J Ambient Intell Human Comput (2010) 1:163–184
DOI 10.1007/s12652-010-0016-0

ORIGINAL RESEARCH

Ambient intelligence in self-organising assembly systems using the chemical reaction model

Regina Frei · Giovanna Di Marzo Serugendo · Traian Florin Șerbănuță

Received: 8 December 2009 / Accepted: 7 May 2010 / Published online: 16 June 2010
© Springer-Verlag 2010

Abstract This article discusses self-organising assembly systems (SOAS), a type of assembly systems that (1) participate in their own design by spontaneously organising themselves in response to the arrival of a product order and (2) manage themselves during production. SOAS address the industry's need for agile manufacturing systems to be highly responsive to market dynamics. Manufacturing systems need to be easily and rapidly changeable, but system re-engineering/reconfiguration and especially their (re-)programming are manual, work-intensive and error-prone procedures. With SOAS, we try to facilitate this by giving the systems gradually more self-* capabilities. SOAS eases the work of the SOAS designer and engineer when designing such as system for a specific product, and supports the work of the SOAS operator when supervising the system during production. SOAS represent an application domain of ambient intelligence and humanised computing which is not frequently considered, but therefore none the less important. This article explains how an SOAS produces its own design as the result of a self-organising process following the Chemical Abstract Machine (CHAM) paradigm: industrial robots self-assemble according to specific chemical rules in response to a product order. This paper reports on SOAS in general, the specification of the chemical reactions and their simulation in Maude.

Keywords Self-organisation · Evolvable assembly systems · Chemical reaction model

This work was developed while Regina Frei received a PhD grant from the Portuguese Foundation for Science and Technology (SFRH/BD/38608/2007); she currently receives a post-doc grant from the Swiss National Science Foundation.

R. Frei (✉) · G. Di Marzo Serugendo
Birkbeck College, University of London, Malet Street,
London WC1E 7HX, UK
e-mail: work@reginafrei.ch

G. Di Marzo Serugendo
e-mail: dimarzo@dcs.bbk.ac.uk

T. F. Șerbănuță
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

1 Introduction

Suppliers of automation technology provide industrial manufacturing companies with components (called *modules* in this article) such as electric drives, valves, grippers, or motors. The manufacturing companies, active in areas such as consumer electronics, watches industry, food and package industry, or medical equipments, purchase these components for building robotic assembly systems to manufacture products for their own consumers.

Over the years, the needs of manufacturing companies have evolved. In the era of *mass production*, industry produced large quantities of an identical product as rapidly and as cheaply as possible. It was then worth paying the big investments for custom-made installations, which would be disposed of once the product was out of production.

Today, the market tends increasingly towards *mass customisation*, with consumers individually selecting from various product options. Therefore, manufacturing companies now increasingly need agile robotic assembly systems able to cope with dynamic production conditions dictated by frequent changes, low volumes and many variants. There is an accrued awareness in industry that manufacturing systems need to be quickly reconfigurable, have to follow a plug and play approach, avoiding time- and work-intensive re-programming and maintaining productivity under perturbations and failures. Companies are increasingly recognising the advantages of using software agents in manufacturing control systems, where the agents are either pure software agents which are in charge of scheduling tasks, for instance, or are associated with a physical body, that is a robot or a robotic module of a manufacturing system.

Current practice in manufacturing industry does not consider self-* properties at all. Research approaches in the area of changeable manufacturing systems follow various directions: flexibility and reconfigurability concerning pre-planned changes and easy integration of new components (Koren et al. 1999; ElMaraghy 2006); agentified modules for mini-systems containing all the information they need for their execution (Hollis et al. 2003; Hanisch and Munz 2008); bio-inspired techniques such as holonic architectures for optimising shop-floor production (Valckenaers and Van Brussel 2005) or DNA-inspired techniques where the product to be assembled contains the knowledge about how it must be assembled (similar to DNA information) (Ueda 2006); exploiting autonomous software agents for coping with changes during production (Ferrarini et al. 2006), dynamic agent coalitions for production planning (Pechoucek et al. 2000), or for learning how to coordinate and adapt to production (Shen et al. 1998).

These approaches do not consider the mutual interrelations between product, processes and systems. These interrelations are of major importance to assure the agility of manufacturing systems, as argued for instance in (Onori 2002), because any change in the product impacts the assembly processes, which in turn impact the assembly system, and vice-versa. Furthermore, from an engineering point of view, the assembly layout structure is designed manually and individual robots (modules) programs are still written manually. The (re-)engineering of assembly systems and the programming of the robots are error-prone and very time-consuming procedures when compared with the exploitation time of the so obtained system. The situation is even more exacerbated when product life-cycles are getting shorter. Any reduction or simplification of the system design and programming will facilitate the system's reconfiguration and agility. To address the mutual interrelations between product, processes and systems, and to ease designer's task, the main requirement of an SOAS is to actively participate to its own design and management, and to some extent even influence the design of the product itself. This paper proposes to equip assembly systems with self-* capabilities that let them participate in their own design and become gradually more autonomous, and thus easier to manage from the user's (designer or operator) point of view (Frei 2010). To this end, we use the Chemical Abstract Machine paradigm.

1.1 Research on evolvable and self-organising assembly systems

Our research is part of a series of works initially started by Onori (Onori 2002) who coined the term *Evolvable Assembly Systems (EAS)* to refer to assembly systems that can seamlessly adapt to changing product requirements and assembly processes. Research around evolvable assembly systems advances in various directions:

- *Agent-based technologies:* A multi-agent shopfloor control system has been developed with all modules of the shop-floor agentified and coordinating their work during production (Barata 2005; Barata et al. 2006a). The use of agent-based architectures for EAS was compared with the use of service-oriented architectures, modules offer their skills under the form of services (Ribeiro et al. 2008a). Dynamic coalitions (Frei et al. 2008b) have been defined that allow the modules to spontaneously create and change coalitions, as well as to request other modules to join them in order to satisfy all requirements of a task to be fulfilled. Notice that coalitions in assembly systems are closely related to those investigated in *virtual enterprises* and *collaborative networks* (Barata and Camarinha-Matos 2003; Camarinha-Matos et al. 2009), where also chemical programming has been introduced (Arenas et al. 2009).
- *Ontologies and specifications:* EAS ontologies were developed for products, processes and systems (Lohse et al. 2005, 2006; Lohse 2007); for a consumer electronics industrial test case (Maffei and Rossi 2007), and for controlling production (Ribeiro et al. 2008b). To better specify the interfaces between system modules, the intermodular receptacle (a specific interface for EAS modules) (Adamietz 2007) was defined, as well as blueprints for module descriptions (Siltala et al. 2009).
- *Roadmaps and the EUPASS project:* Directions for further research in adaptive assembly technology were formulated (Barata et al. 2006b). Within the EUPASS project[1] (Onori et al. 2008), numerous academic and industrial partners made EAS advance towards industrial deployment of evolvable assembly (Semere et al. 2007), which goes hand-in-hand with our work.
- *Diagnosis:* Evolvable diagnosis (Barata et al. 2007b, c) has been investigated, as well as the suitability of service-oriented architectures for EAS diagnosis (Barata et al. 2007a).

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

– *Self-* properties and emergence:* The possibility of emergent behaviour in EAS was investigated (Barata and Onori 2006), a proposal for self-reconfigurabililty of the assembly system has been proposed (Maffei et al. 2009). SOAS belong to this direction of research, and will be detailed below.

These works have investigated the notion of Evolvable Assembly Systems under different complementary angles (ontology, diagnosis, dynamic coalitions, etc.).

Our contribution to this body of work consists of actually including self-* principles (self-organisation and self-management) both in the design of assembly systems and in their control during production. More precisely, we consider a design of assembly systems that follows a self-organising mechanism, and a control at production time based on self-managing tasks (such as modules monitoring themselves and their neighbours and applying appropriate policies if necessary). Results so far include a preliminary concept for self-organising and self-managing assembly systems (Frei et al. 2008a); an architecture for production control through self-management (Frei et al. 2009a); and preliminary implementation work using Jade and Jess (Frei et al. 2010). For further details on self-organising assembly systems and their background refer to the above-cited literature and (Frei 2010), where also verification and validation are discussed.

This article details exclusively the design of assembly systems following a self-organising mechanism inspired by chemical reactions. This will facilitate the work of designers and engineers when creating systems, and prepare the grounds for easing their re-engineering/reconfiguration when receiving new orders or when facing failures and disturbances. This article does not discuss *production time*, that is, when the robots are self-managing while assembling products.

The self-organising process is triggered by the arrival of a product order: the system's modules spontaneously select each other (suitable or preferred partners) and their position in the assembly system layout. They also derive their own micro-instructions for robot movements. The result of this self-organising process is a new or reconfigured assembly system that will assemble the ordered product. In the terminology of self-organising systems, the appropriate assembly system *emerges* from the self-organisation process—there is no central entity, modules progressively aggregate to each other in order to fulfil the product order. We use the CHAM paradigm (detailed in Sect. 3) for expressing the self-organisation rules: chemical reactions "fire" whenever a module has the appropriate skill(s) to perform one of the tasks specified in the product order (e.g. holding a screw of specified dimensions and screwing it by rotational movements) and whenever two modules have compatible interfaces (e.g. one holding the other and providing additional movements).

1.1.1 *Organisation of this article*

Section 2 presents assembly systems and how they relate to ambient intelligence. Section 3 briefly presents the chemical reaction model and its interest for self-organising systems. Section 4 details the concept, architecture and infrastructure of self-organising assembly systems. Section 5 explains the rule modelling according to the chemical reaction model, while Sect. 6 reports on the specification and simulation work done in Maude. Section 7 briefly discusses the Wermelinger properties of our Maude and CHAM application, and finally, Sect. 8 concludes this article.

2 Ambient intelligence and assembly systems

2.1 Assembly systems and modules

A *product* is composed of *parts* assembled in a specified way, either by human hands or by robots in an *assembly system* (Frei et al. 2008a), which is an industrial installation able to receive parts and join them in a coherent way to form the final product. An assembly system consists of a set of equipment items (*modules* or *manufacturing resource agents* (*MRAs*)) such as conveyors, pallets, simple robotic axes for translation and rotation as well as more sophisticated industrial robots, grippers, sensors of various types, etc. [for details see (Frei et al. 2006) or (Frei 2010)].

Basic module types which are needed for executing assembly operations are as follows (see Figs. 1a and 2): An *axis* is a module which can execute a movement along or around a certain direction (axis). A *gripper* is a device which is mounted on an axis and allows a part to be grabbed, 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 (see Fig. 1a), which - due to the vibrations - delivers them well-aligned on a rail, where a robot can pick them up. 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.

2.2 Evolvable and self-organising assembly systems

An *evolvable assembly system (EAS)* is an assembly system which can co-evolve together with the product and the assembly processes; it can continuously and dynamically

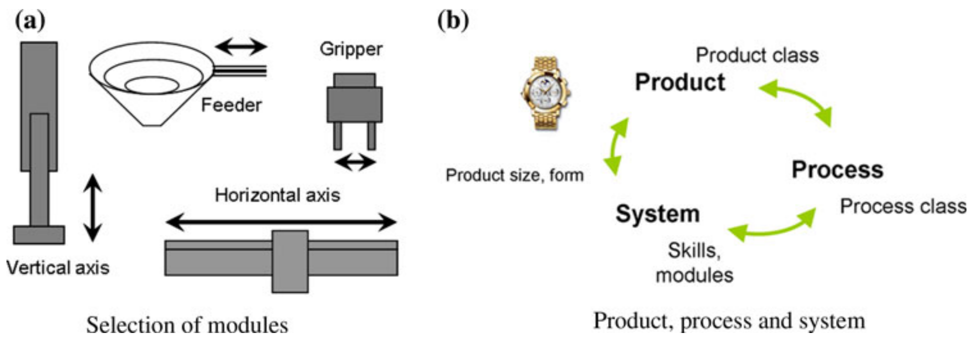


Fig. 1 Aspects of assembly systems

undergo modifications of varying importance—from small adaptations to big changes. Product, production processes and assembly systems are intrinsically related to each other, as illustrated in Fig. 1b. Each *product* belongs to a certain product class (Semere et al. 2008), where a production *process* means a coherent suite of assembly operations which lead to the finished product by assembling a set of parts. Production processes, the product design and the assembly system are intimately linked. 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 vision encompasses seamless integration of new modules independently of their brand or model. Modules carry tiny controllers for local intelligence. Thanks to software wrappers, every module is an agent, forming a homogeneous society with the others, despite their original heterogeneity (nature, type and vendor). Several modules dynamically group to form coalitions offering composite skills. For instance, a gripper alone can only seize and release parts (*simple skills*: “open” and “close”), when it combines with a linear axis, together they can seize a part and move it a long a linear axis (*composite skill*: “pick and place”). Additionally, grippers need to be hold by another module in order to work, as illustrated on Fig. 2.

Self-organising assembly systems (SOAS) are a specific category of evolvable assembly systems where first, a self-organisation process supports the modules in creating the assembly system: in response to an incoming product order, modules progressively select their own partners and their location in a layout; and second, the so obtained assembly system self-manages during production time.

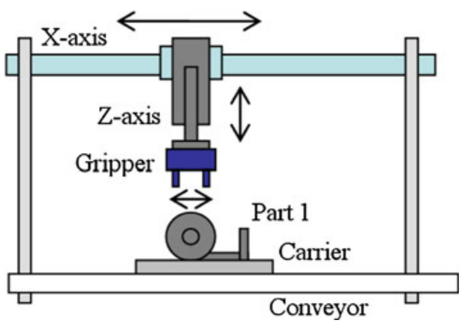


Fig. 2 A robot composed of three modules (*z*-axis, *x*-axis and gripper), manipulating *part 1* on the carrier, which is on the conveyor

Each module carries 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).

2.3 Ambient intelligence in self-organising assembly systems

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. He/she then hands it to the engineer in charge of building the appropriate assembly system. The *engineer*, with his/her know-how, builds an appropriate assembly system: he/she selects a series of modules or complete robots, decides their arrangement in the shop-floor layout, connects them physically and programs their specific movements. 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:

– *Product requirement 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. According to the Evolvable Assembly System paradigm (Onori 2002), the assembly processes and the assembly system technology can even iteratively influence the product design. 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.

We are working on both the layout design and production time aspect described above. The focus of this paper in on a concrete proposal for the layout design.

3 Chemical reaction model

3.1 Gamma and CHAM

The *Gamma chemical reaction model* (Banâtre et al. 2000) has been introduced in 1986 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 (*Reaction Condition*, *Action*). The execution consists of removing from the set the elements that appear in the *Reaction Condition* 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.

A well known example is given by the computation of the sum of a multi-set of integers. Let us consider a multi-set of integers, for instance $\{1, 1, 4, 5, 8\}$. The Gamma program *sum*: $x, y \rightarrow x + y$ defines one reaction *sum* that removes two integers (e.g. 4, 8) from the multi-set and replaces them by their sum (12). This program clearly converges and stops when the multi-set contains only one integer (19), which is then the sum of the integers originally present in the multi-set. The interesting characteristics is that *sum* is applied in parallel to any two different integers of the multi-set. A Gamma program can also contain more than one reaction, applying in an unspecified order and possibly in parallel when their conditions are met.

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.

3.2 Rewriting logic

Rewriting logic was introduced by Meseguer (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. The distinction between equations and rewrite rules is only semantic. They are both executed as rewrite rules by rewrite engines, following the simple, uniform and parallelisable principle of term rewriting. 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 don't overlap.

An immediate advantage of defining a model as a theory in rewriting logic is that one can use the entire arsenal of techniques and tools developed for it to obtain corresponding techniques and tools for the particular defined model; in particular, the model can be executed, which is not the case with most other formalisms. Since rewriting logic is a computational logical framework, "execution" of models becomes logical deduction. That means that one can formally analyze instances of the model or their executions/evolution directly within the rewriting logic definition of their model.

3.3 Maude

Maude (Clavel et al. 2007) is a rewrite engine offering full execution and analysis support for rewriting logic specifications. Maude provides an execution and debugging platform, a breadth-first search (BFS) state-space exploration, and an 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.

3.4 CHAM within rewriting logic

Cham within rewriting logic: As pointed out by Meseguer (Meseguer 1992; Şerbănuţă al. 2009), the Chemical Abstract Machine can be viewed as a particular definitional style within rewriting logic. That is, every Cham is a specific rewrite theory in rewriting logic, and CHAM computation is precisely concurrent rewriting computation.

There is a common syntax shared by all chemical abstract machines, with each machine possibly extending the basic syntax by additional function symbols. The common syntax is typed, and can be expressed as the following order-sorted algebraic signature Ω (we here use the Maude notation for it):

sorts Molecule, Molecules, Solution .
subsorts Solution < *Molecule* < *Molecules* .
op $\cdot : \longrightarrow$ *Molecules* .`*** empty set of Molecules`
op __ : *Molecules Molecules* $\longrightarrow$ *Molecules* .`*** set constructor`
op {|_|} : *Molecules* $\longrightarrow$ *Solution* . `*** membrane operator`
op _◁_: *Molecule Solution* $\longrightarrow$ *Molecule* . `*** airlock operator`

In rewriting logic terms, one may regard each CHAM as a rewrite theory $\mathcal{C} = (\Sigma, ACI \cup Heating - Cooling \cup AirlockAx, Reaction)$, where the signature Σ extends the base Cham signature Ω with constructs for the defined model, and in which the *Heating-Cooling* rules and the *AirlockAx* axiom of the Cham have been made part of the theory's equational axioms, together with the ACI (associative, commutative, and identity applying to '__' and '·', the set constructors). That is, we can more abstractly view the *Reaction* rules of the Cham as being applied *modulo* $ACI \cup Heating - Cooling \cup AirlockAx$.

3.5 Application to SOAS

In our work we intend to exploit the chemical reaction model in the following way:

– Defining a *self-organising mechanism* for layout design inspired by chemical reactions: the main idea is to consider a chemical solution made of all available modules, the tasks specified in the assembly sequence and positions in the shop-floor. The reactions are of different types:

  1. Reactions combining modules according to their physical compatibility. Progressively, according to the chemical reaction model, groups of modules (coalitions) will appear in the chemical solution. Coalitions' skills are the individual skills of the modules as well as composite skills. These concepts are equivalent to those used in virtual organisations and their breeding environments (Camarinha-Matos et al. 2009).
  2. Reactions combining modules and their skills (or coalitions and their composite skills) with assembly tasks. Progressively, modules or coalitions and corresponding tasks will disappear and be replaced by pairs [module/coalition,task], according to matching of available skill and requested task. When the process stops, the chemical solution will contain single modules or coalitions (not used) and pairs of [coalitions, task]. Additional reactions will then occur:
  3. Combining pairs of [coalitions, tasks] with a physical position in the layout;
  4. Combining conveyor modules between two consecutively positioned coalitions.
  5. Finally, the precise module movements will be derived from the combination of the generically specified tasks and the concretely formed layout.

– Using the rewriting process for *self-programming of the modules*: the rewriting process transforms a generic assembly sequence into a specific one containing the list of actual modules, their position and their physical movements. The derivation of the modules' specific movements take into account their coalition partners in the assembly process, their positions and the positions and properties of the actual parts to assemble.

– Establishing a *formal proof* showing that we actually reach a correct assembly system, i.e. an assembly system that is able to build the specified products according to the initially given assembly sequence. The Maude tool actually produces a trace of the chemical reactions that occur in the system, this trace acts as a proof that the desired outcome (finding a layout that can satisfy the GAP) can be reached, or otherwise an error message will be emitted. If this happens, the user may add modules which can provide the missing skills or take other corrective actions.

3.6 Chemical reactions and self-organisation

More generally, one of the main characteristics of self-organising systems is the use of rules for governing interactions among the different components of the system, i.e. these rules define a self-organising mechanism. The notion of chemical reaction model is a generic model that can subsume many self-organising mechanisms, such as the one concerned in this paper about self-assembly of modules as well as other self-organising mechanisms such as stigmergy, swarms, field forces or gossip.

Stigmergy is usually illustrated with ants foraging (dropping/sensing pheromones). In this case, the chemical solution could be formed by ants, the pieces of food, the pheromones. Chemical reactions then update the chemical solution when: ants move; ants drop additional pheromones; pheromone evaporates on its own; or a piece of food is picked up by an ant. For schools of fish or flocks of birds, each fish/bird updates its position according to well known steering rules. The chemical solution is then formed of the set of birds/fish, a single chemical reaction (applying concurrently on all birds/fish) then updates the position of the birds/fish (or the bird/fish itself seen as process with a new state) according to the steering rules.

A *gossip* system mainly consists of peer nodes exchanging information with a set of selected neighbours and updating the information they maintain. The chemical solution is then the set of peers, the chemical reaction describes both the exchange of information and its update.

More generally, the chemical solution consists of the *environment*, the *agents* and any *artefact* they may manipulate; the chemical reactions represent the *rules* defining the self-organising mechanism. In a SOAS, the initial solution contains all modules, the requested tasks, the available positions in the shopfloor, and rules linking modules to tasks, to other modules, and groups of modules to locations in the shopfloor.

In a more general context of Ambient Intelligence, chemical reactions could be used for supporting interactions among the devices of an ambient intelligence system.

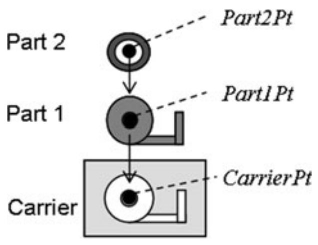


Fig. 3 The adhesive tape roller dispenser

For instance, in a scenario involving a smart home, actions involving one or more home devices occur under the pressure of a corresponding chemical reaction.

4 Running example of SOAS

4.1 Case study example

For illustration purposes, we assume the following simple product to be assembled: an adhesive tape roller dispenser, consisting of a body case ($part_1$) and a tape roll ($part_2$) assembled on top of a carrier, as shown in Fig. 3 [for more details see (Frei et al. 2006)].

4.2 The generic assembly plan

The *Generic Assembly Plan (GAP)* specifies the way a product is to be assembled: it includes the assembly sequence of the different parts and the way they must be joined. Tasks are defined in the form of generic 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 $part_2$) but not *how* (which robotic modules handle which parts and execute which movements at which instant) and is thus independent from any layout. Figure 4 shows the example of a GAP represented as a workflow and written in XML. The four simple illustrated tasks each have an operation type (Op), an object to be handled (Obj), a start point (StPt), 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 $part_1$, and 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

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

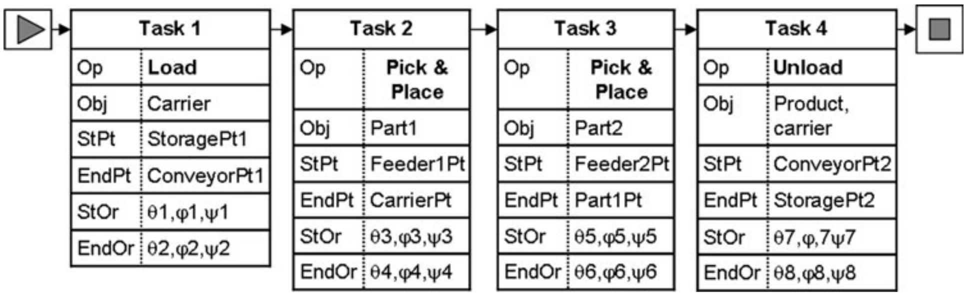


Fig. 4 Example of a GAP written as a workflow

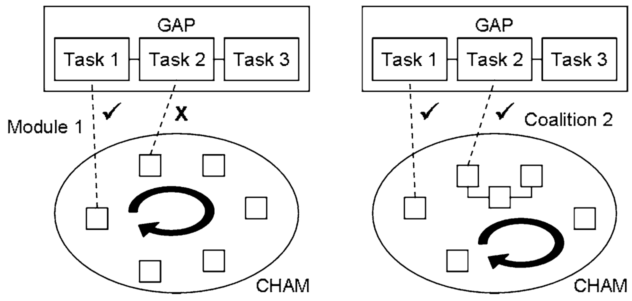


Fig. 5 Self-assembly of coalitions in CHAM

represents the beginning (*IN*), and the square on the right hand side stands for the end (*OUT*).

4.3 Layout design

The layout is incrementally built according to a self-organising process illustrated in Fig. 5: 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.

Figure 6 presents a possible layout for assembling the tape roller. The points and the different orientations of tape roller body case referred to in Fig. 4 are also indicated. In this particular example, a single coalition formed of $robot_1$ and $gripper_1$ moves both $part_1$ and $part_2$ on the *carrier*.

The modelling with Maude of this self-organised process leading to such a layout can be found in Sect. 5; the simulation is in Sect. 6.

4.4 The layout-specific assembly instructions

For realising the assembly, the GAP needs to be transformed into *Layout-Specific Assembly Instructions (LSAI)*. This transformation takes into account the actual modules, the tasks and the parts. The LSAI consists of executable programs for the robotic modules, based on their requested skills. The instructions are generated for a certain layout; if the layout is modified, these instructions must be changed. For an example of an LSAI written as a workflow, refer to (Frei et al. 2009b).

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.

5 Modelling according to the chemical abstract machine

The layout is incrementally built according to a self-organising process modelled by CHAM. It consists of a *molecule solution* and chemical reaction rules. The molecules spontaneously react with each other according to the rules; each time a rule fires, the molecules involved in the reaction are replaced (or *rewritten*) by their new composition (the outcome of the reaction). Afterwards, other rules may apply, and the solution is rewritten again, and so forth, until no rule can be applied any more, and the system has converged to a stable state. The rules are applied in a concurrent and distributed way.

In the case of SOAS, the molecule solution is the set of all modules; rules govern physical combinations of modules and their provided (simple or composite) skills to match requested tasks. The appearance of a GAP in the solution triggers the reaction rules. As a result, modules form coalitions, according to their compatibility rules and composition patterns (compatible sizes and shapes, combination of simple skills providing composite skills), and react with a task specified in the product order.

Just in the same way as molecules have properties on their own and (maybe different ones) when combined, also modules have properties on their own, and potentially different ones in coalitions.

Figure 7 graphically illustrates how modules progressively self-assemble to form coalitions and establish transport links in-between. These coalitions are built according to a set of rules (briefly explained in Sects. 3.5, 5.1–5.6) to progressively match with the tasks defined in the GAP, as shown in Fig. 5. All rules apply at all times; there is no explicitly specified order of application.

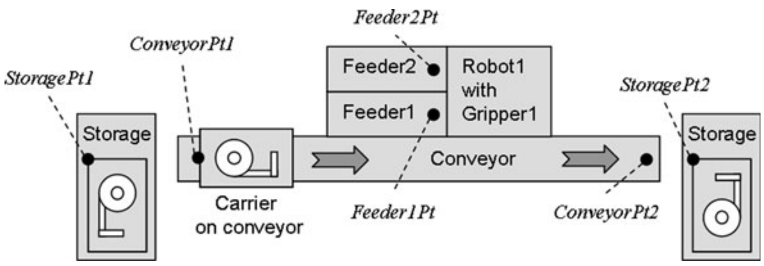


Fig. 6 Possible shop-floor layout for the assembly of the tape roller

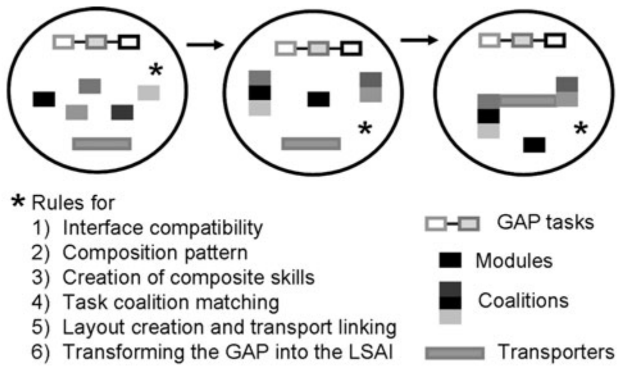

* Rules for
1) Interface compatibility
2) Composition pattern
3) Creation of composite skills
4) Task coalition matching
5) Layout creation and transport linking
6) Transforming the GAP into the LSAI
GAP tasks
Modules
Coalitions
Transporters

Fig. 7 Stepwise application of rules in CHAM (schematic view)

However, implicitly, there is a logical order of application because of the conditions guarding the rules.

5.1 Interface compatibility

Modules match according to the types and characteristics of their physical interfaces. In reality, interfaces are often brand-specific and module-type-specific, and the compatibility between non-related modules is limited, thus avoiding combinatorial explosion.

The number of interfaces a module has determines the maximal number of connections it may establish. For instance, due to physical limitations, a gripper is not able to connect to more than one axis at a time.

5.2 Composition pattern

Typical basic combinations of typical modules are specified as *composition patterns*. For instance, the combination of three linear axes makes a Cartesian robot, and three rotational axes make an ABB-type[3] robot, whereas a linear and a rotational axis build a robot with a cylindrical workspace. In cases where many modules are available, such rules can help reduce the search space by indicating which module combinations typically occur; corresponding coalitions would be formed with priority.

This type of rule can be implemented either using module types or skills (in this work we chose the skills).

5.3 Creation of composite skills

Composite skills are combinations of simple skills which come together when module coalitions are created. Most skill combinations lead to *immediate composite skills*, which are a direct combination (addition) of the simple skills. Some combinations, however, also lead to the emergence of *additional composite skills*. For instance, a gripper has the skills *open-close*, and an axis has *move*. Together, they have *open-close*, *move* and *pick&place*.

5.4 Task coalition matching

When a GAP is introduced, the modules' skills react with the operations requested by the tasks in the GAP. At the same time, the modules react with suitable partner modules, according to their own requirements (interface compatibility and necessary partners, like a gripper needs to be hold by an axis or a robot) as well as the corresponding composition pattern (Sect. 5.2). Once a coalition is formed, a corresponding so-called *dynamic coalition agent* is inserted into the chemical solution. The coalition, again, when suitable, reacts with other tasks and modules. This procedure continues until all the tasks have reacted with modules/coalitions, or until no more module is available. In this case, the task will emit a user alert after a certain time because no satisfactory solution could be found.

If a task reacts with more than one coalition (which will most often be the case), several possibilities can be envisioned:

- Let the user choose.
- Accept the first reaction.
- Accept the reaction which uses the least modules.
- Accept a reaction which uses a coalition that already physically exists, or which requires the least reconfiguration effort.
- Critical tasks or tasks which react only with few modules/coalitions choose their coalitions first; coalitions which react with more coalitions wait for a certain time before choosing their coalitions.

[3] These are commercially available robot types; also a Scara robot, mentioned later, is an example.

A coalition which, in the process of forming itself, cannot find any suitable partner in the module pool, will be discarded. The probability of exactly the same (fruitless) coalition being built again is relatively low, and gets smaller with an increasing number of available modules. Unsuccessful compositions may be recorded to avoid them in future.

5.5 Layout design and transport linking

Generic layout design rules specify how MRAs and coalitions react with places in the layout and with conveyors to establish transportation links in-between the robots. The coalition which was assigned to the first task chooses its place first (random or default position). Coalitions which come later place themselves at default-distances from other coalitions, and ask for transportation skills to move the product from the previous robot to the current. Conveyors will react to such requests and provide the necessary services. A continuous path from IN to OUT must be formed.

5.6 Transforming the GAP into the LSAI (rewriting)

The GAP needs now to be transformed into specific instructions for the selected modules/coalitions in their respective positions; this is the content of the LSAI. The CHAM rewriting technique will be exploited. Commercially available solutions for virtual engineering (Garstenauer 2009) allow the designer to simulate the workspace and verify if a certain point is reachable; to identify potential collisions and cinematic singularities; to calculate trajectories, velocities, cycle times and the stress on robot mechanics.

6 Specifications and simulations in Maude

The rules 5.1–5.4 described above have been implemented in Maude (Clavel et al. 2007), which is a language to model systems through equational and rewriting logic (Meseguer 1990) specifications which provides additional support for executing and analysing (state-space exploration, LTL model checking) the specified models.

The work detailed here was written in Maude version 2.4. Notice 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).

The entire specification will not be discussed in detail here; it can be downloaded from http://www.reginafrei.ch/maude.html. However, we will look into a few issues worth mentioning and explain the most important Maude modules. Before presenting the specification, we highlight some specificities of Maude and our graphical representation.

6.1 From CHAM, through K, to Maude

As discussed in the previous section, CHAM already provides a very good means for modelling SOAS. However, by having only one membrane constructor (‘$\{|\_|\}$’) to wrap a set of molecules and thus help build structured configurations, it makes it hard to write and follow definitions, because, typically, one needs to identify the kind of membrane involved in a rule. The K Framework (Roșu 2007; Roșu and Șerbănuță 2010; Șerbănuță and Roșu 2010), provides a methodology to specify CHAM-like definitions within rewriting logic, while allowing more structure and typing information to be associated to the object representing molecules and solutions. To do that, K replaces solutions by *cells*, which should be regarded as *named solutions*, and, relying on the multi-sorted nature of rewriting logic, it allows molecules and sets of molecules to be typed. For example, using the K methodology, we define the syntax for expressing an MRA by defining two new categories of molecules, Mra, to represent MRAs, and *MraItem* to represent the constituent molecules of an *Mra*, along with a cell constructor “`<mra>_</mra>`”, wrapping a (multi-)set of *MraItem* molecules into an *Mra* molecule; in this case, “mra” would be called the name of the cell. Then, the definition continues, by defining the molecules which can constitute an MRA, such as, name, type, a solution containing the interfaces, and one containing the skills. The *Mra* molecules themselves can be all wrapped in a cell named “mras” which could be typed as a *ConfigItem* molecule. This way of specifying MRAs would correspond to the following Backus Naur Form (BNF)-like definitions, depicted graphically in Figs. 8 and 9:

$$
\begin{aligned}
\mathit{ConfigItem} &::= \ldots \mid <\mathrm{mras}> \mathit{Mra}^* </\mathrm{mras}> \\
\mathit{Mra} &::= <\mathrm{mra}> \mathit{MraItem}^* </\mathrm{mra}> \\
\mathit{MraItem} &::= \ldots \mid <\mathrm{name}> \mathit{MraName} </\mathrm{name}> \\
&\quad \mid <\mathrm{type}> \mathit{Type} </\mathrm{type}> \\
&\quad \mid <\mathrm{skills}> \mathit{Skill}^* </\mathrm{skills}> \\
&\quad \mid <\mathrm{interfaces}> \mathit{Interface}^* </\mathrm{interfaces}>
\end{aligned}
$$

This BNF grammar description directly maps into Maude, by translating grammar non-terminals into sorts, and each production into an operation declaration. For example, “*Mra* :: = <mra> *MraItem** </mra>” maps to the operation declaration `op <mra>_</mra>:Set {MraItem} -> Mra.`, while “*MraItem* ::= <name> *MraName* </name>” maps to `op <name>_</name> : MraName -> MraItem`.

Although the (named) K cells containing types sets of objects give more structure to the configuration than the corresponding CHAM solutions and sets of molecules, one could nevertheless recover the CHAM view of a

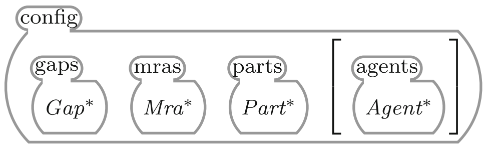


Fig. 8 The top configuration cell

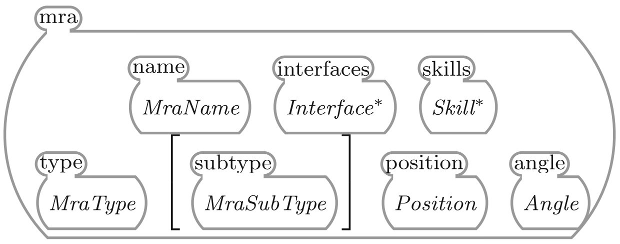


Fig. 9 The mra cell

configuration by forgetting all type information, transforming cell wrappers into unnamed solution constructors and simply adding the name of a cell as a special molecule in the solution. For example, a K-like configuration `<mras><mra><name> r </name><type> robot </type> ... </mra> ... </mras>` could be expressed as the solution $\{|mras\ \{|mra\{|name\,r|\}\ \{|type\ robot|\}...|\}...|\}$. Note that we could have directly used this embedding as a CHAM into Maude, but we prefer to rather go through the K methodology as we find it less error-prone and producing easier-to-read definitions.

6.1.1 The K (visual) notation for rewrite rules

To ease the presentation, the structure of the model and the rules would be presented using the K visual notation for rewrite rules, instead of the pure-ascii Maude notation; nevertheless, we will show how the graphical notation directly corresponds to Maude code. Besides the graphical representation of cells as bubbles having their name attached to the top as a label (see, e.g., Fig. 8), K simplifies writing of reaction rules by making them more local and abstracting away the context. For example, consider the following CHAM rule:

$$cell_1 \triangleleft o_1 \triangleleft M_1\ cell_2 \triangleleft o_2 \triangleleft M_2 \\ \rightarrow cell_1 \triangleleft o_3 \triangleleft M_1\ cell_2 \triangleleft o_2 \triangleleft M_2,$$

where small caps identifiers are constants and capitalized ones are variables, which basically says that if the solution containing cell name $cell_1$ also contains an object o_1—indicated by the fact that the solution can be heated to extract them, through airlocks, from among the remainder of the molecules—and the solution containing cell name $cell_2$ also contains an object o_2, then o_1 should be replaced by o_3. In rewriting logic, we could specify this without using airlocks, but rather relying on the (efficiently implemented) matching modulo ACI, as:

$$\{|cell_1\ o_1\ M'_1|\}\quad \{|cell_2\ o_2\ M'_2|\} \rightarrow \{|cell_1\ o_3\ M'_1|\} \\ \{|cell_2\ o_2\ M'_2|\},$$

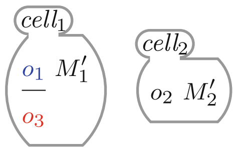
where M'_1 and M'_2 now stand for the sets of molecules comprising the rest of the solution. Moreover, using the K methodology, we can transform solutions into cells, as follows:

$$\langle cell_1\rangle o_1\ M'_1\langle/cell_1\rangle\langle cell_2\rangle o_2\ M'_2\langle/cell_2\rangle \\ \rightarrow \langle cell_1\rangle o_3\ M'_1\langle/cell_1\rangle\langle cell_2\rangle o_2\ M'_2\langle/cell_2\rangle.$$

However, in all the instances presented above, the context matched is much bigger than the actual change. Noticing that this is quite common for interactive systems with complex configurations, the K notation suggests writing the rule such that the rewrites become local, as follows: first write the left-hand side (the context), then underline the parts that change, and finally write their replacement under the line. Thus, our example rule will become:

$$\langle cell_1\rangle \frac{o_1}{o_3}\ M'_1\langle/cell_1\rangle\langle cell_2\rangle o_2\ M'_2\langle/cell_2\rangle,$$

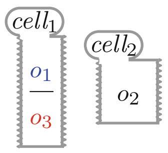
which using the graphical representation of cells can be written as:



A second observation is that there is actually no need for writing the variables M'_1 and M'_2, which are only required for matching, with the sole purpose of abstracting away the remainder of the cell. Instead, we can reduce the symbol load by replacing them with some uniform notation:

$$\langle cell_1\rangle \cdots \frac{o_1}{o_3} \cdots \langle/cell_1\rangle\langle cell_2\rangle \cdots o_2 \cdots \langle/cell_2\rangle,$$

with the intuition that o_1 is replaced by o_3 "in the middle" of $cell_1$, if o_2 can be matched "in the middle" of $cell_2$. Graphically, this is represented by "ripped" cells, that is, ellipses with their margins cut off:



6.2 Structure of the model

We here define the language of our model, which consists of nested cells, containing 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. 8, 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. Notice that the term *agent* used in this context is not to be confused with software agents as mentioned elsewhere in this article.

6.2.1 *Mra*

The mra cell, shown in Fig. 9, contains several cells describing the attributes of a module. For example, the name cell contains a name identifying the MRA, type specifies the type of the module (e.g., *gripper, axis, robot, conveyor*), interfaces contains a collection of interfaces that the module exports, while skills contains a collection of skills which can be performed by the module.

In Maude, one gives life to this intuitive representations through algebraically specified operations. For example, the module declaring the cells which can be part of an MRA named MRA-ITEM, looks as follows:

```
mod MRA-ITEM is
including MRA-TYPE + MRA-NAME + POSITION + ANGLE .
including SET{Skill} + SET{Interface} .
sort MraItem .
op <name>_</name> : MraName -> MraItem .
op <interfaces>_</interfaces> : Set{Interface} -> MraItem .
op <skills>_</skills> : Set{Skill} -> MraItem .
op <type>_</type> : MraType -> MraItem .
op <subtype>_</subtype> : MraSubType -> MraItem .
op <position>_</position> : Position -> MraItem .
op <angle>_</angle> : Angle -> MraItem .
endm
```

This Maude module, named MRA-ITEM consists of several declaration blocks. First, other modules, like MRA-TYPE (declaring the accepted types of MRAs), or Set{Interface} (declaring a set of predefined Interfaces), are included. Next, there is a new sort (MraItem) declaration, which could be seen as a non-terminal in a context-free grammar, which would be used to encompass all possible attributes of an MRA. Then, those attribures are declared, as operations (constructors) having their range MraItem.

Notice that we use XML-like tags to specify cells; moreover, our * in the visual representation, e.g., for

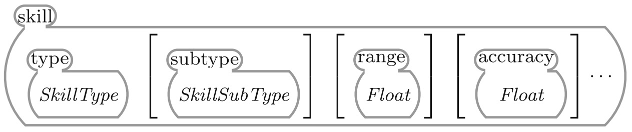


Fig. 10 The skill cell

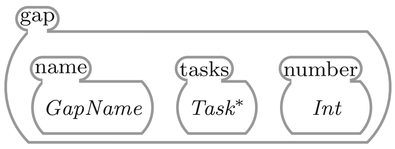


Fig. 11 The GAP cell

Skill and Interface, correspond to algebraically defined (multi-)Sets in our Maude specification.

All potential MRA attributes/items being specified, an Mra is declared as a new sort/type which groups together a set of such attributes under the mra XML-like tag:

```
sort Mra .
op <mra>_</mra> : Set{MraItem} -> Mra .
```

6.2.2 *Skill*

The skill cell, shown in Fig. 10, must contain a mandatory skill type, like *open-close, move, feed, transport, position-carrier*,[4] *store, pick&place, load, unload*, and may contain several optional attributes like subtype (a *move* can for instance be *linear-horizontal* or *rotational-vertical*; a *transport* conveyor can be required to be *linear*, a *corner* or a *junction*), range (how far can it move, how much can it open), and potentially many more.

6.2.3 *Gap*

A GAP contained in a gap cell, shown in Fig. 11, is identified by a name and is composed from the number of products to be assembled, and one or more tasks which need to be performed to achieve the assembly. The definition of a GAP written as a work-flow includes the notion of sequentiality of the tasks. We do, however, not attribute any importance to the order in which modules form coalitions to satisfy the GAP, as in the CHAM model, reactions occur in parallel.

6.2.4 *Task*

Each task cell, shown in Fig. 12, consists of one or more operations, which are abstracted in this model as the

[4] Refers to the indexing devices which assure that a carrier is at the correct position.

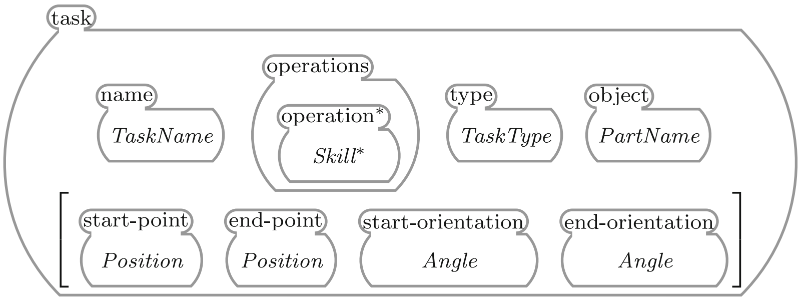


Fig. 12 The task cell

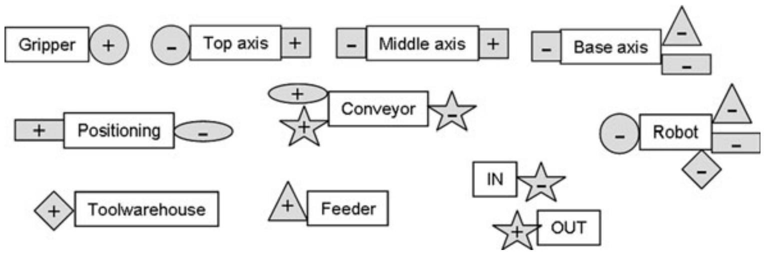


Fig. 13 Interface types as modelled in Maude

`Set` of `Skills` required to perform the task. Additionally, a task would contain the `type` of the task, e.g., *pick&-place*, *transport*, *feeding*, and the name of the `object` (part) it is performed on. Once the task is placed in the layout during the simulation, more attributes would be added such as the start/end point and orientation of the object after the task.

6.2.5 Interface

Interfaces have an `InterfaceType` (symbolically modelled as *circular, square, triangular, oval, straight, diamond* and *star* according to their physical characteristics and functionalities, as illustrated in Fig. 13) and an `InterfaceSign`: plus or minus, referring to the fact that a gripper needs to be held by an axis (gripper is then passive, which is plus), while the axis is holding the gripper (the axis is then active, which is minus). The interface labels were chosen specifically with the goal of avoiding unsuitable compositions, and only to allow them in the correct number; a gripper cannot be held by more then one gripper at once, and it cannot be held by a conveyor or a feeder, for example.

A base axis, as shown in Fig. 13, has three active interfaces and may hold a middle axis or a top axis (square interface), connect to a feeder (triangular interface) and a positioning device (straight interface). Notice that this way of modelling interfaces supports both interface compatibility rules and composition pattern rules.

In our Maude model, the interfaces are specific to each type of connection; even though symbolic, this is close to reality. The following labels for physical compatibility are symbolically used in Maude:

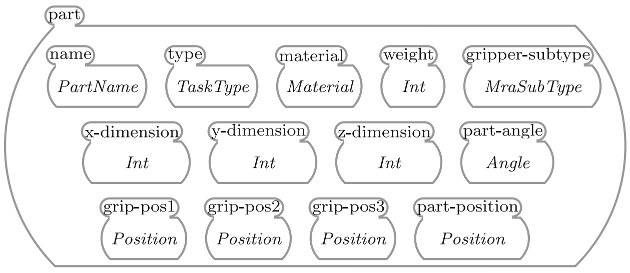


Fig. 14 The part cell

– Circular: gripper to axis or gripper to robot interfaces
– Square: axis to axis or axis to robot interfaces
– Triangular: axis or robot to feeder interfaces
– Diamond: axis or robot to toolwarehouse interfaces
– Oval: conveyor to positioning-device interfaces
– Straight: axis or robot to positioning-device interfaces
– Star: conveyor to conveyor interfaces, including the IN and OUT elements (beginning and end of layout)
– The connection from axes to the ground is not specifically labelled. A robot or axis with only one interface may serve as a basis.

6.2.6 Part

As shown in Fig. 14, a `part` is identified by a `name`, has a `type` (e.g., carrier, body-case, tape-roll, screw), and several properties such as `weight`, `material`, and x/y/z dimensions. *Grip-pos* refers to the position on the part where a gripper should hold the part; *part-angle* and *part-position* refer to the initial or current orientation and position of a part.

6.3 Reaction rules

The reaction rules consist of CHAM-like rules, which match parts of the model to gather information and use that information to transform/evolve the model. To recapitulate, the six types of rules are:

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

The current Maude specification addresses only the first four points. This is enough to simulate and analyse how coalitions are started and formed, driven by the GAP and respecting the composition rules. We will next discuss the most interesting of the rules governing our Maude model, that is the rules for coalition creation and growth. Notice that the empty cell is the neutral element.

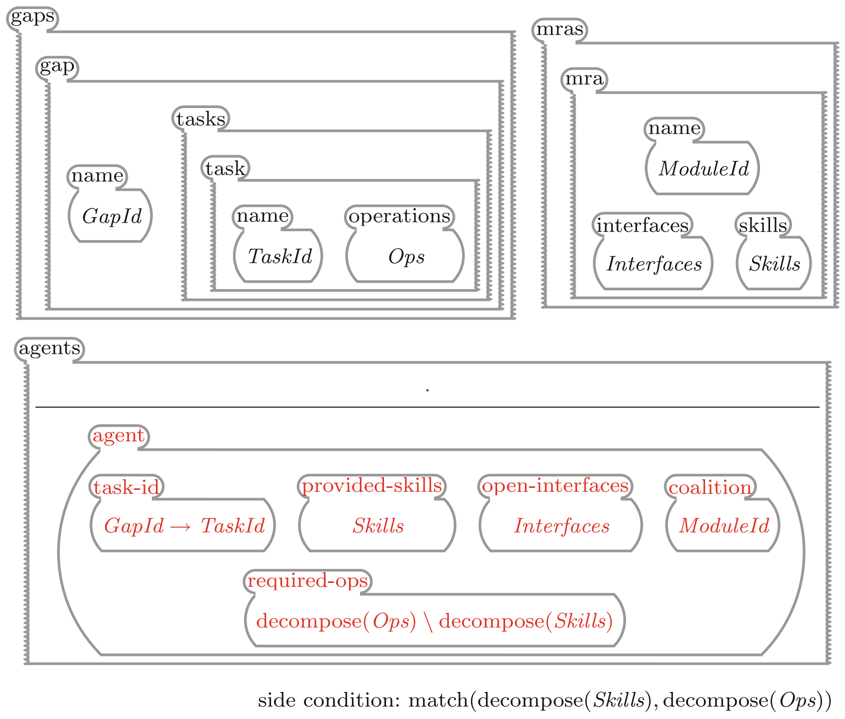
Starting a coalition (rules of type 4) The rule pictured in Fig. 15 specifies that, at any time, an MRA can start a new coalition aimed at addressing a task, provided that at least one of its skills can satisfy at least one of the skills required by the operations corresponding to that task (specified by the *side condition*). If that is the case, then a new agent (standing for a potential coalition) is created, combining features from both the MRA (provided skills, interfaces) and the task (the task-id, the required operations/skills).

The equationally defined predicate `match`, not to be confused with the regular pattern matching performed by Maude, is used to check that the skills of the `mra` are sufficient to complete at least one operations required by the task.

The creation of the new agent is done on the base of an empty cell, here specified by a '·', being underlined in the `agents` cell with its replacement, the new agent cell, being written below the line. The required operations of the new agent are obtained by removing the skills provided by the MRA from those required in each of the task's operations. These required operations are used to find suitable candidates for the coalition, that is modules which can satisfy some of the skills still required to accomplish the task. The `decompose` operation decomposes the complex skills required by tasks into basic skills provided by modules, to facilitate their matching by subsequent rules. (This is the opposite of generating complex skills out of simple ones.)

Two observations need to be made about the visual rules. First, note the use of "ripped" cells (i.e., cells with a zig-zag lateral sides instead of the usual rounded cell walls), used to specify that only a part of the cell is matched in the rule; this idea, which is nothing more than matching modulo associativity, commutativity, and unit, is similar in spirit to the use of *airlocks* (the mechanism used to select one element out of a soup) in CHAMs. Moreover, note that while the change is punctual, it "reads" cells from different levels of the entire configuration. That is why our

Fig. 15 Rule for initiating a coalition



side condition: match(decompose(*Skills*), decompose(*Ops*))

Fig. 16 The coalition initiation rule, as written in Maude

```
crl <gaps>   Gaps                              => <gaps>   Gaps
    <gap>   Gap                                   <gap>   Gap
     <name> GapId </name>                          <name> GapId </name>
     <tasks>   Tasks                               <tasks>   Tasks
      <task>   Task                                 <task>   Task
        <name> TaskId </name>                         <name> TaskId </name>
        <operations> Ops </operations>                <operations> Ops </operations>
      </task>                                       </task>
     </tasks>                                      </tasks>
    </gap>                                        </gap>
    </gaps>                                       </gaps>
    <mras>   Mras                                 <mras>   Mras
    <mra>   Mra                                   <mra>   Mra
     <name> ModuleId </name>                       <name> ModuleId </name>
     <interfaces> Interfaces </interfaces>         <interfaces> Interfaces </interfaces>
     <skills> Skills </skills>                     <skills> Skills </skills>
    </mra>                                        </mra>
    </mras>                                       </mras>
    <agents> Agents </agents>                     <agents>   Agents
                                                  <agent>
                                                   <task-id> GapId -> TaskId </task-id>
                                                   <provided-skills> Skills </provided-skills>
                                                   <required-ops> decompose(Ops) \
                                                      decompose(Skills) </required-ops>
                                                   <open-interfaces> Interfaces </open-interfaces>
                                                   <coalition> ModuleId </coalition>
                                                  </agent>
                                                  </agents>
     if match(decompose(Skills),decompose(Ops)) .
```

graphical notation specifies the change locally, by underlining the part changed and writing its replacement under the line (in red), rather than writing the matching context on both side of a rewrite rule, as customary.

The textual Maude representation of the coalition initiation rule in Fig. 15 is presented in Fig. 16. The rule is introduced by the keyword `crl` (conditional rule) which specifies that the rule is guarded (the guard being introduced by the final `if` ... line). Next, the two sides of the rule are separated by the "=>" keyword and both of them repeat the context required by the rule, while only the right side of the => symbol contains the agent being introduced. Moreover, generic variables (`Gaps`, `Gap`, `Tasks`, ...) are being used to account for the missing parts from each cell. We use the visual representation for rules which extend over several layers of the configuration.

Rule for MRAs joining a coalition (rules of type 1, 2 and 3) The rule depicted in Fig. 17 lets an MRA join a coalition, provided that suitable interfaces and suitable skills are present on both sides. This means that it implements rules of the type 1 (interface compatibility) and 2 (composition pattern) at the same time. The created coalition contains the composing MRAs and has MRA characteristics itself as well.

The use of *Type(+ true)* and *Type(− true)* assures that the joining MRA has a free interface of the same type as one existing in the coalition, but with opposite sign. An additional size constraint could be introduced by giving the interfaces different numbers instead of the (+)/(−), assuring that the opening of the holding interface is at least as large as that of the MRA being held. The values *true* or *false* refer to an interface being free or occupied; also, an occupied interface is not displayed among those available.

If the interfaces match and the side conditions are satisfied, that is, the skills brought by the MRA help advance towards solving the task, and they match the existing skills provided by the coalition in a desirable way, then the MRA is added to the coalition agent: its skills are added to the provided skills, its free interfaces are added to the open interfaces (replacing the interface which is being used to attach the MRA) and newly occupied interfaces are removed, its name is added to the names of the coalition, and the required operations are adjusted to take into account the new skills provided by the MRA.

Technically, rules of type 3 (composite skills) are implemented through the *decompose* equation, which decomposes composite skills into simple skills. For

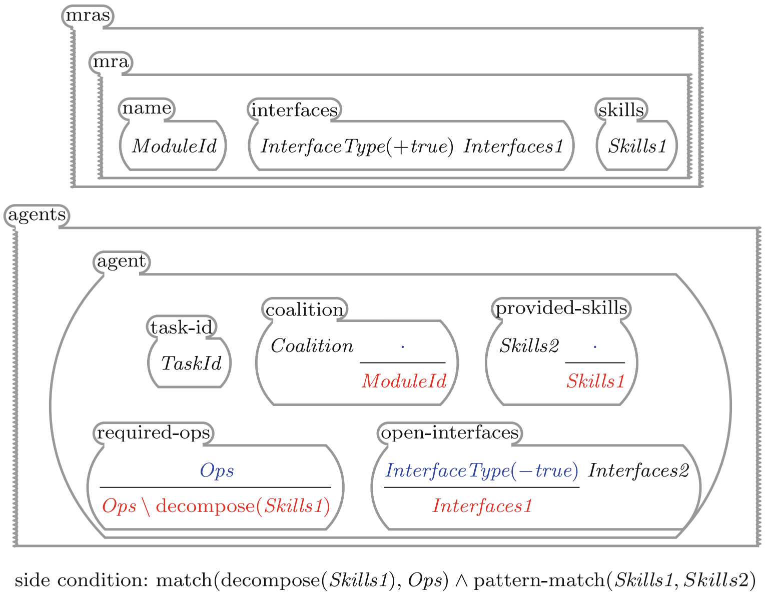

mras
mra
name
ModuleId
interfaces
InterfaceType(+true) Interfaces1
skills
Skills1
agents
agent
task-id
TaskId
coalition
Coalition · / ModuleId
provided-skills
Skills2 · / Skills1
required-ops
Ops / Ops \ decompose(Skills1)
open-interfaces
InterfaceType(−true) Interfaces2 / Interfaces1
side condition: match(decompose(Skills1), Ops) ∧ pattern-match(Skills1, Skills2)

Fig. 17 Rule allowing a module to join an existing coalition

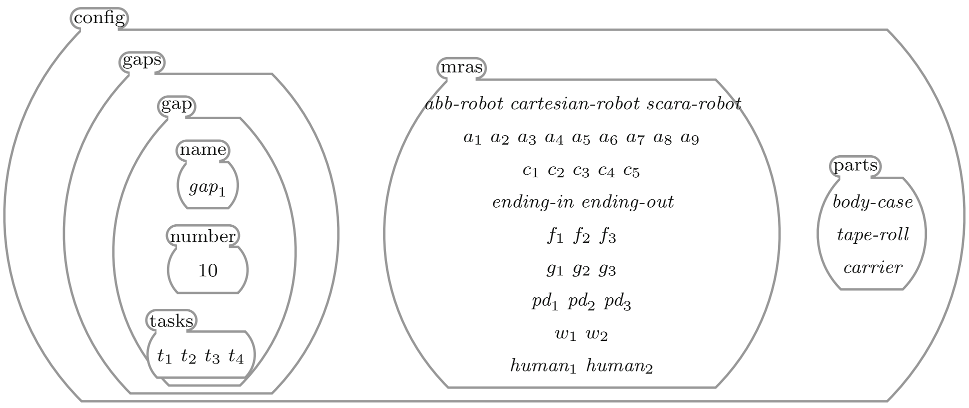


Fig. 18 Initial configuration

example *pick&place* is provided as soon as a coalition contains both *open-close* and *move* skills.

```
eq decompose1(pick&place)
  = open − close(anyPreSkill) move(anyPreSkill).
```

6.4 Case study

Figure 18 is a visual representation of the Maude initial configuration-term used for our case study. It contains only one GAP, gap_1, made of 4 tasks and requesting 10 products to be assembled, several MRAs as well as the product parts. The instances of MRAs specified for our case study are:

– 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 c_1, c_2, c_3, c_4, and c_5;
– Endings of the layout *ending-in* and *ending-out*;
– Feeders f_1, f_2, and f_3;
– Grippers g_1, g_2, and g_3;
– Positioning devices pd_1, pd_2, and pd_3;
– Tool warehouses w_1 and w_2; and
– Humans $human_1$, and $human_2$; technically, the human is wrapped by an MRA with the skills *load* and *unload*.

For example, the *scara-robot* MRA is defined as shown in Fig. 19: the MRA is a *robot* identified as r_3, which has four available interfaces, *circular*, *triangular*, *straight*, and *diamond*, all of them being passive, and provides two rotational *move* skills, one horizontal with range 120°, and one vertical with range 180°.

The GAP introduced in our initial configuration contains four tasks, and is the GAP shown in Fig. 4: the first task is a *load* task, the next two are *pick&place* tasks, and the last one is an *unload* task. For example, task t_3 is defined as shown in Fig. 20.

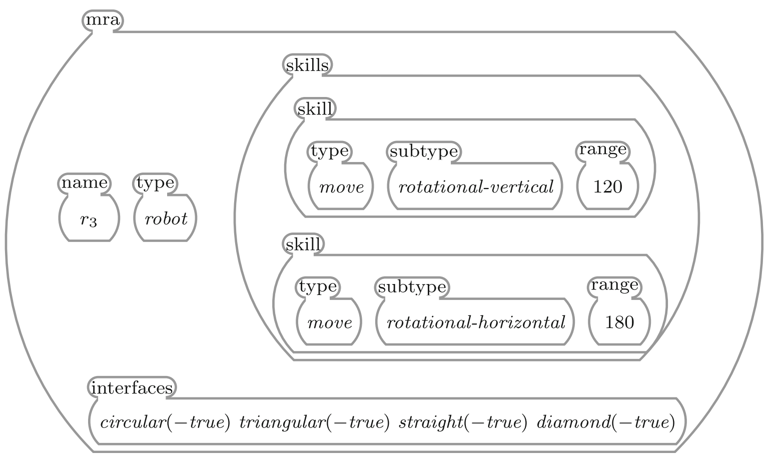


Fig. 19 The Scara robot cell

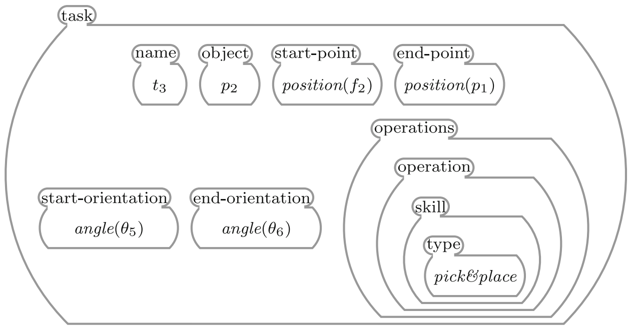


Fig. 20 The task t_3 cell

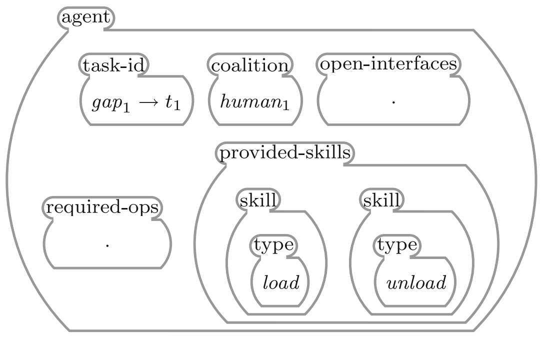


Fig. 22 The task t_1 agent

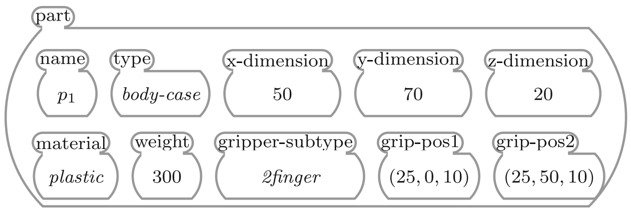


Fig. 21 *Part*$_1$ cell

This configuration term defines a task cell, identified by t_3, acting on part p_2 (which is a tape roll), and requiring a *pick&place* skill to execute an operation taking part p_2 from the feeder f_2 and placing it on top of part p_1 (a body-case).

The parts used for this case study are: a *body-case*, a *tape-roll* and a *carrier*. The body-case cell is shown in Fig. 21.

6.5 Simulation example

The advantage of having a rewriting engine such as Maude is that once we have described the structure of the model and the rules governing its interactions using rewriting rules, the specification becomes *executable*. That is, given a configuration instance such as the one described above, the model can actually be "run" by repeatedly applying the rules to obtain a simulation of its evolution.

Furthermore, analysis tools such as tracing the sequence of rules applied, state-space exploration, and model-checking, are also possible within Maude. Since the rules are non-deterministic and they can potentially apply on any successful match, provided the side conditions are fulfilled, one can have many possible runs of the system, yielding for our example many possible task-coalition matches.

We will here only discuss in detail a sequence of rule applications required for obtaining one solution, as extracted from a specific successful trace provided by Maude. It is observed when directly executing the model (i.e., without searching for all solutions).

First, the rule for initiating an agent is applied for task t_1 from GAP gap_1 and the MRA $human_1$. The rule succeeds since $human_1$ has a skill (*load*) required by task t_1. Moreover, since this was the only skill required by task t_1, the coalition is actually already complete with one participant only, shown by the fact that *required-ops* is empty (pictured as in Fig. 22).

Next,[5] the agent-initiation rule applies on task t_3 and robot r_1, as shown in Fig. 23, since r_1 provides the *move*

[5] The tasks are not addressed in any specific order, as the chemical reaction model works in parallel on all "molecules".

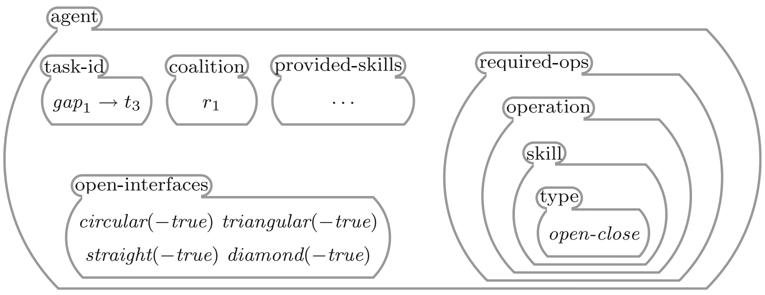


Fig. 23 The task t_3 agent

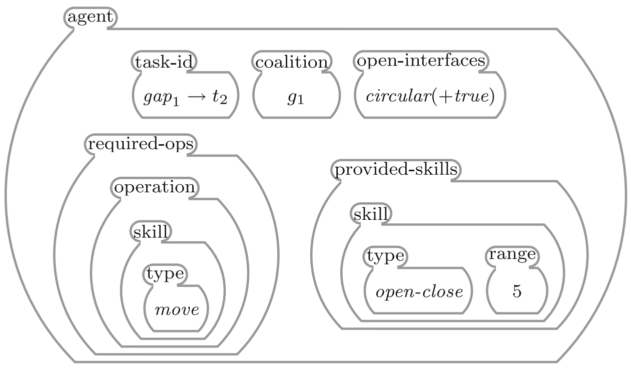


Fig. 24 The task t_2 agent

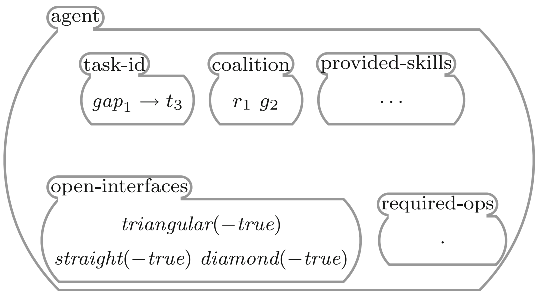


Fig. 25 The task t_3 agent in the completed state

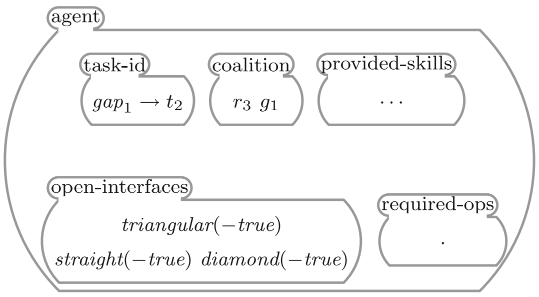


Fig. 26 The task t_2 agent in the completed state

skill required as part of the composite skill *pick&place* of t_3. After creation, the agent provides all skills of r_1 (which we have omitted for reasons of space), and all its interfaces, but is still requires one skill for completion, *open-close*, which together with the already provided *move* skill would yield the composite skill *pick&place*. At this stage, however, the coalition is still incomplete.

Similarly, gripper g_1 initiates an agent for the *pick&place* task t_2, shown in Fig. 24, this time satisfying its *open-close* sub-skill, and still requiring a *move* skill for completion. The coalition has an open interface of the type *circular (+ true)*, and the gripper on of the type *circular (− true)*.

Then the rule in Fig. 17 applies for gripper g_2 to join the coalition initiated by r_1 for solving task t_3, shown in Fig. 25. This is possible because g_2 provides the *open-close* skill which is required by the agent, and, moreover, the agent and g_2 can connect on the compatible *circular* interface. Upon connection their skills and still open interfaces (except for the *circular* one, which is now occupied both on gripper and robot side) are joined, while the required operation becomes empty, meaning this coalition is also complete.

Similarly, the coalition-join rule applies for r_3 to join the coalition started by g_1 to solve task t_2, as shown in Fig. 26.

Finally, MRA *human*$_2$ initiates a coalition for satisfying task t_4, by providing the only required skill, that is *unload*, and thus we have a complete solution to the task assignment problem. The solution found is: t_1 by *human*$_1$, t_2 by coalition $r_3 - g_1$, t_3 by $r_1 - g_2$, and t_4 by *human*$_2$.

6.6 Discussion and remarks

Notice that this article reports on a feasibility study. The work was never intended to deliver a complete industry-compatible prototype. This is why, at this stage, the work does not include existing algorithms for the best shopfloor layout design (Benjaafar et al. 2002), automated trajectory calculation (Chen et al. 2002), automated planning (Ghallab et al. 2004) or any type of elaborate assembly line balancing optimisation (Simaria and Vilarinho 2009).

For a real industry-like implementation working with a CHAM-based control system, we consider that the physical (re-)configurations may only take place once a suitable solution has been found. No real robotic module will be moved before the suggested arrangement has been considered as suitable by a human operator, and it is indeed the operator which will execute the physical reconfiguration. Furthermore, the rules which are currently specified in Maude are not sufficient for a real application yet, as certain requirements have been ignored for the sake of simplicity of the proof-of-concept execution. For instance, certain modules will have further requirements such as the following: a horizontal axis may be able to provide the requested skill of moving a part horizontally, but it will be necessary for a support construction or a vertical axis to hold the horizontal axis in question. Also rules for calculating the overlapping workspaces of collaborating axes have not been added yet. The Maude specifications are currently being extended and completed.

7 Wermelinger's properties in SOAS

According to Wermelinger (1998), to determine if a CHAM specification is correctly defined, it is necessary to consider three properties:

1. *CHAM terminates, that is an inert solution can be reached, which means that either no rule is applicable any more, or there are no components available any more.*
   In our case, the solution will stop reacting as soon as all the tasks in the GAP have been associated with modules that provide the required skills. This means that all tasks of the GAP must have suitable coalitions attributed, and in a complete version, these must have determined their position in the layout, be linked by transport facilities, and have derived their LSAI.
   Otherwise, the CHAM may also terminate because not all the required skills are provided in sufficient number, which means that one or several tasks remain open. At the current stage, there is no rule for dissolving established connections, so a CHAM might remain in that stage, although in a different configuration, a complete and suitable solution would be achievable.
   In the case of our simulations, Maude provides all possible complete and incomplete configurations. In case of a real CHAM implementation, only one system is created at once, and it may be suitable as far as created but incomplete.
2. *The architecture created by CHAM is as intended. That means for SOAS that a layout has been created which can satisfy the LSAI.*
   The skills required by the tasks and the skills provided by the modules will react with each other, and modules which require partners to provide composite skills will react with suitable modules. This assures that the result corresponds well to the desired outcome if the CHAM terminates because all tasks have been attributed with coalitions. If the CHAM terminates because there are no more suitable modules available, the solution is not complete and therefore not suitable.
3. *A reconfiguration does not break the style, that is, all reconfiguration activities are compatible with the rules.*
   In SOAS, the architectural style may be understood as the way modules can combine with each other. The rules for physical compatibility and composition patterns assure that only compatible modules can react with each other. Also the GAP contributes to defining the architectural style, as the task sequence influences the arrangement of the modules. Changes of an individual module or in a module coalition do not alter the adherence of the system to the GAP.

8 Conclusion and Outlook

This article explained self-organising assembly systems (SOAS), which realise agile manufacturing thanks to the chemical reaction model. SOAS are composed of agentified modules carrying distributed local computing power. Self-organisation facilitates the task of designing and changing system layouts for the engineer and provides proactive services by suggesting shop-floor layouts that suit the current product orders. The Chemical Reaction Model is used to describe this self-organising process, and Maude serves as a simulation tool to provide evidence of suitable system behaviour.

Our next steps encompass refining Maude rules, providing (semi-)formal proofs of system properties using the inductive theorem prover and LTL model checker as well as the layout design and transforming the GAP into the LSAI according to rewriting rules (Fig. 12 shows preparatory work). Real implementation will be done with industrial-like robots and using the JESS[6] reasoning engine to enforce rules for self-organisation and policies for self-management (Frei 2010; Frei et al. 2010).

8.1 Resilience and dependability

The introduction of self-* properties to systems is often done with the intention of improving their **resilience**, that is, their *dependability* in case of failures. Dependability is the 'ability of a system to deliver a service that can justifiably be trusted' (Avizienis et al. 2004). For instance, a certain robot must always provide the same service when it is requested, and we rely on this; nothing else may happen. Central to this definition is the notion that it is possible to provide a justification for placing trust in a system. In practice this justification often takes the form of a dependability case which may include test evidence,

[6] http://www.jessrules.com

development process arguments and mathematical or formal proof. The original meaning of resilience refers to the maximal elastic deformation of a material. In the context of computer science and robotics (Bongard et al. 2006; Di Marzo Serugendo et al. 2007) and self-organising systems (Di Marzo Serugendo 2009), resilience means 'dependability when facing changes' (Laprie 2008), or in other words, its ability to maintain dependability while assimilating change without dysfunction. *Dynamic resilience* is a systems capacity to respond dynamically by adaptation in order to maintain an acceptable level of service in the presence of impairments (Di Marzo Serugendo et al. 2010), whereas *predictable dynamic resilience* refers to the capacity to deliver dynamic resilience within bounds that can be predicted at design time.

Now let us apply this to SOAS: *Resilience during system design* means that if the self-* software system generating a layout for a given GAP experiences disturbances or failures, it will recover and either continue the ongoing CHAM process or start a new one, based on the same GAP. Failures during system design could be caused by a module for some reason suddenly becoming unavailable; in this case, the coalition is dissolved[7] and the reaction starts afresh, which means that the system is resilient. The same happens if the coalitions for a GAP remain incomplete because of missing modules. Disturbances could mean that modules and coalitions appear or disappear while the CHAM is executing, or even that the GAP is changed. Also in these cases, the system is resilient.

Resilience during production means that if one or several robotic modules fail, the remaining modules self-organise to build a repaired or alternative system. This may happen either through the self-adaptation and self-management process among the active modules, or through triggering a CHAM self-organisation process as described above, which will modify the current configuration.

8.2 Feasibility—strong and weak points

The strong points of our approach include that we ease the design and programming of agile assembly by equipping the modules with self-* properties. Our approach addresses the requirements for SOAS defined at the end of Sect. 1, such as the mutual interrelations between product, processes and system. Furthermore, the rules are relatively easy to define, and a new system design (if possible) may be received more quickly than if it is created by human, as the self-organised system helps the designers with their work. As for the weak points, the suggested system may not be optimal (there are no rules for optimising performance or minimising the number of modules, yet), and it may be more complicated than necessary. To mitigate this, further constraints and optimisation rules may be added to the CHAM.

To advance towards a real application, certain tools are required, such as a graphical user interface for visualising the solution, which makes the self-organising process more accessible for a human user.

Concerning the **feasibility** of combining self-* behaviours with industrial assembly systems, several issues need to be considered:

CHAM very well suits the needs of agile assembly systems, where a number of heterogeneous robotic components need to form a coherent system which can provide the required assembly operations. The modelling of SOAS with CHAM is quite intuitive and not particularly challenging.

The feasibility of formal/informal proof of properties has been started and is currently being further investigated. It is challenging, but also very promising and worth the effort. A relevant question is how much the assumptions simplify reality, and if they exclude any important characteristics from being considered. This issue needs to be further researched.

The actual application of the CHAM mechanisms to real SOAS is a step which has only been initiated recently, and future results will show the feasibility of the approach in reality. It will require truly parallel/concurrent software execution and the application to a real industrial robotic system.

Finally, the human resistance to change and to systems which could be suspected of getting out of control cannot be neglected. In particular manufacturing industry is rather conservative. To be persuaded of the benefits of self-* systems, realistic prototypes are necessary to demonstrate dependable and safe system behaviour under all conditions.

Acknowledgments The authors also thank the EU-funded coordination action PerAda, http://www.perada.org, for financially supporting travel exchange between authors' institutions.

References

Adamietz R (2007) Development of an intermodular receptacle: a first step in creating EAS modules. PhD thesis, Faculty of Mechanical Engineering, Institute of Applied Computer Science/Automation (AIA), Universität Karlsruhe (TH), Karlsruhe, Germany

Arenas A, Banatre J-P, Priol T (2009) Developing autonomic and secure virtual organisations with chemical programming. In: 11th International symposium on Stabilization, safety, and security of distributed systems (SSS). LNCS, vol 5873. Springer, Berlin, pp 75–89

Avizienis A, Laprie J, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33

[7] As the next step of our ongoing work, a rule to dissolve coalitions must be added to the current CHAM.

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

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

Barata J, Camarinha-Matos L (2003) Coalitions of manufacturing components for shop floor agility: the cobasa architecture. Int J Netw Virtual Organ 2:50–77

Barata J, Onori M (2006) Evolvable assembly and exploiting emergent behaviour. In: IEEE International symposium on industrial electronics (ISIE), vol 4. Montreal, Canada, pp 3353–3360

Barata J, Cândido G, Feijão F (2006a) A multiagent based control system applied to an educational shop floor. In: IFIP International conference on information technology for balanced automation systems (BASYS), Niagara Falls, Canada, pp 119–128

Barata J, Santana P, Onori M (2006b) Evolvable assembly systems: a development roadmap. In: Dolgui A, Morel G, Pereira C (eds) IFAC symposium on information control problems in manufacturing (INCOM), vol 12. Elsevier, St Etienne, France, pp 167–172

Barata J, Ribeiro L, Colombo A (2007a) Diagnosis using service oriented architectures (SOA). In: 5th IEEE International conference on industrial informatics (INDIN), vol 2. Vienna, Austria, pp 1203–1208

Barata J, Ribeiro L, Onori M (2007b) Diagnosis on evolvable assembly systems. In: IEEE International symposium on industrial electronics (ISIE), Vigo, Spain, pp 3221–3226

Barata J, Ribeiro L, Onori M (2007c) Diagnosis on evolvable production systems. In: IEEE International symposium on industrial electronics (ISIE), Vigo, Spain, pp 3221–3226

Benjaafar S, Heragu S, Irani S (2002) Next generation factory layouts: research challenges and recent progress. Interfaces 32(6):58–77

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

Bongard J, Zykov V, Lipson H (2006) Resilient machines through continuous self-modeling. Science 314:1118–1121

Camarinha-Matos L, Afsarmanesh H, Galeano N, Molina A (2009) Collaborative networked organizations: concepts and practice in manufacturing enterprises. Comput Ind Eng 57(1):46–60

Chen H, Sheng W, Xi N, Song M, Chen Y (2002) Cad-based automated robot trajectory planning for spray painting of free-form surfaces. Ind Robot Int J 29(5):426–433

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

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

Di Marzo Serugendo G (2009) Robustness and dependability of self-organising systems: a safety engineering perspective. In: International symposium on stabilization, safety, and security of distributed systems(SSS). LNCS, vol 5873. Springer, Berlin, pp 254–268

Di Marzo Serugendo G, Fitzgerald J, Romanovsky A (2010) Metaself: an architecture and development method for dependable self-* systems. In: Symposium on applied computing (SAC), Sion, Switzerland (page To appear)

Di Marzo Serugendo G, Fitzgerald J, Romanovsky A, Guelfi N (2007) A metadata-based architectural model for dynamically resilient systems. In: ACM symposium on applied computing (SAC), ACM, Seoul, Korea, pp 566–573

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, vol 2648. Springer, Berlin, pp 230–234

ElMaraghy H (2006) Flexible and reconfigurable manufacturing systems paradigms. Int J Flex Manuf Syst 17(4):261–276

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

Ferrarini L, Veber C, Lüder A, Peschke J, Kalogeras A, Gialelis J, Rode J, Wunsch D, Chapurlat V (2006) Control architecture for reconfigurable manufacturing systems: the PABADIS'PROMISE approach. In: IEEE International conference on emerging technologies and factory automation (ETFA), Prague, Czech Republic, pp 545–552

Frei R (2010) Self-organisation in evolvable assembly systems. PhD thesis, Department of Electrical Engineering, Faculty of Science and Technology, Universidade Nova de Lisboa, Portugal

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

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, 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 (2009a) An architecture for self-managing evolvable assembly systems. In: IEEE International conference on systems, man, and cybernetics (SMC), San Antonio, TX, USA

Frei R, Pereira N, Belo J, Barata J, Di Marzo Serugendo G (2009b) Self-awareness in evolvable assembly systems. Technical report, BBKCS-09-07, School of Computer Science and Information Systems, Birbeck College, London, UK

Frei R, Pereira N, Belo J, Barata J, Di Marzo Serugendo G (2010) Implementing self-organisation and self-management in evolvable assembly systems. In: To appear in IEEE International symposium on industrial electronics (ISIE), Bari, Italy

Garstenauer M (2009) Das virtuelle engineering. Comput Autom 9:24–26

Ghallab M, Nau D, Traverso P (2004) Automated planning, theory and practice. Morgan-Kaufman

Hanisch C, Munz G (2008) Evolvability and the intangibles. Assembl Autom 28(3):194–199

Hollis R, Rizzi A, Brown H, Quaid A, Butler Z (2003) Toward a second-generation minifactory for precision assembly. In: International advances robotics program workshop on microrobots, micromachines and microsystems, Moscow, Russia

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

Koren Y, Heisel U, Jovane F, Moriwaki T, Pritchow G, Ulsoy A, Van Brussel H (1999) Reconfigurable manufacturing systems. CIRP Ann Manuf Technol 48(2):6–12

Laprie J (2008) From dependability to resilience. In: IEEE/IFIP International conference on dependable systems and networks, DSN 2008, Fast Abstracts

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

Lohse N, Hirani H, Ratchev S, Turitto M (2005) An ontology for the definition and validation of assembly processes for evolvable assembly systems. In: 6th IEEE International symposium on assembly and task planning: from nano to macro assembly and manufacturing (ISATP). Montreal, QC, Canada, pp 242–247

Lohse N, Ratchev S, Barata J (2006) Evolvable assembly systems: on the role of design frameworks and supporting ontologies. In: IEEE International symposium on industrial electronics (ISIE), vol 4. Montreal, Canada, pp 3375–3380

Maffei A, Rossi T (2007) Development of an ontology to support the EAS: ELECTROLUX Test case. M. Sc. thesis, University of Pisa, Italy

Maffei A, Dencker K, Bjelkemyr M (2009) From flexibility to evolvability: ways to achieve self-reconfigurability and full autonomy. In: International IFAC symposium on robot control (SYROCO), Gifu, Japan

Meseguer J (1990) Rewriting as a unified model of concurrency. In: Concur Conf., Amsterdam, The Netherlands. LNCS, vol 458. Springer, Berlin, 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 International symposium on robotics (ISR), Stockholm, Sweden, pp 617–621

Onori M, Hanisch C, Barata J, Maraldo T (2008) Adaptive assembly technology roadmap 2015, project report-public, document 1.5f, nmp-2-ct-2004-507978

Pechoucek M, Marik V, Stepankova O (2000) Coalition formation in manufacturing multi-agent systems. In: International conference on database and expert systems application (DEXA), London, UK, pp 241–246

Ribeiro L, Barata J, Mendes P (2008a) MAS and SOA: complementary automation paradigms. In: Azevedo A (ed) Innovation in manufacturing networks, vol 266. Springer, Boston, pp 259–268

Ribeiro L, Barata J, Onori M, Amado A (2008b) OWL ontology to support evolvable assembly systems. In: 9th IFAC workshop on intelligent manufacturing systems (IFAC-IMS), Szczecin, Poland, pp 393–398

Roșu G (2007) K: a rewriting-based framework for computations: preliminary version. Technical Report Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign, USA

Roșu G (2010) An overview of the K semantic framework. J Log Algebraic Program. doi:10.1016/j.jlap.2010.03.012

Sasse R, Meseguer J (2007) Java+ITP: a verification tool based on Hoare logic and algebraic semantics. In: Denker G, Talcott C (eds) 6th International workshop on rewriting logic and its applications (WRLA). Electronic Notes in Theoretical Computer Science, vol 176(4), pp 29–46

Semere D, Barata J, Onori M (2007) Evolvable systems: developments and advance. In: IEEE International symposium on assembly and manufacturing (ISAM), Ann Harbor, MI, USA, pp 288–293

Semere D, Onori M, Maffei A, Adamietz R (2008) Evolvable assembly systems: coping with variations through evolution. Assembl Autom 28(2):126–133

Șerbănuță TF, Roșu G (2010) K-Maude: a rewriting based tool for semantics of programming languages. In: Proceedings of the 8th International workshop on rewriting logic and its applications (WRLA'09), LNCS (To appear)

Șerbănuță TF, Roșu G, Meseguer J (2009) A rewriting logic approach to operational semantics. Inf Comput 207(2):305–340

Shen W, Maturana F, Norrie D (1998) Learning in agent-based manufacturing systems. In: AI & manufacturing research planning workshop, Albuquerque, NM, USA, pp 177–183

Siltala N, Hofmann A, Tuokko R, Bretthauer G (2009) Emplacement and blue print an approach to handle and describe modules for evolvable assembly systems. In: International IFAC symposium on robot control (SYROCO), Advances in soft computing, Gifu, Japan

Simaria A, Vilarinho P (2009) 2-antbal: An ant colony optimisation algorithm for balancing two-sided assembly lines. Comput Industrial Engineering 56(2):489–506

Ueda K (2006) Emergent synthesis approaches to biological manufacturing systems. In: 3rd International CIRP conference on digital enterprise technology (DET), Keynote paper, Setubal, Portugal

Valckenaers P, Van Brussel H (2005) Holonic manufacturing execution systems. CIRP Ann Manuf Technol 54(1):427–432

Wermelinger M (1998) Towards a chemical model for software architecture reconfiguration. IEEE Proc Softw 145(5):130–136