Methodologies for self-organising systems: a SPEM approach

Mariachiara Puviani  
Department of  
Information Engineering,  
University of Modena  
and Reggio Emilia,  
Modena, Italy  
mariachiara.puviani@unimore.it

Giovanna Di Marzo Serugendo,  
Regina Frei  
Computer Science and Information Systems,  
Birkbeck College,  
London, United Kingdom  
dimarzo@dcs.bbk.ac.uk,  
regina.frei@uninova.pt

Giacomo Cabri  
Department of  
Information Engineering,  
University of Modena  
and Reggio Emilia,  
Modena, Italy  
giacomo.cabri@unimore.it

Abstract

This article summarises five relevant methods for developing self-organising multi-agent systems. It identifies their most promising aspects and provides a meta-model of each under the form of ‘SPEM fragments’. These fragments can be combined and be part of a larger ad hoc methodology. Self-organising traffic lights were chosen as an illustrating example for the relevant features of the different methods considered.

1. Introduction

A large amount of work has focused on translating natural self-organising mechanisms into artificial systems. These developments remain ad hoc solutions, usually highly dependent on finely tuned parameters. To convince potential industrial buyers of such a technology, more systematic development and validation techniques are needed.

This article investigates five software engineering techniques (section 4, 5, 6, 7 and 8), which explicitly address the development of self-organising systems, applies them to a common self-organisation problem, described in section 2 and presents them using ‘SPEM fragments’. Spem [17] is a meta-modelling language, useful for representing development methodologies, comparing them and possibly combining them (see sections 3 and 10). In section 9, we compare the considered methodologies and in section 10 we give some guidelines to compose an ad hoc methodology. Conclusions and future work follow in section 11.

2. Case study - Traffic Lights Control

To illustrate how the main aspects of the different methods presented in sections 4, 5, 6, 7 and 8 apply to a concrete example, we consider a system composed of cars and traffic light controllers (TL). The global goal of the system is to optimise traffic throughput. Cars need to reach their destination as fast as possible without stopping; TLs need to allow vehicles to travel as fast as possible while mediating their conflicts for space and time at intersections. Traffic lights are independent from each other, and the global system behaviour will appear from the traffic flow of the whole system.

We consider four traffic lights, disposed in a grid as shown in Figure 1. To simplify the system, cars travel along horizontal and vertical lines only. TLs synchronise with each other (two by two) using the traffic flow information given by cars that pass by.

Figure 1. Traffic Light Case Study

3. Spem and fragments

To build a multi-agent system, different existing methodologies can be helpful, but not every methodology provides a solution to any problem in any context. It is therefore convenient to join reusable fragments from existing methodologies. This combines the designer’s need for an individual methodology with the advantages and the experience of existing and documented methodologies.

Methodologies are decomposed into so-called method fragments, which are pieces of process. With these fragments and using SPEM (Software Process Engineering Metamodel)[17], the FIPA Methodology Technical Committee[17] constructed a repository (the method base). In this article, we use the FIPA fragment type. For others see [2], [21] and [8]. Thanks to using a common language, Spem
helps comparing method fragments. A standard language or notation is the best way to enable the reuse of methodologies. The Spem specification is structured as a UML profile, and provides a complete Meta Object Facility (MOF) based metamodel [16]. UML diagrams (e.g. use cases diagrams, activity diagrams, sequence diagrams) facilitate the integration of different methodologies. According to the OMG specification [17], a method fragment is a portion of a methodology, which is a set of the following components:

1. A process specification, defined by a Spem diagram.
2. Deliverables which permit process reconstruction.
3. Preconditions which represent constraints.
4. A list of concepts (related to the methodology’s metamodel) to be defined, designed or refined.
5. Guidelines fragment application and best practice.
6. A glossary of terms used in the fragment.
7. Composition guidelines: descriptions of the context/problem that is behind the source methodology.
9. Fragment dependency relationships.

A software development process is a collaboration between abstract active entities (process roles) which perform operations (activities) on concrete entities (work products). The overall goal of a process is to deliver a set of work products in a well-defined state [17].

Extracted fragments: The fragments presented are particularly relevant because they cover significant features of self-organizing system. When presenting the fragments, we use the following abbreviations: DL for deliverables, PC for preconditions, and GL for guidelines.

4. Adelfe

ADELFE [1] is an agent-based development methodology targeting self-organising systems with decentralised control and emergent functionality. It is based on UML (Unified Modelling Language) and AUML (Agent-UML) [18], and proposes a design process based on the RUP (Rational Unified Process). Adelfe is based on the AMAS (Adaptive Multi-Agent System) theory [3] where cooperation is fundamental. During cooperation an agent tries to:

- anticipate problems;
- detect cooperation failures, called Non Cooperative Situations (NCS): e.g. a perceived signal is ambiguous.
- repair NCS [12].

The designer not only needs to describe what an agent has to do in order to achieve its goal, but also which situations must be avoided (NCS), and how to suppress them when they are detected.

A cooperative agent in the AMAS theory is autonomous and unaware of the global function of the system; it can detect NCSSs and acts to return to a cooperative state; it is not altruistic but benevolent.

Adelfe is divided into six main phases or Work Definitions (WD): Preliminary Requirements, Final Requirements, Analysis, Design, Implementation and Test (see Figure 2, from [19]). Each phase consists of several activities (A), and each activity consists of several steps (S). We focus on how Adelfe creates self-organising systems [19].

**WD2, Final requirements:** A6 supports the identification of the environment (entities and context). It is characterised as being accessible or not, deterministic or not, dynamic or static and discrete or continuous. In A7, the designer identifies the possible cooperation failures (NCS).

**WD3, Analysis:** an interactive tool helps to decide if the use of the AMAS theory is required or not (A11). In A13, entity relationships which are useful for cooperation are studied.

**WD4, Design:** protocol diagrams serve to study agent interactions (A15). Adelfe provides a model to design cooperative agents (A16). A set of generic NCS are suggested, such as: incomprehension, ambiguity, uselessness or conflict. The designer fills in a table for each NCS.

Adelfe and the AMAS theory have been applied to a large range of cases such as flood forecast, robot transport, manufacturing control or emergent programming [1].

**Case study:**

We used the Adelfe Toolkit to produce the documents and diagrams which are related to the self-organizing part of the methodology, and to simulate the created system in the end. Here we report only some activities; for further details see [20].

A1-A3: The stakeholders are TLs or cars. Each actor individually owns some constraints that must be (best) fulfilled: TLs have constraints about the status of their lights: They cannot be both green and red. TLs have to turn green if there is traffic flow. If the traffic flow in one direction is bigger than the flow in the other direction, TLs have to turn green in the direction with the bigger traffic flow. TLs have to turn red if there is no traffic flow. Cars have constraints about their traveling: they have to stop if there is a red light or another car ahead and must go on if the street ahead is...
<table>
<thead>
<tr>
<th>Name</th>
<th>Equal flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Any</td>
</tr>
<tr>
<td>Description</td>
<td>The horizontal traffic flow is equal to the vertical traffic flow</td>
</tr>
<tr>
<td>Condition(s)</td>
<td>$flow_{\text{horizontal}} = flow_{\text{vertical}}$</td>
</tr>
<tr>
<td>Action(s)</td>
<td>The TL decides which flow to consider first.</td>
</tr>
<tr>
<td>Name</td>
<td>Too many cars</td>
</tr>
<tr>
<td>State</td>
<td>Any</td>
</tr>
<tr>
<td>Description</td>
<td>The number of cars exceeds the chosen threshold $\theta$.</td>
</tr>
<tr>
<td>Condition(s)</td>
<td>$flow_{\text{horizontal}} &gt; \theta$ or $flow_{\text{vertical}} &gt; \theta$</td>
</tr>
<tr>
<td>Action(s)</td>
<td>Prevent cars from entering, let them leave quickly.</td>
</tr>
</tbody>
</table>

Table 1. Equal flow NCS and too many cars NCS

empty and the light is green. Constraints on TLs are more important than constraints on cars.

In A11 the AMAS adequacy are verified by answering the AMAS questions. Global (1-8) and local (9-11) level:

1) The global task is not completely specified.
2) Cars and TLs are not subject to a fix order of action.
3) Several attempts are required to find a solution.
4) Dynamic environment: cars can enter/exit the system.
5) The system is physically distributed.
6) The system can consist of many components.
7) The system is non-linear.
8) Cars can appear and disappear dynamically.
9) A component has limited rationality.
10) TLs can perform various actions.
11) TLs and cars adapt to changes in the environment.

We identify the NCS summarised in Table 1.

**SPEM:**

The following fragments describe the most important steps for self-organizing systems according to [13].

**Environmental Description Fragment (A6):** (Figure 3a). DL: an environment definition document and UML diagrams (scenarios) which describe the situation in the environment. PC: a requirement set document that defines the system requirements. GL: determine entities, define the context and characterize the environment.

**Use Cases Description Fragment (A7):** (Figure 3b). DL: a functional description model, and the now completed environment definition document. PC: the preliminary environment definition document. GL: draw up an inventory of the use cases, identify cooperation failures and elaborate on sequence diagrams.

**Adequacy Verification Fragment (A11):** (Figure 4a). DL: the final AMAS adequacy synthesis document. PC: the preliminary software architecture document, described in the Adelfe domain description fragment (see [13] for details). GL: verify the AMAS adequacy at local and global level.

**Agent Identification Fragment (A12):** (Figure 4b). DL: the software architecture document, including the agents. PC: the preliminary software architecture document and the final AMAS adequacy synthesis document. GL: study the entities in their context, identify the potentially cooperative entities and define the agents.

**Interaction Between Entities Identification Fragment (A13):** important for the agent relationships step, but also other fragments can be used to identify agent interactions. See [13] for further details.

**Agent Specification Fragment (A15, A16, A17):** (Figure 5). DL: the AUML protocol diagrams, which specify the interaction language, the final interaction language document and the detailed architecture document, including the agent model. PC: the initial detailed architecture document defined in the Adelfe architecture definition fragment. GL: define and test agent behaviours.

5. The Customised Unified Process

The Customised Unified Process (CUP) [4] is an iterative process that provides support for the design of self-organising emergent solutions in the context of an engineering process. It is based on the Unified Process (UP) [14], and is customised to explicitly focus on engineering macroscopic behaviour of self-organizing systems (see Figure 6).

During the **Requirement Analysis phase** the problem is structured into functional and non-functional requirements, using techniques such as use cases, feature lists and a domain model that reflects the problem domain. Macroscopic requirements (at the global level) are identified.

The **Design phase** is split into **Architectural Design** and **Detailed Design** addressing microscopic issues. Information
Flow (design abstraction) traverses the system and forms feedback loops. Locality is 'that limited part of the system for which the information located there is directly accessible to the entity' [4]. Activity diagrams are used to determine when a certain behaviour starts and what its inputs are. Information flows are enabled by decentralised coordination mechanisms. Therefore, following the idea of design patterns, a set of patterns of decentralised coordination mechanisms are provided [5] (e.g. the Gradient Field).

During the Implementation phase, the design is realised by using a specific language. When implementing, the programmer focuses on the microscopic level of the system.

In the Testing and Verification phase, agent-based simulations are combined with numerical analysis algorithms for dynamical systems verification at macro-level.

The CUP approach has been applied to autonomous guided vehicles and document clustering [4].

Case Study:

The four TLs are defined as agents; cars are considered as entities. The four circles around the TLs define the four localities (see Figure 1). The main goal can be decomposed into sub-goals, which consist of flow optimisation for each TL. The information needed for each TL is the status of its light, and the horizontal and vertical flow in its area.

We chose gradient fields as patterns of decentralised coordination mechanism [15]. Traffic flow automatically propagates information between TLs.

SPEM:

The following fragments were extracted from the architectural design phase of the CUP.

(1) Locality Identification Fragment: (Figure 7a). DL: an UML diagram (e.g. activity diagram) and a localities model. PC: a system requirement document that defines the system requirements given by the users, and an agent model. GL: determine localities for each agent.

(2) Information Flow Definition Fragment: (Figure 7b). DL: an UML diagram (e.g. an activity diagram) and an information flow model that describes the information flow in the entire system, starting from each locality. PC: a system requirement document that defines the system requirements given by the users, and the localities model. GL: first decompose the system behaviour in sub-goals, then determine the information flow.

Figure 7. (a) Locality Identification Fragment, (b) Information Flow Definition Fragment

We decided not to create specific fragments for the patterns of decentralised coordination mechanisms because the pattern list is only useful for the developer to choose the communication and coordination mechanism. This means that it is not possible to define a real fragment. It is, however, important to consider this list while defining a self-organising system. The list is a document (similar to the system requirement document) given by the user and can be integrated into other fragments while building the system.

6. MetaSelf

The MetaSelf approach considers a self-organising system as a collection of loosely coupled autonomous components. Metadata describes components’ functional and non-functional characteristics such as availability levels and
environment-related metadata (e.g. artificial pheromones). The system’s behaviour (e.g. reconfiguration to compensate for component failure) is governed by policies which describe the response of system components to detected conditions and changes in the metadata. When the system is running, both the components and the run-time infrastructure exploit updated metadata to support decision-making and adaptation in accordance with the policies.

The MetaSelf approach proposes both a system architecture and a development process. The MetaSelf system architecture involves autonomous components, repositories of metadata and executable policies, and reasoning services which dynamically enforce the policies on the basis of metadata values. Metadata may be stored, published and updated at run-time by the run-time infrastructure and by the components themselves, both of which can also access policies at run-time (Figure 8).

Guiding policies are high-level goals (e.g. starting or stopping a swarm of robots); bounding policies define environmental limitations; sensing/monitoring policies define reflex behaviour for the components (e.g. if a metadata value reaches a threshold, an action must be taken). Policies may be generic, e.g. replacing a current (slow) component with a higher-performance equivalent. By accessing metadata about current performance, the reasoning engine can determine which of the available components must replace the failing one. Policies can change dynamically.

Figure 8. MetaSelf - Run-time infrastructure [7]

Figure 9 shows the MetaSelf development method. The **Requirement and Analysis phase** identifies the functionality of the system along with self-* requirements specifying where and when self-organisation is needed or desired.

The **Design phase** consists of two steps: (a) the patterns and mechanisms decision step: choice of architectural patterns (e.g. autonomic manager or observer/controller architecture) and adaptation mechanism (e.g. trust, gossip, or stigmergy) (b) the application system design step: instantiate the chosen patterns for the specific application, architecture and policies, design the individual components (agents), select and describe the necessary metadata.

The **implementation phase** produces the run-time infrastructure (Figure 8).

MetaSelf development process has been applied to dynamically resilient Web services [6] and to self-organising industrial assembly systems [9].

Case Study:

We determine the following: **Components**: The four TLs are active, and the cars are passive entities (they can change their state only due to environmental change). **Self-* requirements**: Traffic flow is optimised as a result of self-organisation. **Architectural Pattern**: The generic observer/controller is chosen. Each TL comes with its observer and controller components. **Self-organization mechanism**: Gradient Fields (traffic flows) created by cars movement. Cars follow gradients, TL react to gradients by changing lights. **Coordination mechanism**: TL knows colour of light in both lanes backwards. **Guiding Policies**: Optimise traffic flow on each direction. **Coordination policies for TLs**: Green Wave: 'Keep green light if TL backwards also has green light'. **Bounding policies**: No cars are allowed at the intersections (to avoid blockage): 'If distance to car in outgoing lane is too small, switch to red light'. **Sensing/Monitoring policies**: 'If horizontal flow is bigger than vertical flow (gradients fields), switch the horizontal TL to green and the vertical TL to red.' 'If vertical flow is bigger than horizontal flow, switch vertical TL to green and horizontal TL to red.' **Metadata**: Traffic flow on each incoming lane for each TL; distance to car in each outgoing lane for each TL.

**SPEM:** The most relevant MetaSelf fragments are: Identification of Pattern and Mechanism fragment: (Figure 10a). **DL**: the architectural design pattern and the adaptation and coordination mechanism. **PC**: a list of self-* requirements which represents the required system proprieties. **GL**: define the self-organisation / self-adaptation architectural design pattern and the adaptation and coordination mechanism.
Identification of Software Architecture Fragment: (Figure 10b). DL: the metadata model, the agent model and the policies model. PC: the self-* requirements document, the architectural design patterns document and the adaptation and coordination mechanism document, which describes the necessary conditions to build the system. GL: define design phase entities.

Run Time Infrastructure Definition Fragment: (Figure 11). DL: the final version of the agent model, the execution policies, the metadata repository, and adaptation / coordination services repository. PC: the adaptation and coordination mechanism document, the architectural design pattern document, as well as the agent model, the policies model and the metadata model. GL: define everything which is needed to build the run time infrastructure.

7. A General Methodology

The General Methodology [11] provides guidelines for system development. Particular attention is given to the vocabulary used to describe self-organising systems. For the five iterative steps or phases see Figure 12 from [11].

In the Representation phase, according to given constraints and requirements, the designer chooses an appropriate vocabulary, the abstractions level, granularity, variables, and interactions that have to be taken into account during system development. Then the system is divided into elements by identifying semi-independent modules, with internal goals and dynamics, and with interactions with the environment. Similar interacting variables are grouped, as an increasing number of interaction increases complexity. The representation of the system should consider at least two different level of abstractions: agents are referred to as $x$-agents, where $x$ denotes the level of abstraction relative to the simplest agent in the group. Each of them has goals that can be described and interrelated.

In the Modeling phase, a control mechanism is defined. This mechanism should be internal and distributed. It will ensure the proper interaction between the elements of the system, and produce the desired performance. However, the mechanism cannot have strict control over a self-organizing system; it can only steer it. To develop such a control mechanism, the designer should find aspects or constraints that will prevent the negative interferences between elements (reduce friction) and promote positive interferences (promote synergy). The control mechanism needs to be adaptive, able to cope with changes within and outside the system (i.e. be robust) and active in the search of solutions. It will not necessarily maximize the satisfaction of the agents, but rather of the system. It can also act on a system by bounding or promoting randomness, noise, and variability. A mediator should synchronize the agents to minimize waiting times.

In the Simulation phase, the model(s) developed in the modeling phase are implemented and different scenarios and mediator strategies are tested. Simulation development should proceed in stages: from abstract to particular. Based on the simulation results, the modeling and representation phases can be improved.

The Application phase is used to develop and test model(s) in a real system. Finally, in the Evaluation phase, the performances of the new system are measured and compared with the performances of the previous ones.

This methodology was applied to traffic lights, self-organising bureaucracies and self-organising artefacts [11].

Case Study:

We define the following: Requirements: The main goal is to develop a feasible and efficient traffic light control system.

Representation phase: The system can be modelled on two levels: car level and TL level. The cars’ goal is to maximise their satisfaction $\sigma$. This is the case when travelling freely and without stopping at intersections. The minimal value of $\sigma = 0$ corresponds to a car stopping indef-
ininitely. The TL system’s goal is to maximise the system’s satisfaction \( \Sigma_{\text{system}} \). This is the case when all cars travel as fast as possible, and are able to flow through the city without stopping. The minimal value of \( \Sigma_{\text{system}} = 0 \) corresponds to a traffic jam where all cars stop indefinitely.

**Modeling phase (1):** Find a mechanism that will coordinate TLs so that these mediates between cars, to reduce their friction. This will maximize the satisfactions of the cars and the TLs. As all vehicles contribute equally to \( \Sigma_{\text{system}} \), frictions are minimised through compromise.

**Simulation phase (1):** The simulation consists of an abstract traffic grid with intersections between cyclic single-lane arteries of two types: vertical or horizontal.

**Modeling phase (2):** Each TL keeps a count \( (\kappa_i) \) of the number of cars time steps \( (c + ts) \) approaching only the red light, from a distance \( \rho \). \( \kappa_i \) can be seen as the integral of waiting/approaching cars over time. When it reaches a threshold \( \theta \), the opposing green light turns yellow, and in the following time step it turns red with \( \kappa_i = 0 \), while the red light turns green.

**Simulation phase (2):** Performance measurements showed that this algorithm achieves very good results for low traffic densities, but very poor results for high traffic densities. This is because depending on the value of \( \theta \), high traffic densities can cause the traffic lights to change instantly, which obstructs traffic flow.

**Modeling (3):** The following constraints were added to prevent fast changing: a traffic light will not be changed if the time passed since the last light change is less than a minimum phase \( \psi_{\text{min}} \). Once \( \psi_i \geq \psi_{\text{min}} \), the lights will change when \( \kappa_i \geq \theta \).

**Simulation (3):** For very low traffic densities, the new algorithm performed less effectively than the previous, but still much better than classic methods. For high densities, the new algorithm outperformed. For certain traffic densities, full synchronization was achieved: no car stopped. Thus, satisfaction was maximal for vehicles, traffic lights, and the city as a whole.

For the complete result of these simulations and other traffic light control models with more constraints, see [11].

**SPERM:**

**Control Mechanism Definition Fragment:** (Figure 13). Creates a communication model based on how to optimize the system, which is relevant for self-organising systems. **DL:** an UML diagram which describes the communication protocol of the control mechanism. **PC:** an agent model and a list of constraints. **GL:** divide the labour, to promote synergies and reduce friction. The two produced documents define the model of the specified system and help during the creation of the communication and control model. All these steps can have feedback (not shown in Figure 13).

8. A Simulation Driven Approach

The Simulation Driven Approach (SDA) to build self-organising systems [10] is not a complete methodology, but rather a way of integrating a middle phase into existing methodologies. To describe the environment, suitable abstractions for environmental entities are necessary: the Agent & Artefact metamodel [22] considers agents as autonomous and proactive entities driven by their internal goal/task. Artefacts are passive and reactive entities providing services and functionalities to be exploited by agents through a usage interface.

To overcome many methodologies’ limitations regarding the environment, the notion of environmental agents is introduced. They are responsible for managing artefacts to achieve the targeted self-* properties. Environmental agents are different form standard agents (user agents). These exploit artefact services to achieve individual and social goals.

SDA is situated between the analysis and the design phase, as an **Early design phase** (Figure 14). It assumes that system requirements have just been collected and the analysis has been performed; identifying the services to be performed by environmental agents.

To design environmental agents, a model of agents and environmental resources has to be provided. This model is analysed using simulation, with the goal to describe the desired environmental agent behaviour and a set of working parameters. These are calibrated in a tuning process.

SDA consists of three iterative phases. During the **Modelling phase**, strategies are formulated to make the system behaviours explicit. These behaviours should form an abstract model for possible architectural solutions. To enable further automatic elaborations and reduce ambiguity, these descriptions should be provided in a formal language (not specified in [10]). The model is expected to provide a
characterisation of user agents, artefacts and environmental agents. Feedback loops are necessary in the entire system.

In the **Simulation phase**, the created specifications are used in combination with simulation tools, to generate simulation traces. These will provide feedback about the suitability of the created solution.

As self-organising systems tend to display different qualitative dynamics depending on initial conditions, it may happen that simulations do not exhibit interesting behaviours. In this case, in the **Tuning phase**, the model has to be tuned until the desired qualitative dynamics is reached. The tuning process may provide unrealistic parameter values, or may not reach the required behaviour. This means that the chosen model cannot be implemented in a real scenario. The designer then needs to return to the modelling phase and start again with a new model.

This methodology was applied to collective sorting [10].

**Case Study:**

This approach is very similar to the one described in section 7. They are both based on modeling, simulation and testing. We therefore did not implement the case study with SDA.

**SPEM:**

An important fragment which can be extracted from this methodology is the **Describe the Environment Fragment** (Figure 15), as the main innovations of this methodology are the introduction of environmental agents and the perspective on resources as artefacts. This fragment can be very useful in systems where the environment plays an important role, but it may be difficult to combine with fragments from other methods which do not share this view on the environment. **DL:** UML diagrams, the agent model, the artefact model and the environmental agent model. **GL:** the system requirements. **PC:** first describe the environment by extracting agent and artefacts, then create the environmental agents which manage artefacts to achieve the system’s self-* properties.

![Figure 15. Environment Fragment](image)

**9. Methodologies comparisons**

Figure 16 shows the main characteristics of each methodology as identified by the SPEM fragments above.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Design/Model</th>
<th>Implement</th>
<th>Simulation</th>
<th>Test/Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env. <em>x</em></td>
<td><em>x</em></td>
<td>Agent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelfe</td>
<td></td>
<td>NCS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUP</td>
<td>x</td>
<td>Patterns</td>
<td>Information Flows</td>
<td></td>
</tr>
<tr>
<td>MetaSelf</td>
<td>x</td>
<td>Patterns</td>
<td>Agent Model</td>
<td>Policy Metadata</td>
</tr>
<tr>
<td>GI</td>
<td></td>
<td>x</td>
<td>Fraction</td>
<td>Agent</td>
</tr>
<tr>
<td>GridDriven</td>
<td>x</td>
<td>x Env.</td>
<td>Agent</td>
<td></td>
</tr>
</tbody>
</table>

After studying methodologies for creating self-organising systems and their fragmentation using SPEM, we shed light on advantages and weaknesses of these methodologies.

**AMAS and Adelfe:** If the AMAS adequacy verification fails, the Adelfe process cannot be applied. Additionally, this verification strongly depends on the user, who has to answer by giving his/her opinion. It is for instance difficult to determine if the system is linear or not (question 9, local level). It is also difficult to determine and select the essential NCS. Furthermore, in Adelfe, the global properties of the system are not guaranteed. Adelfe forces the use of cooperation between the agents, which may be beneficial for many systems, but not for all. Several supporting tools help developers build their system.

**The Customized Unified Process:** An important innovation of CUP is the information flow diagram. It is useful for understanding how a system works and how information can be exchanged between agents. The information flow diagram can be integrated in a communication protocol. This methodology also provides a formal technique for evaluation macroscopic properties. Patterns of decentralised coordination mechanisms are important to help the developer find the best mechanism. However, no indications for the actual application of these patterns are given.

**MetaSelf:** High-level and low-level control and predictability are supported through the dynamic enforcement of policies. MetaSelf relies on loosely coupled components, metadata and executable dynamically changing policies. No support for the use of conflicting policies is provided.

**A General Methodology:** The designer receives assistance for understanding how to develop a system, but no information about actual development. The system model needs to be chosen, and simulation attempts realised. No guidance for simulation program choice is given. This can be negative because a methodology is supposed to give guidelines to users. Besides, the author states that the novelty of this methodology lies in the vocabulary used to describe self-organising systems, but then the vocabulary is not really defined. The choice of appropriate vocabulary for the actual system is left to the user.

**A Simulation Driven Approach:** SDA considers the environment and agents as first class entities (artefacts). It
is not a complete methodology per se, it must be combined with other methodologies. Elements like environmental agents and artefacts are not necessarily supported by other existing methodologies.

10. Building a methodology

We sketch a possible methodology for self-organisation, using the prioritisation algorithm by [23]. In Figure 17 we suggest where to position the method fragments within the five development phases. A stands for Adelfe, S for the simulation driven approach, M for MetaSelf, G for the general methodology and C for the customised unified process. Grey bullets refer to fragments which are not defined here.

Figure 17. Fragments positioning

Figure 18 shows an attempt to integrate the fragments. Solid lines indicate a connection that does not need any change, while dashed lines indicate connections which require modifications in one of the concerned fragments because the used concepts are not perfectly equal. The numbers in the following text refer to the numbers in Figure 18.

Notice that there are no direct connections between the first, second and third fragment; they are connected via goals (some connecting fragments or entities are needed). As these three fragments come from the same methodology (Adelfe), their connection can be verified. For our purpose, we do not need the complete fragment that Adelfe uses (domain description fragment). Only environment definition (1) needs to be adapted in order to get a document that can be used to identify the AMAS adequacy. This can be done creating an ad hoc fragment. In the second fragment, use case description (2), the NCS are identified, but afterwards, there is no specific fragment that will use them. Therefore they need to be integrated when identifying the agents and the coordination mechanism, as the NCS will interfere with the agents’ behaviours. The goals of the system are defined with the help of existing scenarios. The goals, along with the requirement set and the AMAS adequacy synthesis (3), are useful for building the self-* requirements document, needed for the identification of pattern and mechanism fragment (4). The role concept may be useful for the identification of agents and how they satisfy their goals: software architecture fragment (5). The self-* requirements can be used instead of the system requirements for the locality identification fragment (6), because they are very similar and they do not need to be adjusted. What needs to be changed are the input models for the information flow definition fragment (7). The policies model was inserted because it is composed by the communication policies which must be taken into account while building the information flow model. For the run time infrastructure fragment (8), data coming from the information flow model are necessary in order to define the adaptation and coordination services. Notice that the methodology is not complete (missing implementation and simulation), but can be a useful guideline.

11. Conclusions

Developers of self-organising systems need the support of customised design methodologies. Most of the existing methodologies focus on specific characteristics, such as the communication process, collaboration between the agents, and the environment. They can therefore never be sufficiently generic to cover all the existing problems. This is
why we have studied how these methodologies create the required features of self-organisation. With the help of the SPEM approach, we suggest that designers could extract method fragments and reuse them. To create customised methodologies, the readily available fragments can be used in combination with ad hoc created application-specific fragments.

1http://www.fipa.org/activities/methodology.html
2http://www.omg.org/spec/MOF/2.0/
3http://www.omg.org/spec/SPEM/
4http://www.irit.fr/ADELFE/Download.html

References


