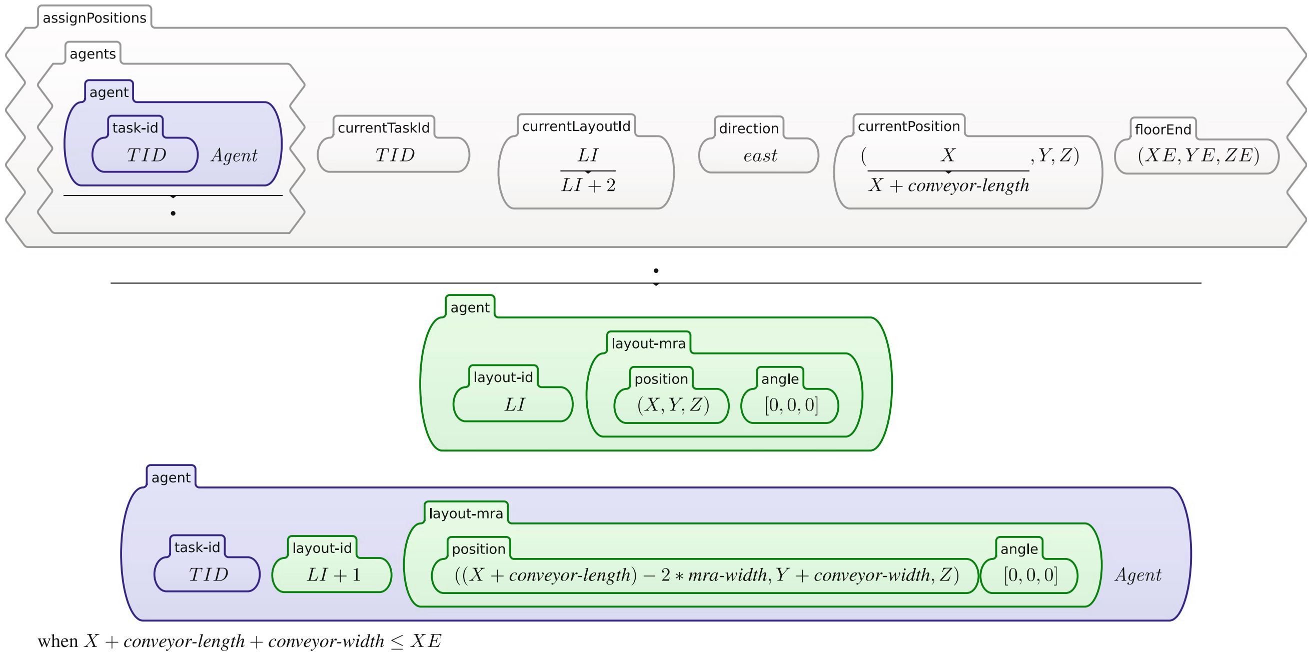
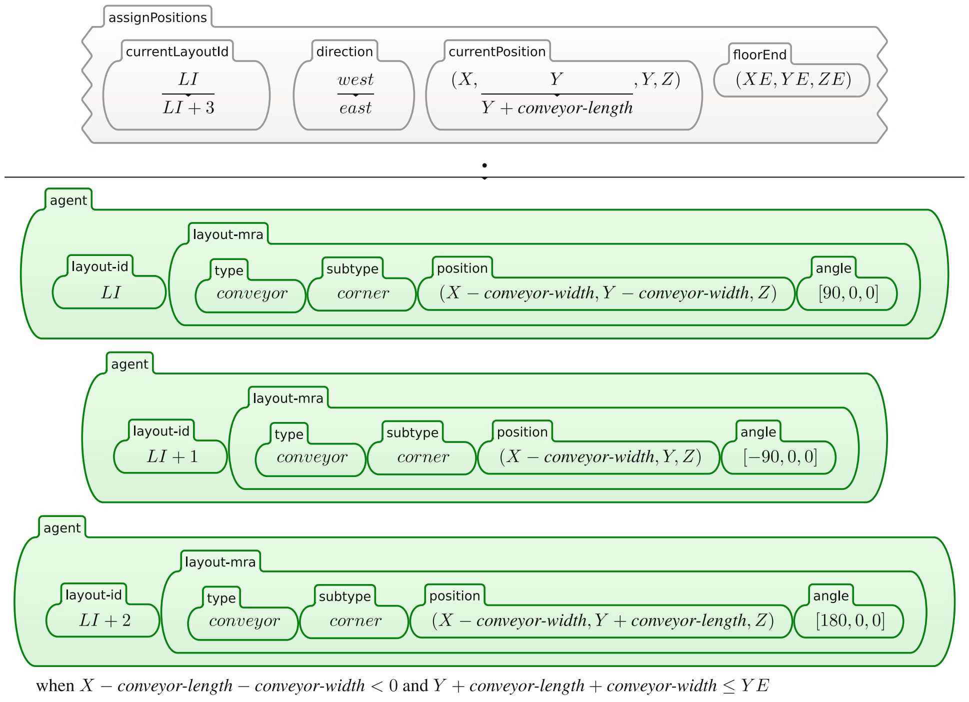


Fig. 12 Setting an agent in the layout

Fig. 13 Switching the direction of the assembly line



in the direction of advancing (here `east`). If so, it creates a new agent to represent the conveyor taking the product from its previous station to the current one and then it sets the position of the current agent at the end of the newly added conveyor. At the same time, the current position is updated to the right of the positioned agent. Note that in this rule each agent and carrier gets assigned an id identifying its position in the assembly line. The rules for the west direction are similar.

When the end of the shop floor is reached (either towards `west` or `east`), a corner rule applies. Figure 13 presents the rule for changing direction of the assembly line when the west end of the shop floor is reached. The rule first checks in the side condition if adding a new agent in the same direction will exceed the limits of the shop floor (here westwards), and if this is the case, it verifies that the shop floor has enough space to continue northwards. If so, three new conveyors are positioned: a corner one, to take the product from the previous conveyor and orient it northwards, then a regular conveyor to carry the product north, followed by another corner conveyor to change the direction to the opposite of the original one (here from `west` to `east`).

5.4 Rules for tranforming the GAP into the LSAI

Based on the GAP and the created layout, a corresponding LSAI needs to be generated. This is done according to a set of rules which can be modified depending on user preferences and robot characteristics.

Taking into consideration the coordinates of each module's position in the layout, the previously abstract movements are now instantiated. Abstract points become concrete, and skills become operations.

For example, the main rule for deriving the pick&place movement sequence described in Sect. 5.2 is formalized in our Maude implementation by the following equation:

```
eq createP&PMovements(⟨agent⟩ AgentIs ⟨/agent⟩, AP, DP, Skills)
  = ⟨agent⟩ AgentIs
      ⟨procedure⟩
        moveDown(AP,Skills),
        ⟨ skill ⟩
          ⟨type⟩ open−close ⟨/type⟩ ⟨subtype⟩ close ⟨/subtype⟩
        ⟨/ skill ⟩,
        moveUp(AP, Skills),
        moveFromTo(AP, DP, Skills, −1.0),
        moveDown(DP, Skills),
        ⟨ skill ⟩
          ⟨type⟩ open−close ⟨/type⟩ ⟨subtype⟩ open ⟨/subtype⟩
        ⟨/ skill ⟩,
        moveUp(DP, Skills),
        moveFromTo(DP, AP, Skills, 1.0)
      ⟨/procedure⟩
    ⟨/agent⟩ .
```

This equation gives a high-level formalisation of the pick and place procedure, by translating it in concrete movements and open/closing of grippers executed by the MRAs coalition with its available skills. The idle position is δ_z above the middle of the feeder with the gripper open. From this position the coalition uses the available `Skills` to `moveDown` to the middle of the feeder (position `AP`), then it uses the open-close skill of the gripper to close and grip the part being fed, then it first moves back up to the idle position (`moveUp`), then in the horizontal plane (`moveFromTo`) above the position of the conveyor (position `DP`), where it descends (`moveDown`) and it opens the gripper to place the part on top of the current product. Finally, it retraces its moves up and then back to the idle position. Figure 7 illustrates this sequence of movements.

As mentioned above, the precise movements are computed based on the available skills, and could be a composition of linear and rotational movements, with preference given to linear movements. For example, the equational specification for the `moveDown` operation consists of the two conditional equations depicted below, the first used when linear movement skills are available, while the second when only rotational skills can be used.

```
ceq moveDown((X,Y,Z), Skills ⟨skill⟩ Skill
                                    ⟨type⟩ move ⟨/type⟩
                                    ⟨subtype⟩ linear ⟨/subtype⟩
                                    ⟨direction⟩ vertical ⟨/direction⟩
                                    ⟨range⟩ Range ⟨/range⟩
                                  ⟨/ skill ⟩)
  = ⟨ skill ⟩ Skill
      ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ − float(deltaz − Z) ⟨/range⟩
    ⟨/ skill ⟩
  if rat(Range) ⟩ − deltaz + Z .

ceq moveDown((X,Y,Z), Skills ⟨ skill ⟩ Skill
                                    ⟨type⟩ move ⟨/type⟩
                                    ⟨subtype⟩ rotational ⟨/subtype⟩
                                    ⟨direction⟩ vertical ⟨/direction⟩
                                    ⟨range⟩ Range ⟨/range⟩
                                  ⟨/ skill ⟩)
  = ⟨ skill ⟩ Skill
      ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ MRange ⟨/range⟩
    ⟨/ skill ⟩
  if MRange := − 2.0 ∗ asin(float(deltaz − Z) /
                    float (2 ∗ mra−length)) ∗ 180.0 / pi
  /\ Range ≥ abs(MRange) .
```

The `moveDown` takes as argument a three-dimensional position but only moves on the vertical axe, from the current position (δ_z) to the one specified (`Z`). The equations first detect a skill which can be used for the movement (signified by the side condition of the equation) and then they create a new variant of the skill constituting the actual operation, in which the range made available by the skill is replaced by the concrete range and direction required for the specific move operation. The equations defining the `moveUp` and `moveFromTo` operation follow a similar pattern.

The Maude implementation also includes definitions for the concrete conveyors movements, as well as for the humans loading/unloading the final product.

Without going into additional gory details here, the interface for deriving the LSAI is specified in Maude as follows:

```
eq deriveLSAI(SAgents, Cfg)
 = deriveLSAI(sortedList(SAgents), nil, Cfg,
                   ⟨product⟩ nil ⟨/product⟩) .

eq deriveLSAI(nil, Agents', Cfg, Product)
 = ⟨final−agents⟩ Agents' ⟨/final−agents⟩
   ⟨ final −product⟩ Product ⟨/final−product⟩ .
```

The first equation defines the `deriveLSAI` function taking as arguments the set of agents obtained after the layout generation has completed and the remainder of the configuration, containing the specifications for parts, modules, and the assembly plan. This function delegates work to an auxiliary function (named `deriveLSAI` as well), which takes as arguments the list of agents sorted by task identifiers and the configuration, and uses the additional two arguments to hold the agents which have already been associated LSAIs to, as well as for dynamically constructing the final products. Once the processing has completed, making use of rules as those for the pick&place operation mentioned above to associate LSAI information to all agents, then the second equation applies, presenting the resulting agents and the final product as a result.

5.5 Example of an execution result

For the case study presented in Sect. 3 and the modules listed in Sect. 5.1, Maude generates 47 possible solutions in just a few seconds. The final product generated by Maude, including the concrete positioning of all parts is presented in Appendix A.

Each solution also includes the coalition formed to solve each task, with their concrete positioning on the shop floor, and the conveyors linking them. The LSAI procedures are also represented in these agents, including the movements of the carrier and loading/unloading the product. The Maude generated output for the first solution describing these agents and their corresponding LSAI procedures is presented in Appendix B. Additionally, Fig. 9 illustrates the procedure to assemble the product:

It associates a person to load the carrier on the conveyor, satisfying task 1. Then a conveyor carries the carrier eastward, where the coalition associated with task 1 picks the `body-case` from the feeder and places it on top of the carrier to complete task 2. The another conveyor carries the product eastward where the coalition associated with task 2

picks the `tape-roll` from the feeder and places it on top of the body case to complete task 3. Again, the product is carried eastward by another conveyor, where the coalition associated with task 3 picks the `body-case` from the feeder and places it on top of the tape roll to complete task 4. Given that the eastward end of the shop floor is about to be reached, the direction of movement is changed first northward by using a corner conveyor, then a regular conveyor is used to move the product north to create enough separation space from the existing coalitions, and then again Sa corner conveyor is used to change the direction westward. A conveyor then transports the product westward, where a coalition associated with task 4 (and thus containing a gripper that is able to insert a screw) picks the screw and uses it to connect the existing parts on top of the carrier to complete task 5. Finally, the product is carried away westward by a conveyor at the end of which a person awaits to unload it.

As mentioned above, Maude generates 48 solutions for this GAP. The solutions differ from each other in that the available modules compose different coalitions, which then also reflects in the generated LSAI. For example, one difference between the first and the 25th solution is that in the 25th solution, the coalition addressing task 2 contains robot `r3` instead of robot `r1`, and thus the skills used for movements have to be rotational instead of linear; similarly, for task 3, robot `r1` is used instead robot `r2`, changing the movement type from rotational to linear; finally, for task 4, `r2` is used instead of `r3`, again changing the movement type to linear from rotational.

6 Discussion and conclusion

The original formal specifications of *Self-organising assembly systems (SOAS)* have been refined and improved; they are now richer, more detailed, and more realistic. Given a *generic assembly plan (GAP)*, a set of manufacturing resource agents (MRA) as well as user preferences, the Maude software produces a self-organising shop-floor layout that is able to execute the required assembly operations, together with the *layout-specific assembly instructions (LSAI)*.

The execution traces serve as a proof that Maude was able to find a solution, meaning that the *manufacturing* *resource agents (MRAs)* can arrange themselves in a suitable way to execute the required assembly operation and thus to assemble the final product.

A valid question which the informed reader might ask is how to make sure that the obtained solutions are correct, and whether all possible executions are obtained. With respect to the first issue, we assume all of our rules, implemented as equations and rewrite rules in Maude, are faithfully modeling the rules and the intuition described in the paper. Moreover, given the constructive nature of the problem being addressed, any solution given by Maude can first be traced (to check that each rule applied correctly) and validated by effectively checking that the solution conforms to the problem. With respect to the technique being exhaustive, we do not see this as being crucial, as plenty of solutions are found nevertheless. The Maude rewrite engine does guarantee that all the behaviors of a rewrite specification are being explored using its search command if the rewrite system associated to it by orienting equations from left to right is confluent and terminating on its equational part, while its rules are coherent with regard to the equations (Clavel et al. 2007). *Confluence* of the equational part comes from the fact that the equational part is deterministic. This can be syntactically checked by noticing that equations are non-overlapping, i.e., no more than one left-hand-side matches the same part of a term to be rewritten concurrently. *Termination* is harder to prove, requiring decreasing orders; however, the specifications describe terminating processes, so intuitively rewriting by equations should converge. Moreover, if the equations were non-terminating, there are high chances that the rewrite process would be non-terminating too, and thus not able to produce any solution, which clearly is not the case. Finally, *coherence* itself is nontrivial to prove in general; however, it can be checked that the patterns used by our rewrite rules are normal forms with regard to the equations, which is a strong syntactic criterion for coherence.

While the current work is only preliminary, it could be transformed into a user-friendly software system that manufacturing companies could use to verify if their manufacturing resources are sufficient to assemble a product, and to generate one or many suggestions for how to arrange the shop-floor layout, as well as to produce a rough sketch of the necessary assembly movements of each robotic module. Refinements would then be made using the individual control software of each robot or existing system integration tools. An advantage from using the approach we suggest is that it considerably shortens and facilitates the assembly system design phase, both when setting up a new system (which may take up to six months) and when modifying an existing one.

In the current version of the specification, the first step is to generate the layout for a given GAP, and then to derive the corresponding LSAI. This is suitable for a proof of concept, that such specifications and rules are capable of generating a layout. The currently implemented model is simplified and only searches for a viable solution, without giving any importance to performance characteristics or efficiency.

If the goal is to produce an optimised solution, the procedure could also be done the other way round: more specific assembly instructions could be specified or derived first, and then a suitable layout generated. The best results would be received iteratively: having a rough concept of a suitable layout, deriving suitable assembly instructions, then improving the layout, and adapting the instructions, and so forth.

The currently used sequence of movements for a ‘pick&place’ operation is very simple; it is described in Sect. 5.2. In a more realistic implementations, more sophisticated sequences and algorithms could be used to produce movements that are optimised for each robot and its kinematic characteristics.

For the sake of simplicity, the system currently only generates a linear layout for a linear GAP; given a dependency graph for a more complicated assembly, potentially with several sub-assemblies to produce first, a suitable layout could be generated as well. Again, user preferences would guide what kind of layout would be designed.

The main accomplishment of this work is that an assembly system is able to design itself, that is, to select suitable modules, create coalitions to provide all composite skills needed to assemble the product, to arrange the coalitions in a shop floor layout, and to produce the robotic modules’ movements (i.e. to derive the LSAI). Future work is directed towards mechanisms for self-management while the system is executing the assembly.

For this work to be useable in an industrial setting, it requires for the specifications to be translated into a multi-agent system that is able to communicate with the robots and the order processing system used by the manufacturing company. Suitable multi-agent manufacturing shopfloor control systems exist, with CoBASA (Barata 2005) being the basis on which this work was developed. CoBASA has been augmented over the last few years, adding more autonomy at the module level and including features for a variety of self-* properties, including self-adaptivity, self-diagnosis and self-learning (Barata et al. 2010; Candido et al. 2012; Frei 2010; Ribeiro et al. 2008).

In terms of the formal specifications, future work includes the preparation of a template for the user to comfortably specify any type of desired layout as well as other preferences, and the specification of the self-management process that governs the actual production execution, as opposed to the creation of the assembly system, which was covered in this article.

A: The product as generated by the Maude specification

```
⟨ final −product⟩
 ⟨product⟩
  ⟨part⟩
    ⟨type⟩  carrier  ⟨/type⟩
    ⟨name⟩ wpca ⟨/name⟩
    ⟨material⟩  metal ⟨/material⟩
    ⟨weight⟩ 350 ⟨/weight⟩
    ⟨x−dimension⟩ 50 ⟨/x−dimension⟩
    ⟨y−dimension⟩ 50 ⟨/y−dimension⟩
    ⟨z−dimension⟩ 30 ⟨/z−dimension⟩
    ⟨part−position⟩ (350,450,0) ⟨/part−position⟩
    ⟨part−angle⟩  [0,0,0]  ⟨/part−angle⟩
  ⟨/part⟩,
  ⟨part⟩
    ⟨type⟩  body−case ⟨/type⟩
    ⟨name⟩ p1 ⟨/name⟩
    ⟨material⟩  plastic  ⟨/material⟩
    ⟨weight⟩ 300 ⟨/weight⟩
    ⟨x−dimension⟩ 50 ⟨/x−dimension⟩
    ⟨y−dimension⟩ 70 ⟨/y−dimension⟩
    ⟨z−dimension⟩ 20 ⟨/z−dimension⟩
    ⟨gripper−subtype⟩ 2finger ⟨/gripper−subtype⟩
    ⟨grip−pos1⟩ (25,0,10) ⟨/grip−pos1⟩
    ⟨grip−pos2⟩ (25,60,10) ⟨/grip−pos2⟩
    ⟨part−position⟩ (350,440,15) ⟨/part−position⟩
    ⟨part−angle⟩  [0,0,0]  ⟨/part−angle⟩
  ⟨/part⟩,
  ⟨part⟩
    ⟨type⟩  tape−roll ⟨/type⟩
    ⟨name⟩ p2 ⟨/name⟩
    ⟨material⟩  plastic  ⟨/material⟩
    ⟨weight⟩ 90 ⟨/weight⟩
    ⟨x−dimension⟩ 40 ⟨/x−dimension⟩
    ⟨y−dimension⟩ 40 ⟨/y−dimension⟩
    ⟨z−dimension⟩ 15 ⟨/z−dimension⟩
    ⟨gripper−subtype⟩ 2finger ⟨/gripper−subtype⟩
    ⟨grip−pos1⟩ (20,0,10) ⟨/grip−pos1⟩
    ⟨grip−pos2⟩ (20,40,10) ⟨/grip−pos2⟩
    ⟨part−position⟩ (355,455,25) ⟨/part−position⟩
    ⟨part−angle⟩  [0,0,0]  ⟨/part−angle⟩
  ⟨/part⟩,
  ⟨part⟩
    ⟨type⟩  body−case ⟨/type⟩
    ⟨name⟩ p3 ⟨/name⟩
    ⟨material⟩  plastic  ⟨/material⟩
    ⟨weight⟩ 30 ⟨/weight⟩
    ⟨x−dimension⟩ 50 ⟨/x−dimension⟩
    ⟨y−dimension⟩ 70 ⟨/y−dimension⟩
    ⟨z−dimension⟩ 18 ⟨/z−dimension⟩
    ⟨gripper−subtype⟩ vacuum ⟨/gripper−subtype⟩
    ⟨grip−pos1⟩ (25,0,9) ⟨/grip−pos1⟩
    ⟨grip−pos2⟩ (25,50,9) ⟨/grip−pos2⟩
    ⟨part−position⟩ (350,440,32) ⟨/part−position⟩
    ⟨part−angle⟩  [0,0,0]  ⟨/part−angle⟩
  ⟨/part⟩,
  ⟨part⟩
    ⟨type⟩  screw ⟨/type⟩
    ⟨name⟩ p4 ⟨/name⟩
    ⟨material⟩  metal ⟨/material⟩
    ⟨weight⟩ 10 ⟨/weight⟩
    ⟨x−dimension⟩ 5 ⟨/x−dimension⟩
    ⟨y−dimension⟩ 5 ⟨/y−dimension⟩
    ⟨z−dimension⟩ 15 ⟨/z−dimension⟩
    ⟨gripper−subtype⟩ screw−driver ⟨/gripper−subtype⟩
    ⟨grip−pos1⟩ (2,2,15) ⟨/grip−pos1⟩
    ⟨part−position⟩ (373,473,41) ⟨/part−position⟩
    ⟨part−angle⟩  [0,0,0]  ⟨/part−angle⟩
  ⟨/part⟩
 ⟨/product⟩
⟨/ final −product⟩
```

B: A solution for the generation of the LSAI as being by Maude

```
⟨ final −agents⟩
  ⟨agent⟩
    ⟨task−id⟩ gap2 −> t(1) ⟨/task−id⟩
    ⟨open−interfaces⟩ empty ⟨/open−interfaces⟩
    ⟨provided−skills⟩
      ⟨ skill ⟩ ⟨type⟩ load ⟨/type⟩ ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ unload ⟨/type⟩ ⟨/ skill ⟩
    ⟨/provided−skills⟩
    ⟨ coalition ⟩ unassigned−human
    ⟨/ coalition ⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ load ⟨/type⟩
        ⟨from⟩ (−10,50,−20) ⟨/from⟩ ⟨to⟩ (0,0,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −> t(2) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨position⟩ (0,0,0) ⟨/position⟩
      ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
        ⟨from⟩ (0,0,0) ⟨/from⟩ ⟨to⟩ (300,0,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −> t(2) ⟨/task−id⟩
    ⟨open−interfaces⟩
      straight(− true) diamond(− true)
    ⟨/open−interfaces⟩
    ⟨layout−mra⟩
      ⟨position⟩ (200,50,0) ⟨/position⟩
      ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨provided−skills⟩
      ⟨ skill ⟩ ⟨type⟩ feed ⟨/type⟩
        ⟨subtype⟩ feeds(body−case) ⟨/subtype⟩
      ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ grip ⟨/type⟩
        ⟨range⟩ 7.0e+1 ⟨/range⟩
        ⟨gripper−type⟩ 2finger ⟨/gripper−type⟩
      ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ 1.25e+2 ⟨/range⟩
      ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ 2.2e+2 ⟨/range⟩
      ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ 2.25e+2 ⟨/range⟩
      ⟨/ skill ⟩
    ⟨/provided−skills⟩
    ⟨ coalition ⟩ r1 f1 g1 ⟨/ coalition ⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ −5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
        ⟨subtype⟩ close ⟨/subtype⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ 5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ 5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ −5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ −3.5e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
      ⟨subtype⟩ open ⟨/subtype⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 3.5e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ −5.0e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 5.0e+1 ⟨/range⟩
    ⟨/ skill ⟩
  ⟨/procedure⟩
⟨/agent⟩,
⟨agent⟩
  ⟨task−id⟩ gap2 −>t(3) ⟨/task−id⟩
  ⟨layout−mra⟩
    ⟨type⟩ conveyor ⟨/type⟩
    ⟨subtype⟩ linear ⟨/subtype⟩
    ⟨position⟩ (300,0,0) ⟨/position⟩
    ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
  ⟨procedure⟩
    ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
      ⟨from⟩ (300,0,0) ⟨/from⟩ ⟨to⟩ (600,0,0) ⟨/to⟩
    ⟨/ skill ⟩
  ⟨/procedure⟩
⟨/agent⟩,
⟨agent⟩
  ⟨task−id⟩ gap2 −>t(3) ⟨/task−id⟩
```

```
⟨open−interfaces⟩
  straight(− true) diamond(− true)
⟨/open−interfaces⟩
⟨layout−mra⟩
  ⟨position⟩ (500,50,0) ⟨/position⟩
  ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
⟨provided−skills⟩
  ⟨ skill ⟩ ⟨type⟩ feed ⟨/type⟩
    ⟨subtype⟩ feeds(tape−roll) ⟨/subtype⟩
  ⟨/ skill ⟩
  ⟨ skill ⟩ ⟨type⟩ grip ⟨/type⟩
    ⟨range⟩ 5.5e+1 ⟨/range⟩
    ⟨gripper−type⟩ 2finger ⟨/gripper−type⟩
  ⟨/ skill ⟩
  ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
    ⟨subtype⟩ linear ⟨/subtype⟩
    ⟨direction⟩ vertical ⟨/direction⟩
    ⟨range⟩ 3.0e+2 ⟨/range⟩
  ⟨/ skill ⟩
  ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
    ⟨subtype⟩ rotational ⟨/subtype⟩
    ⟨direction⟩ horizontal ⟨/direction⟩
    ⟨range⟩ 9.0e+1 ⟨/range⟩
  ⟨/ skill ⟩
  ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
    ⟨subtype⟩ rotational ⟨/subtype⟩
    ⟨direction⟩ horizontal ⟨/direction⟩
    ⟨range⟩ 1.8e+2 ⟨/range⟩
  ⟨/ skill ⟩
⟨/provided−skills⟩
⟨coalition⟩ r2 f2 g2 ⟨/coalition⟩
⟨procedure⟩
  ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
    ⟨subtype⟩ linear ⟨/subtype⟩
    ⟨direction⟩ vertical ⟨/direction⟩
    ⟨range⟩ −5.3e+1 ⟨/range⟩
  ⟨/ skill ⟩,
  ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
    ⟨subtype⟩ close ⟨/subtype⟩
  ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 5.3e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ −9.0e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ −2.8e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
      ⟨subtype⟩ open ⟨/subtype⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 2.8e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 9.0e+1 ⟨/range⟩
    ⟨/ skill ⟩
  ⟨/procedure⟩
⟨/agent⟩,
⟨agent⟩
  ⟨task−id⟩ gap2 −>t(4) ⟨/task−id⟩
  ⟨layout−mra⟩
    ⟨type⟩ conveyor ⟨/type⟩
    ⟨subtype⟩ linear ⟨/subtype⟩
    ⟨position⟩ (600,0,0) ⟨/position⟩
    ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
  ⟨procedure⟩
    ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩ ⟨
      from⟩ (600,0,0) ⟨/from⟩ ⟨to⟩ (900,0,0) ⟨/to⟩
    ⟨/ skill ⟩
  ⟨/procedure⟩
⟨/agent⟩,
⟨agent⟩
  ⟨task−id⟩ gap2 −>t(4) ⟨/task−id⟩
  ⟨open−interfaces⟩
    straight(− true) diamond(− true)
  ⟨/open−interfaces⟩
  ⟨layout−mra⟩
    ⟨position⟩ (800,50,0) ⟨/position⟩
    ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
  ⟨provided−skills⟩
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩ ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ feed ⟨/type⟩
      ⟨subtype⟩ feeds(body−case) ⟨/subtype⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ grip ⟨/type⟩
      ⟨gripper−type⟩ vacuum ⟨/gripper−type⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 2.2e+2 ⟨/range⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 6.0e+1 ⟨/range⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 2.8e+2 ⟨/range⟩
    ⟨/ skill ⟩
  ⟨/provided−skills⟩
  ⟨coalition⟩ r3 f3 g3 ⟨/coalition⟩
  ⟨procedure⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
```

```
        ⟨range⟩ −6.132e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
        ⟨subtype⟩ close ⟨/subtype⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ rotational ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ 6.132e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ rotational ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ −9.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ rotational ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ −2.19e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
        ⟨subtype⟩ open ⟨/subtype⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ rotational ⟨/subtype⟩
        ⟨direction⟩ vertical ⟨/direction⟩
        ⟨range⟩ 2.19e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ rotational ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ 9.0e+1 ⟨/range⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(17/4) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ corner ⟨/subtype⟩
      ⟨position⟩ (900,0,0) ⟨/position⟩
      ⟨angle⟩ [0,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
        ⟨from⟩ (900,0,0) ⟨/from⟩ ⟨to⟩ (950,0,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(9/2) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨position⟩ (900,50,0) ⟨/position⟩
      ⟨angle⟩ [−90,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
        ⟨from⟩ (950,0,0) ⟨/from⟩ ⟨to⟩ (950,300,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(19/4) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ corner ⟨/subtype⟩
      ⟨position⟩ (900,350,0) ⟨/position⟩
      ⟨angle⟩ [−90,0,0] ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
        ⟨from⟩ (950,300,0) ⟨/from⟩ ⟨to⟩ (950,350,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(5) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨position⟩ (600,350,0) ⟨/position⟩
    ⟨angle⟩  [180,0,0]  ⟨/angle⟩ ⟨/layout−mra⟩
  ⟨procedure⟩
    ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
      ⟨from⟩ (950,350,0) ⟨/from⟩ ⟨to⟩ (650,350,0) ⟨/to⟩
    ⟨/ skill ⟩
  ⟨/procedure⟩
⟨/agent⟩,
⟨agent⟩
  ⟨task−id⟩ gap2 −>t(5) ⟨/task−id⟩
  ⟨open−interfaces⟩ straight(− true) ⟨/open−interfaces⟩
  ⟨layout−mra⟩
    ⟨position⟩ (600,300,0) ⟨/position⟩
    ⟨angle⟩  [180,0,0]  ⟨/angle⟩ ⟨/layout−mra⟩
  ⟨provided−skills⟩
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩ ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ hold ⟨/type⟩ ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ hold ⟨/type⟩ ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ base−hold ⟨/type⟩ ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ feed ⟨/type⟩
      ⟨subtype⟩ feeds(screw) ⟨/subtype⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ grip ⟨/type⟩
      ⟨gripper−type⟩ screw−driver ⟨/gripper−type⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 1.5e+2 ⟨/range⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 1.5e+2 ⟨/range⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 2.5e+2 ⟨/range⟩
    ⟨/ skill ⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 1.8e+2 ⟨/range⟩
    ⟨/ skill ⟩
  ⟨/provided−skills⟩
  ⟨coalition⟩ a1 a4 a7 f4 g4 ⟨/coalition⟩
  ⟨procedure⟩
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ −6.401e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
      ⟨subtype⟩ close ⟨/subtype⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ 6.401e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ −5.0e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ linear ⟨/subtype⟩
      ⟨direction⟩ horizontal ⟨/direction⟩
      ⟨range⟩ 5.0e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
      ⟨direction⟩ vertical ⟨/direction⟩
      ⟨range⟩ −1.3786e+1 ⟨/range⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ open−close ⟨/type⟩
      ⟨subtype⟩ open ⟨/subtype⟩
    ⟨/ skill ⟩,
    ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
      ⟨subtype⟩ rotational ⟨/subtype⟩
```

```
        ⟨direction⟩ vertical  ⟨/direction⟩
        ⟨range⟩ 1.378e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear  ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ 5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩,
      ⟨ skill ⟩ ⟨type⟩ move ⟨/type⟩
        ⟨subtype⟩ linear  ⟨/subtype⟩
        ⟨direction⟩ horizontal ⟨/direction⟩
        ⟨range⟩ −5.0e+1 ⟨/range⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(6) ⟨/task−id⟩
    ⟨layout−mra⟩
      ⟨type⟩ conveyor ⟨/type⟩
      ⟨subtype⟩ linear  ⟨/subtype⟩
      ⟨position⟩ (300,350,0) ⟨/position⟩
      ⟨angle⟩ [180,0,0]  ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ transport ⟨/type⟩
        ⟨from⟩ (650,350,0) ⟨/from⟩ ⟨to⟩ (350,350,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩,
  ⟨agent⟩
    ⟨task−id⟩ gap2 −>t(6) ⟨/task−id⟩
    ⟨open−interfaces⟩ empty ⟨/open−interfaces⟩
    ⟨layout−mra⟩
      ⟨position⟩ (300,300,0) ⟨/position⟩
      ⟨angle⟩ [180,0,0]  ⟨/angle⟩ ⟨/layout−mra⟩
    ⟨provided−skills⟩
      ⟨ skill ⟩ ⟨type⟩ load ⟨/type⟩ ⟨/ skill ⟩
      ⟨ skill ⟩ ⟨type⟩ unload ⟨/type⟩ ⟨/ skill ⟩ ⟨/provided−skills⟩
    ⟨coalition⟩ unassigned−human
    ⟨/coalition⟩
    ⟨procedure⟩
      ⟨ skill ⟩ ⟨type⟩ unload ⟨/type⟩
        ⟨from⟩ (350,350,0) ⟨/from⟩ ⟨to⟩ (350,450,0) ⟨/to⟩
      ⟨/ skill ⟩
    ⟨/procedure⟩
  ⟨/agent⟩
⟨/ final −agents⟩
```

References

Banâtre J-P, Fradet P, Le Métayer D (2000) Gamma and the chemical reaction model: fifteen years after. In: WMP, volume 2235 of LNCS, Springer, Berlin, pp 17–44.

Barata J (2005) Coalition based approach for shopfloor agility. Edições Orion, Amadora - Lisboa

Barata J, Ribeiro L, Colombo A-W (2010) A service-oriented shop floor to support collaboration in manufacturing networks. In: Benyoucef L, Grabot B (eds) Artificial intelligence techniques for networked manufacturing enterprises management. Springer series in advanced manufacturing, Springer, London, pp 483–503

Berry G, Boudol G (1998) The chemical abstract machine. Theor Comput Sci 96(1):217–248

Boncheva M, Bruzewicz DA, Whitesides GM (2003) Millimeter-scale self-assembly and its applications. Pure Appl Chem 75(5): 621–630

Candido G, Di Orio G, Barata J, Scholze S (2012) Adapter for self-learning production systems. In: Camarinha-Matos L, Shahamatnia E, Nunes G (eds) Technological innovation for value creation, volume 372 of IFIP advances in information and communication technology. Springer, Boston, p 178

Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2007) LNCS, All about Maude—a high-performance logical framework: How to specify, program, and verify systems in rewriting Logic. Springer Verlag, New York

Clavel M, Palomino M, Riesco A (2006) Introducing the ITP tool: a tutorial. J Univ Comput Sci 12(11):1618–1650

Di Marzo Serugendo G, Frei R (2010) Experience report in developing and applying a method for self-organisation to agile manufacturing. In: IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Budapest, Hungary

Di Marzo Serugendo G, Frei R (2012) Self-awareness in agile assembly systems. Awareness magazine: self-awareness in autonomic systems. http://www.awareness-mag.eu, 12 Mar 2012

Eker S, Meseguer J, Sridharanarayanan A (2003) The Maude LTL model checker and its implementation. In: 10th International SPIN Workshop on Model Checking of Software, LNCS. Springer, New York pp 230–234

Farzan A, Chen F, Meseguer J, Rosu G (2004) Formal analysis of Java programs in JavaFAN. In: Computer Aided Verification (CAV), pp 501–505

Frei R (2010) Self-organisation in Evolvable Assembly Systems. PhD thesis. Universidade Nova de LisboaDepartment of Electrical Engineering, Faculty of Science and Technology, Portugal

Frei R, Di Marzo Serugendo. G (2011a) Advances in complexity, engineering, Int J Bio Inspired Comput 3(4):199–212

Frei R, Di Marzo Serugendo G (2011b) Concepts in complexity engineering. Int J Bio Inspired Comput 3(2):123–139

Frei R, Di Marzo Serugendo G (2011c) Self-organising assembly systems. IEEE Trans Syst Man Cybern Part C Appl Rev 41(6):885–897

Frei R, Di Marzo Serugendo G (2012) The future of complexity engineering. To appear in Central European Journal of Engineering

Frei R, Di Marzo Serugendo G, Barata J (2008a) Designing self-organization for evolvable assembly systems. In: IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Venice, Italy, pp 97–106

Frei R, Di Marzo Serugendo G, Serbanuta TF (2010a) Ambient intelligence in self-organising assembly systems using the chemical reaction model. J Ambient Intell Humanized Comput 1(3):163–184

Frei R, Ferreira B, Barata J (2008b) Dynamic coalitions for self-organizing manufacturing systems. In: CIRP International Conference on Intelligent Computation in Manufacturing Engineering (ICME), Naples, Italy

Frei R, Ferreira B, Di Marzo Serugendo G, Barata J (2009) An architecture for self-managing evolvable assembly systems. In: IEEE International Conferenca on Systems, Man, and Cybernetics (SMC), San Antonio

Frei R, Pereira N, Belo J, Barata J, Di Marzo Serugendo G (2010) Implementing self-organisation and self-management in evolvable assembly systems. In: IEEE International Symposium on Industrial Electronics (ISIE), Bari, Italy, pp 3527–3532

Frei R, Tiwari A, McWilliam R, Purvis A (2012) Self-healing technologies. IEEE Trans Syst Man Cybern Part C Appl Rev

Gross R, Dorigo M (2008) Self-assembly at the macroscopic scale. In: Proceedings of the IEEE 96(9):1490–1508

ISTAG (2001) Scenarios for ambient intelligence in 2010. information society technologies advisory group report. http://www.cordis.lu/ist/istag.htm,

ISTAG (2003) Ambient intelligence: from vision to reality. information society technologies advisory group report. http://www.cordis.lu/ist/istag.htm,

Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36(1):41–50

Meseguer J (1990) Rewriting as a unified model of concurrency. In: Concur Conf., vol 458, LNCS, Springer Berlin Heidelberg, Amsterdam, The Netherlands, pp 384–400

Meseguer J (1992) Conditional rewriting logic as a unified model of concurrency. Theor Comput Sci 96(1):73–155

Onori M (2002) Evolvable assembly systems: a new paradigm? In: 33rd Int. Symposium on Robotics (ISR), Stockholm, Sweden, pp 617–621

Onori M, Semere D, Barata J (2008) Evolvable assembly systems: from evaluation to application. In: Azvedo A (ed) Innovation in Manufacturing Networks, vol 266 of IFIP International Federation for Information Processing, Springer, New York, pp 205–214

Onori M, Semere D, Lindberg B (2011) Evolvable systems: an approach to self-X production. Int J Comput Integr Manuf 24(5):506–516

Phili D, Stoddart JF (1996) Self-assembly in natural and unnatural systems. Appl Chem Int Ed 35(11):1154–1196

Ribeiro L, Barata J, Colombo A (2008) MAS and SOA: A case study exploring principles and technologies to support self-properties in assembly systems. In: 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), pp 192–197

Sasse R Meseguer J (2007) Java+ITP: A verification tool based on Hoare logic and algebraic semantics. In: Denker G, Talcott CL (eds) 6th Int. Workshop on Rewriting Logic and its Applications (WRLA), vol 176(4) of, Electronic Notes in Theoretical Computer Science, pp 29–46

Traian FlorinŞerbănuţă, Grigore Roşu, and José Meseguer.A rewriting logic approach to operational semantics. Information and Computation, 207(2): 305–340, 2009.

Ulieru M, Doursat R (2011) Emergent engineering: a radical paradigm shift. J. of Autonomous and Adaptive Communications Systems 4(1):39–60.