A method fragments approach to methodologies for engineering self-organising systems

MARIACHIARA PUVIANI, University of Modena and Reggio Emilia, Italy
GIOVANNA DI MARZO SERUGENDO, University of Geneva, Switzerland
REGINA FREI, Imperial College London, United Kingdom
GIACOMO CABRI, University of Modena and Reggio Emilia, Italy

This article summarises five relevant methods for developing self-organising multi-agent systems. It identifies their most relevant aspects and provides a description of each one under the form of *method fragments expressed using SPEM* (Software & System Process Engineering Meta-Model). The use of a "meta-model" to describe fragments facilitates the comparison of the methods and their respective fragments. These fragments can be combined and be part of a more general *ad hoc* methodology, created according to the needs of the designer. Self-organising traffic lights controllers and self-organising displays are chosen as case studies to illustrate the methods and to underline which fragments are important for self-organising systems. Finally, we illustrate how to augment PASSI2, an agent-based methodology which does not consider self-organisation aspects, with some of the identified fragments for self-organisation.

1. INTRODUCTION

Nature provides many examples of self-organising systems: from *non-living systems* (e.g. Bénard convection cells, crystal growth formation, glass cracks, sand dune ripples, mud cracks) to *living systems*, such as biological processes of pattern formation (e.g. zebra stripes, giraffe coat patterns, vermiculated rabbit fish, cone shells, plants patterns) and *collective behaviour* (cells and immune system, social insects [swarms] or human behaviour: social networks, small worlds, markets, game theory), etc.

These systems demonstrate desirable complex properties such as robustness, resilience or self-reconfiguration, while the individual entities forming these systems are relatively simple. A large body of work has focused on translating self-organisation mechanisms such as stigmergy, swarm behaviour, firefly synchronisation, trust, etc. into artificial systems. Artificial (engineered) systems, made of relatively simple individual entities interacting with each other and their environment then produce results that go beyond the individuals' behaviour.

Although they act as proofs of concepts, some of these examples remain *ad hoc* solutions, usually highly dependent on finely tuned parameters. In order to convince potential industrial buyers of such technologies, more systematic development and validation techniques are necessary.

The aim of this article is to report on current development methods specifically addressing self-organisation aspects. It identifies relevant features of each method and

Authors' addresses: M. Puviani, Department of Science and Methods for Engineering, University of Modena and Reggio Emilia, Reggio Emilia, Italy.
G. Di Marzo Serugendo, Department of Science, University of Geneva, Switzerland.
R. Frei, Department of Electrical and Electronic Engineering, Imperial College London, UK.
G. Cabri, Department of Information Engineering, University of Modena and Reggio Emilia, Modena, Italy.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

provides corresponding fragments described using SPEM (Software & System Process Engineering Meta-Model). Acting as a common language, SPEM is useful to express features of different methods in a common formalism. This allows the comparison and possible composition of features from different methods. Moreover the use of fragments is useful to underline, identify and analyse in details, the self-organisation features used in each method. According to their needs, developers can reuse fragments for self-organisation in their own methods, to create a better self-organising system.

Organisation of the article:
In section 2, we briefly introduce our working definitions of self-organisation and emergence, and then proceed to section 3 by presenting the concept of *method fragments* for analysing and composing methodologies. Section 4 describes two case studies that we will later use to illustrate the reviewed development methods. We then investigate five software engineering methods (sections 5 to 9), which explicitly address the development of self-organising systems. We report on the case studies applications as well as the most relevant fragments. In section 10, we extract, analyse and compare the most relevant fragments from each method, and then, in section 11, we create the foundations for composing customised *ad hoc* methodologies and we illustrate how to augment the PASSI2 (evolution of PASSI) methodology with self-organising features. Conclusions follow in section 12.

2. SELF-ORGANISATION AND EMERGENCE

The scientific community has suggested many different definitions of self-organisation and emergence. We report on a selection. It is not within the scope of this article to review the literature on self-organising systems. The interested reader may refer to [Bonabeau et al. 1999] and [Camazine et al. 2001].

In a definition inspired by swarm intelligence, self-organisation can be seen as 'a set of dynamical interactions whereby structures appear at the global level of a system from interactions among its lower-level components. (...) The rules specifying the interactions are executed on the basis of purely local information, without reference to the global pattern' [Bonabeau et al. 1999].

A more general definition addressing engineered systems is: 'Self-Organisation is the mechanism or the process enabling a system to change its organisation without explicit external command during its execution time' [Di Marzo Serugendo et al. 2005].

In [De Wolf and Holvoet 2005] we find: 'Self-organisation is a dynamical and adaptive process where systems acquire and maintain structure themselves, without external control.' A system described as self-organising in [Gershenson 2007] is one 'in which elements interact in order to achieve dynamically a global function or behaviour.' In an engineered self-organising system 'the elements are designed to dynamically and autonomously solve a problem or perform a function at system level.'

The definition of emergence is more controversial than the definition of self-organisation. Consider the following: 'An emergent phenomenon is a functionality, structure/organisation, characteristics or property of a system not explicitly coded in the local components, visible by an observer at the macro-level but not necessarily at the micro-level' [Di Marzo Serugendo et al. 2005]. Similarly, 'a system exhibits emergence when there are coherent emergents[1] at the macro-level that dynamically arise from the interactions between the parts at the micro-level. Such emergents are novel with regard to the individual parts of the system' [De Wolf 2007].

For some researchers, the concept of *emergence* is strongly connected to self-organisation. According to [Camazine et al. 2001]: 'Self-organisation is a process in which pattern at the global level of a system emerges solely from numerous interac-

[1]The term *emergents* refers to the emerging phenomenon, independent from its nature.

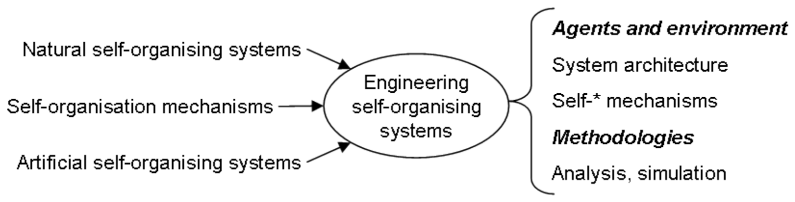

Natural self-organising systems
Self-organisation mechanisms
Artificial self-organising systems
Engineering self-organising systems
Agents and environment
System architecture
Self-* mechanisms
Methodologies
Analysis, simulation

Fig. 1. Self-organising systems

tions among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern.'

A view focusing on evolutionary aspects is presented in the Principia Cybernetica Web [Heylighen 1999], clarifying that self-organisation is different from emergence:

> 'Self-organization is a process where the organization (constraint, redundancy) of a system spontaneously increases, i.e. without this increase being controlled by the environment or an encompassing or otherwise external system. Self-organization is basically a process of evolution where the effect of the environment is minimal, i.e. where the development of new, complex structures takes place primarily in and through the system itself.'

2.1. Engineering self-organising systems

It is intuitive to think of the active elements that compose an artificial self-organising system as autonomous agents [Wooldridge 2002]. Agents follow local rules that apply on the basis of local information. These rules may be fixed or can evolve as the system executes. These rules constitute the self-organisation mechanism that drives the behaviour of the system in a decentralised way. Agents are embedded into an environment. They have a certain degree of autonomy, pro-activity and ability to interact with each other and the environment.

Self-organising systems are thus composed of the following 'ingredients': *agents*, *self-organisation mechanisms* and a *dynamic environment* [Di Marzo Serugendo 2009]. The environment plays a very important role in self-organising systems because it provides events that perturb the system and thus lead it to change its behaviour.

Figure 1 shows how the engineering of self-organising systems usually takes inspiration from natural self-organising systems. Their behaviour follows self-organisation mechanisms used as metaphors to derive / translate self-* mechanisms for artificial systems, taking into account any constraint related to the artificial system to build.

The engineering process includes: designing and implementing agents, considering the environment, establishing the system architecture, selecting / designing and implementing self-organising mechanisms, using a development method for modelling and simulation.

3. METHOD FRAGMENTS

Different existing methodologies can be helpful to build a multi-agent system, but not every methodology provides a solution to any problem in any context. A methodology may be so focused on a specific problem that it is difficult to reuse the same methodology for different cases; on the other hand, methodologies may also be too generic to be easily applicable. It is therefore convenient to join method parts or *reusable fragments* from existing methodologies. This combines the designer's need for a specific methodology with the advantages and the experience of existing and documented methodologies.

Method fragments are process parts in which every methodology can be decomposed. Different definitions or notions of fragments exist:

According to [Brinkkemper et al. 1998], a method fragment is a coherent piece of information system development. Two types of method fragment exist: the *process fragment* that describes stage, activities and tasks; and the *product fragment* that concerns the structure of a process product (deliverables, diagrams, etc.).

Method chunk [Ralyté and Rolland 2001] defines a fragment as a consistent and autonomous component that represents a portion of process and its resulting work products. It is represented using a meta-model (UML – Unified Modelling Language – notation) composed of two parts: the *process aspect* and the *product aspect*.

The OPEN Process Framework (OPF) method fragment [Firesmith and Henderson-Sellers 2002] is based on the *object-oriented process, environment, and notation (OPEN) approach*, and it is generated and stored in a repository with all its guidelines based on the OPF meta-model. This approach considers five meta-classes that produce a method fragment (process or product fragment).

The FIPA method fragment[2] is a reusable part of a design process composed of a set of activities performed by process roles in order to produce a kind of artefact (work product). It is based on the process description model from the OMG SPEM (Software & Systems Process Engineering Meta-Model)[3] and uses the related notation.

In this article, we use the FIPA fragment type because it adopts the use of SPEM, which is more suitable for describing methodologies and integrating different fragments. SPEM is defined as a meta-model used to define systems development processes and their components. It uses UML as a notation and takes an agent-oriented approach. Its main goal is to accommodate a large range of development methods and processes (fragments) of different styles, ideological background, levels of formalism, etc. The SPEM specification is structured as a UML profile, and provides a complete *Meta Object Facility (MOF)*[4] based meta-model [Object Management Group 2002]. UML diagrams facilitate the integration of fragments from different methodologies. SPEM helps describe method fragments which can then be chosen according to specific needs, and, using the UML notation, act as a common language (even though it cannot be defined as a language itself), helping compare fragments. A standard notation is the best way to enable the reuse of methodology fragments.

Furthermore, the choice of FIPA fragments is supported by the following: 1) The other fragment approaches consist of mere guidelines to define fragments, but it is left to the designer to decide how to write a fragment. This may lead to a weakly structured approach. 2) Contrary to other approaches, SPEM leads to homogeneous fragments, which can be used like a language. 3) SPEM is well-suited for agent-based methods.

The work described in this article was initiated in spring 2008. At this time, SPEM version 2.0 was in its preliminary state and not stable yet. This is why the described fragments follow SPEM v1.0.

A fragment, as we use it, is a subset of the following elements:

— A portion of a process, defined by a SPEM diagram.
— Deliverables which permit process reconstruction. They include references to a recommended notion/language/structure to be used for representation.
— Preconditions which represent constraints for the system.

[2]http://www.fipa.org/activities/methodology.html
[3]http://www.omg.org/spec/SPEM/
[4]http://www.omg.org/spec/MOF/2.0/

— A list of concepts (related to the methodology's meta-model) to be defined (designed) or refined during the use of the process fragment.
— Guidelines for the fragment application and related best practice.
— A glossary of terms used.
— Composition guidelines: descriptions of the context/problem that is behind the source methodology.
— Aspects of fragment: textual descriptions of special issues.
— Fragment dependency relationships useful to assemble fragments.

When presenting fragments, we use the following abbreviations: *DL* for deliverables which are the output of a fragments, *PC* for preconditions which are the inputs of a fragment, and *GL* for guidelines which help the user apply the fragment under the correct conditions. The *black dot* stands for the starting point of the considered fragment, and the *circled black dot* stands for end of the fragment. We can see an example of fragment and its notation in Figure 4.

4. CASE STUDIES

To illustrate how the main aspects of the methods presented in sections 5 to 9 apply to concrete examples, we consider the following two case studies.

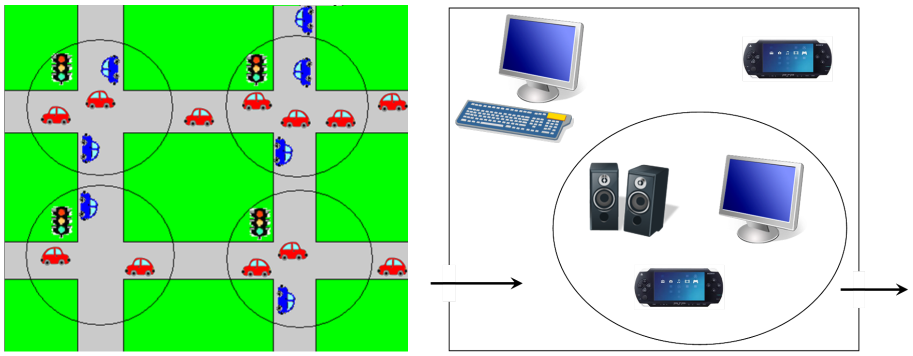
4.1. Traffic lights control

The system is 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 of each other but may communicate with each other, and the global system behaviour will appear from the traffic flow of the whole system. TLs have information on their local neighbourhood, such as the number of cars on the local lanes. We consider traffic lights, disposed in a grid as shown in Figure 2(a). To simplify, cars travel along horizontal and vertical lines only. TLs synchronise with each other using the traffic flow information given by cars that pass by. Although simple, we chose this example as it was one of the first used to illustrate methodologies for self-organising systems [Gershenson 2007].

4.2. Self-organising displays

The system consists of networked displays, where the term *display* includes different types of screens and speakers. The displays are in charge of dynamically and adaptively providing images, sound and information, in a coordinated and context-aware manner. All users have their own displays, which accompany them everywhere, but there are also static displays in the environment. When the user needs to reach a goal (e.g. watch a movie or video-clip, listen to music or send an e-mail), the displays in the local environment can compose their services to best serve the user. Displays can interact with each other and coordinate what to disseminate and how to disseminate it, as they can exist individually or in aggregation with others.

A **take-over scenario** is illustrated in Figure 2(b): a user enters a room while watching a movie on its personal play station portable (PSP). In the room, there are displays of various types. As the user enters, the PSP starts communicating with the available displays and eventually finds partners for composition: an LCD screen may show the movie, while the hifi-speakers provide enhanced sound quality. The PSP remains available with its command function, for the user to control the playing of the movie. Once the user has left the room, the composition dissolves and the PSP returns to being the only device playing the movie.



(a) Traffic lights case study (b) Self-organising displays case study

Fig. 2. Case studies

For this case study we make the following assumptions:

— Every user carries computing power in the form of a PSP or similar;
— Every PSP and every display is connected to a wireless communication network.

Possible display states are:

— disconnected from the network (no power supply, no network connection);
— connected to the network and:
  — available for composition (if compatible and idle / alone);
  — unavailable for composition (incompatible or occupied);
  — failing.

We consider the following cases: **Self-adaptation:** For adaptive interaction with the user, displays have to spontaneously react to the specific profile of users nearby, without human configuration. **Self-management:** Displays in the environment can be added, removed, changed or updated. Some of them may fail. **Interoperability and composability:** The system has to compose itself considering the interoperability of the accessible devices. A composite device must be able to act as a single logical display while being composed of several ones.

The following sections detail the considered software engineering methods and how they can be exploited to address the presented case studies.

5. ADELFE

Adelfe [Bernon et al. 2005] 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) [Odell et al. 2001], and proposes a design process based on the RUP (Rational Unified Process). Adelfe is based on the AMAS (Adaptive Multi-Agent System) theory [Picard and Gleizes 2004] where cooperation is fundamental.

During cooperation an agent tries: to anticipate problems; to detect cooperation failures, called *Non Cooperative Situations (NCS)*; and to recover from NCS [Capera et al. 2004]. 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 remove 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 NCSs 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* (WD1), *Final Requirements* (WD2), *Analysis* (WD3), *Design* (WD4), *Implementation* (WD5) and *Test* (WD6) (see Figure 3). Each phase consists of several activities (A), and each activity consists of several steps (S). We focus on the Adelfe architecture directly related to self-organisation issues, fur further details on the other activities see [Bernon et al. 2005].

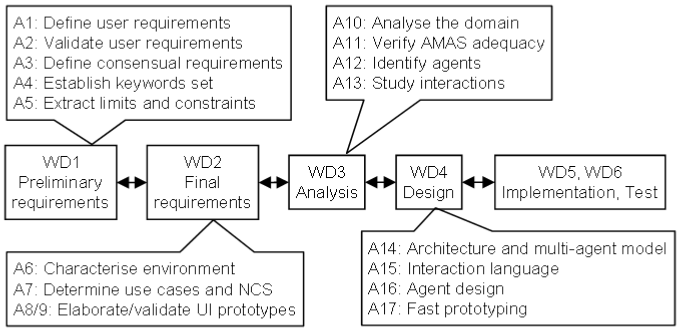

A1: Define user requirements
A2: Validate user requirements
A3: Define consensual requirements
A4: Establish keywords set
A5: Extract limits and constraints
A10: Analyse the domain
A11: Verify AMAS adequacy
A12: Identify agents
A13: Study interactions
WD1 Preliminary requirements
WD2 Final requirements
WD3 Analysis
WD4 Design
WD5, WD6 Implementation, Test
A6: Characterise environment
A7: Determine use cases and NCS
A8/9: Elaborate/validate UI prototypes
A14: Architecture and multi-agent model
A15: Interaction language
A16: Agent design
A17: Fast prototyping

Fig. 3. The ADELFE methodology [Picard and Gleizes 2004]

In **WD2**, 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 (S2). In **WD3**, an interactive tool helps decide if the use of the AMAS theory is suitable or not for the considered system (A11), answering to some global level and local level questions (examples of questions include: Is the global task incompletely specified? Is an algorithm a priori unknown?; If several entities are required to solve the global task, do they need to act in a certain order?; Can the behaviour of an entity evolve? Does it need to adapt to the changes of its environment?; etc.). In A12 agents of the system are identified starting from the analysis of the entities present in the system itself. In A13, entity relationships useful for cooperation are defined, especially agents relationships (S3). In **WD4**, protocol diagrams serve to study agent interactions (A15). Adelfe provides a model to design cooperative agents (A16). A set of generic NCSs 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 [Bernon et al. 2005].

For each investigated software engineering method, we list the self-organising features that characterise the method and for which we have developed SPEM fragments.

Characteristic self-organisation features:

(1) Environment
(2) Cooperation failures
(3) AMAS adequacy
(4) Agent
(5) Interactions between entities
(6) NCS

5.1. Fragments

The above-mentioned features are now expressed as method fragments which describe the most important steps for self-organising systems according to ADELFE. In each fragment one of the feature is analysed and created (e.g. NCS are identified in the fragment called “Agent Specification Fragment”).

(1) Environmental Description Fragment (A6 – Figure 4a). **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 characterise the environment.

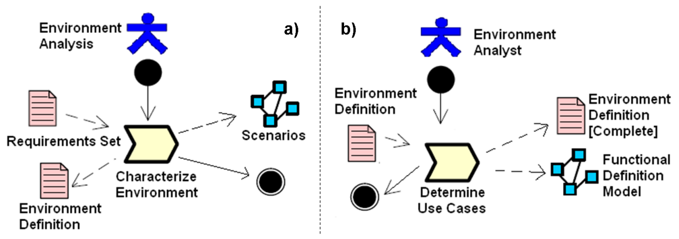


Fig. 4. (a) Environment Description Fragment, (b) Use Cases Description Fragment

(2) Use Cases Description Fragments (A7 – Figure 4b). **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.

(3) Adequacy Verification Fragment (A11 – Figure 5a). **DL:** the final AMAS adequacy synthesis document. **PC:** the preliminary software architecture document, described in the Adelfe domain description fragment (see [Capera et al. 2004] for details). **GL:** verify the AMAS adequacy at local and global level.

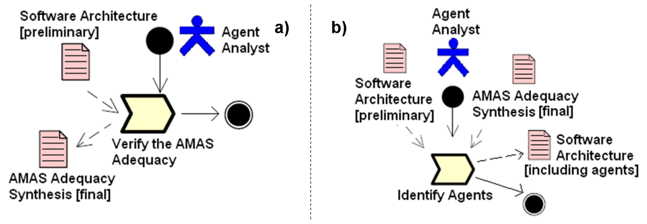

Software Architecture [preliminary]
Agent Analyst
a)
Verify the AMAS Adequacy
AMAS Adequacy Synthesis [final]
b)
Agent Analyst
Software Architecture [preliminary]
AMAS Adequacy Synthesis [final]
Software Architecture [including agents]
Identify Agents

Fig. 5. (a) Adequacy Verification Fragment, (b) Agent Identification Fragment

(4) Agent Identification Fragment (A12 – Figure 5b). **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.

(5) Interaction Between Entities Identification Fragment (A13). Important for agent relationships, but also other fragments can be used; see [Capera et al. 2004]. textbfDL: the relationships found will be used to update the final environment definition document, the software architecture and the internal interaction between domain classes UML diagram. **PC:** the software architecture document (including agents), and the complete environment definition document. **GL:** consider interactions of of three types:

active-passive entities relationships, active entities relationships, and agents relationships.

(6) Agent Specification Fragment (A15, A16 – Figure 6). **DL:** the AUML protocol diagrams, which specify the interaction language, the final interaction language document and the detailed architecture document, including the agent model, skills, aptitudes, world representation and NCSs. **PC:** the initial detailed architecture document defined in the Adelfe architecture definition fragment. **GL:** define and test agent behaviours.

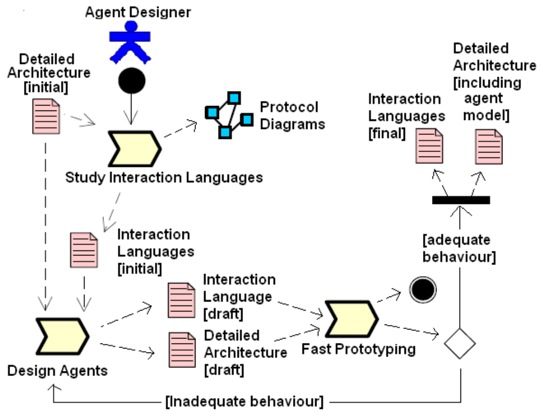

Agent Designer
Detailed Architecture [initial]
Protocol Diagrams
Study Interaction Languages
Interaction Languages [initial]
Interaction Language [draft]
Detailed Architecture [draft]
Design Agents
Fast Prototyping
Interaction Languages [final]
Detailed Architecture [including agent model]
[adequate behaviour]
[Inadequate behaviour]

Fig. 6. Agent Specification Fragment

5.2. Case study 1: traffic lights control

We used the Adelfe Toolkit[5] to produce the documents and diagrams which are related to the self-organising part of the methodology, and to simulate the created system in the end. For further activities see [Puviani et al. 2009].

(1) Environment **A6:** The environment definition document here created states that TL are active entities and cars are passive entities because they are simple resources. Then the system is defined as accessible, non-deterministic, dynamic, and continuous.

(2) Cooperative failures **A7-S2:** Failures can occur when the traffic flow is equal in both directions (horizontal flow and vertical flow), or when the number of cars in a directions exceeds a chosen threshold.

(3) AMAS adequacy **A11:** the AMAS adequacy is verified by answering the AMAS questions.

Global level:

(1) *Q*: Is the global task incompletely specified? Is an algorithm a priori unknown? *A*: The global task is not completely specified and *a priori* algorithm is unknown.
(2) *Q*: If several entities are required to solve the global task, do they need to act in a certain order? *A*: Components (TL and cars) are not subject to a fix order of action.
(3) *Q*: Is the solution generally obtained by repetitive tests, are different attempts required before finding a solution? *A*: Several attempts are required to find a solution.
(4) *Q*: Can the system environment evolve? Is it dynamic? *A*: The system environment is dynamic because cars can enter and exit the system.

[5]http://www.irit.fr/ADELFE/Download.html

Table I. *Equal flow* NCS and *Too many cars* NCS

Name	**Equal flow**
State	Any
Description	The horizontal traffic flow is equal to the vertical traffic flow
Condition(s)	$flow_{horizontal} == flow_{vertical}$
Action(s)	The TL decides which flow to consider first.

Name	**Too many cars**
State	Any
Description	The number of cars exceeds the chosen threshold θ.
Condition(s)	$flow_{horizontal} > \theta$ or $flow_{vertical} > \theta$
Action(s)	Prevent cars from entering, let them leave quickly.

(5) *Q*: Is the system process functionally or physically distributed? Are several physically distributed entities needed to solve the global task? Or is a conceptual distribution needed? *A*: The system is physically distributed.
(6) *Q*: Is a great number of entities needed? *A*: The system can consist of a great number of components.
(7) *Q*: Is the studied system non-linear? *A*: The system is non-linear because TLs are autonomous and have only local information.
(8) *Q*: Is the system evolutionary or open? Can new entities appear or disappear dynamically? *A*: Some new components (cars) can appear and disappear dynamically.

Local level:

(9) *Q*: Does an entity only have a limited rationality? *A*: A component has limited rationality.
(10) *Q*: Is an entity "big" or not? Is it able to do many actions, to reason a lot? Does it need great abilities to perform its own task? *A*: It (TL) is able to perform several actions.
(11) *Q*: Can the behaviour of an entity evolve? Does it need to adapt to the changes of its environment? *A*: It needs to adapt to the changes in its environment.

(4) Agent **A12:** TLs are agents because they: (a) are autonomous, (b) have a local goal: optimize the traffic flow, (c) interact with other TLs exchanging flow information, and (d) have a partial view of the environment: their state, the traffic flow in their local area. Instead cars are non autonomous entities and cannot be viewed as agents; they follow the TLs and progress at a given speed.

(5) Interactions **A13** and **A15:** defining the relationship between agents and their communication protocol. Consider a situation where Y is a TL, and X is a TL as well, and they are in the same direction (same horizontal or vertical line). They have to communicate to exchange their traffic flow information; with this information they can change their status.

(6) NCS **A16:** design cooperative agents. For TL defines skills (e.g. *ManageConstraints*, *ManageFlowInformtion* and *ManageLight*), aptitudes (e.g. *ChangeLightColour* and *SendMessage*), interaction language by messages and NCS; see Table I.

5.3. Case study 2: self-organising displays

(1) Environment **A6:** The environment definition document states that both users and displays are active entities. Then the system is defined as accessible, non-deterministic, dynamic, and continuous.

(2) Cooperation failures **A7-S2:** Failures can occur when trying to connect to a disconnected display; if a display attempts to connect to an equivalent display (e.g. a touch screen can try to connect to another touch screen, which does supposedly not make much sense). Summarised in Table II.

Table II. Connection to a disconnected or equivalent display NCS

Name	**Connection to a disconnected display**
State	Any
Description	Display X tries to connect to display Y that is not connected
Condition(s)	X tries connection with Y AND status(Y) == not connected
Action(s)	X verifies the status of Y, then connects to another display

Name	**Connection to an equivalent display**
State	Any
Description	Display X tries to connect to display Y that has a functionality equivalent to X
Condition(s)	X tries connection with Y AND Function(Y) $\equiv$ Function(X)
Action(s)	X verifies functionality of Y, then connects to another display

(3) AMAS adequacy **A11:** verified by answering the AMAS questions in the following way (for questions see Subsection 5.2):

Global level:

(1) *A*: The global task is not completely specified and *a priori* algorithm is unknown.
(2) *A*: Components are not subject to a fix order of action.
(3) *A*: Several attempts are required to find a solution.
(4) *A*: The system environment is dynamic.
(5) *A*: The system is physically distributed.
(6) *A*: The system can consist of a great number of components.
(7) *A*: The system is non-linear because the displays are autonomous and act locally.
(8) *A*: Some new components (displays) can appear and disappear dynamically.

Local level:

(9) *A*: A component has limited rationality.
(10) *A*: It is able to perform several actions.
(11) *A*: It needs to adapt to the changes in its environment.

(4) Agents **A12:** Human users are agents because they: (1) are autonomous, (2) have a local goal: getting their service request satisfied, (3) interact with displays, and (4) have a partial view of the environment: their position, the state of their own displays, and the state of displays near their position, when they interact with them. Also Displays are agents because they: (1) are autonomous, (2) have a local goal: optimising the given function by collaborating with other displays, (3) interact with other displays, (4) have a partial view of the environment: their position, their state, and the states of displays which they are able to connect to, and (5) can negotiate with other displays in order to collaborate with them.

(5) Interactions **A13** and **A15:** defining the relationship between agents and their communication protocol. Consider a situation where display Y is a screen, and user display X is a screen as well. As they cannot compose their services, the connection between them must be cancelled.

(6) NCS **A16:** design cooperative agents. For user agent defines skills (e.g. *ManageConstraints*, *ManagePosition* and *ManageOwnDisplay*), aptitudes (e.g. *SendListOfDisplay*). For display agents define skills (e.g. *ManageStatus*, *ManageConstraints* and *ManagePossibleConnection*), aptitudes (e.g. *AskForConnection* and *ManageOptimisedConnection*). Defines the interaction language (by messages) and NCS (see Table II).

6. THE CUSTOMISED UNIFIED PROCESS

The *Customised Unified Process (CUP)* [De Wolf 2007] 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)* [Jacobson et al.

1999], and is customised to explicitly focus on engineering macroscopic behaviour of self-organising systems (see Figure 7).

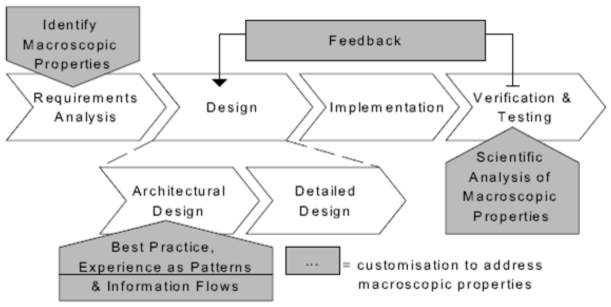

Identify Macroscopic Properties
Feedback
Requirements Analysis
Design
Implementation
Verification & Testing
Scientific Analysis of Macroscopic Properties
Architectural Design
Detailed Design
Best Practice, Experience as Patterns & Information Flows
... = customisation to address macroscopic properties

Fig. 7. Customised Unified Process Methodology [De Wolf 2007]

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* (a 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' [De Wolf 2007]. Activity diagrams are used to determine when a certain behaviour starts and what its inputs are. Information flows are enabled by decentralised coordination mechanisms, defined by provided design patterns. 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 (agent behaviour). 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 [De Wolf 2007].

Characteristic self-organisation features:

(1) Locality
(2) Information flow

6.1. Fragments

We identified the following two main fragments for self-organisation in CUP:

(1) Locality Identification Fragment: (Figure 8a). **DL:** a UML diagram (e.g. an 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 8b). **DL:** a 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 into sub-goals, then determine the information flow.

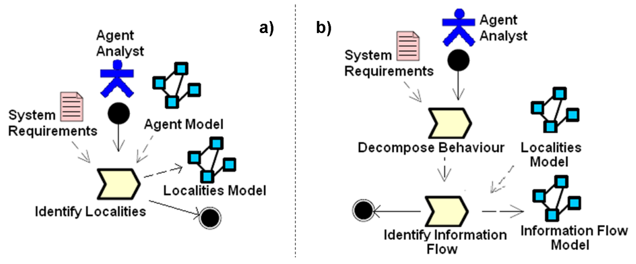


Fig. 8. (a) Locality Identification Fragment, (b) Information Flow Definition Fragment

We did not create specific fragments for the patterns of decentralised coordination mechanisms as the pattern list is only useful for the developer to choose the communication and coordination mechanism. It is therefore not possible to define a real fragment. It is, however, important to consider this list while defining a self-organising system. The list can be integrated into other fragments while building the system.

6.2. Case study 1: traffic light control

We do not describe the *Requirement Analysis phase*, but we only define the main structure of the system: TLs are defined as agents; and cars are considered as entities.

(1) Localities: For the *Design phase*, the circles around the TLs define the localities (see Figure 2(a)) which are represented by each TL within the information of its states and of the traffic flow in the surrounding area.

(2) Information flow: The main goal of the system (flow optimization) can be decomposed into sub-goals, which consist of flow optimisation for each TL. The information needed for each TL to act in the system are: the status of its light, the horizontal and vertical flow in its area. We chose *gradient fields* as patterns of decentralised coordination mechanism [Mamei et al. 2004]: spatial, contextual, and coordination information is automatically and instantaneously propagated by the environment as computational fields. Agents simply follow the 'waveform' of these fields to achieve the coordination task. So here the traffic flow automatically propagates information between TLs.

6.3. Case study 2: self-organising displays

Displays as well as users are agents. There are no passive entities.

(1) Localities: The circle around the user and the next displays in Figure 2(b) illustrates the relevant user locality; display localities are not shown, but they consist of an area around the display and all the entities within. The sub-goals in which the main one can be divided are connecting displays and verifying the quality of service of the new devices (created by the composition of displays).

(2) Information flow: The information needed for each user is the user's position and the list of displays in the respective locality. For each display the information needed is its state (connected or not) and the possibility of connection. Each displays attempts to connect to peers, until the composite display being formed can fulfil the requested function F. Optimising F means matching the user quality of service. Again, we chose the gradient fields explained in the previous sub-section as patterns of decentralised coordination mechanism. The communication is automatically propagated by display connections.

7. METASELF

MetaSelf proposes both a *system architecture* and a *development process* [Di Marzo Serugendo et al. 2010]. It addresses the development of dependable self-* systems (both self-organising and self-managing). *MetaSelf* considers a self-organising system as a collection of loosely coupled autonomous components (agents or services). Metadata describes the components' functional and non-functional characteristics, such as availability levels, and environment-related metadata (e.g. artificial pheromones). The system's behaviour (self-organisation and self-management, e.g. reconfiguration to compensate for component failure) is governed by rules for self-organisation and policies for dependability and self-management that 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 rules and policies.

The MetaSelf system architecture involves autonomous components, repositories of metadata, rules for self-organisation and policies for dependability and self-adaptation, 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 rules and policies at run-time (Figure 9). *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.

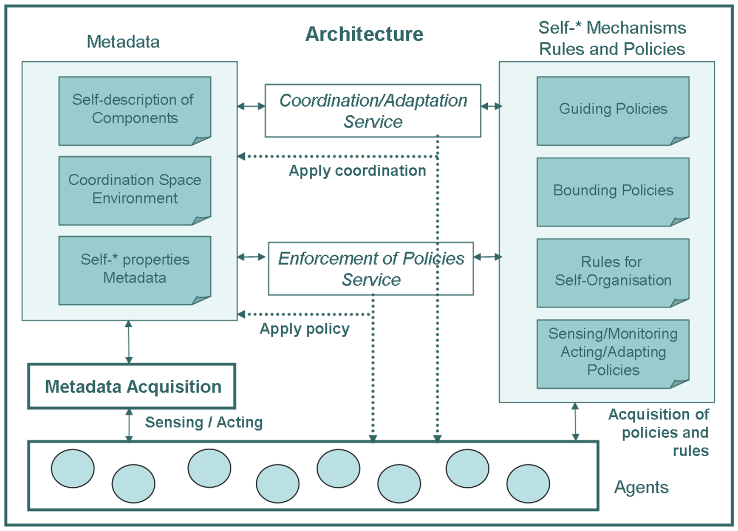

Architecture
Metadata
Self-* Mechanisms Rules and Policies
Self-description of Components
Coordination Space Environment
Self-* properties Metadata
Coordination/Adaptation Service
Apply coordination
Enforcement of Policies Service
Apply policy
Guiding Policies
Bounding Policies
Rules for Self-Organisation
Sensing/Monitoring Acting/Adapting Policies
Metadata Acquisition
Sensing / Acting
Acquisition of policies and rules
Agents

Fig. 9. MetaSelf Run-time Generic infrastructure [Di Marzo Serugendo et al. 2010]

The MetaSelf development process consists of 4 phases (Figure 10).

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 required quality of service is determined.

The **Design phase** consists of two sub-phases: **D1** - the designer chooses architectural patterns (e.g. autonomic manager or observer/controller architecture) and self-* mechanisms (governing the interactions and behaviour of autonomous components (e.g. trust, gossip, or stigmergy). Generic rules for self-organisation and dependability policies are defined. In the second part; **D2** - the individual autonomous components (services, agents, etc.) are designed. The necessary metadata, rules and policies are selected and described. The self-* mechanisms are simulated and adapted or improved.

The **Implementation phase** produces the run-time infrastructure (Figure 9) including agents, metadata and executable policies.

In the **Verification phase**, the designer makes sure that agents, the environment, artefacts and mechanisms work as desired. Potential faults arising on one of these design elements and their consequences are identified, similar to the way *failure modes and effects analysis (FMEA) [Leveson 1995]* works. Corrective measures (redesign or dependability policies) to avoid, prevent or remove the identified faults are taken accordingly [Di Marzo Serugendo 2009].

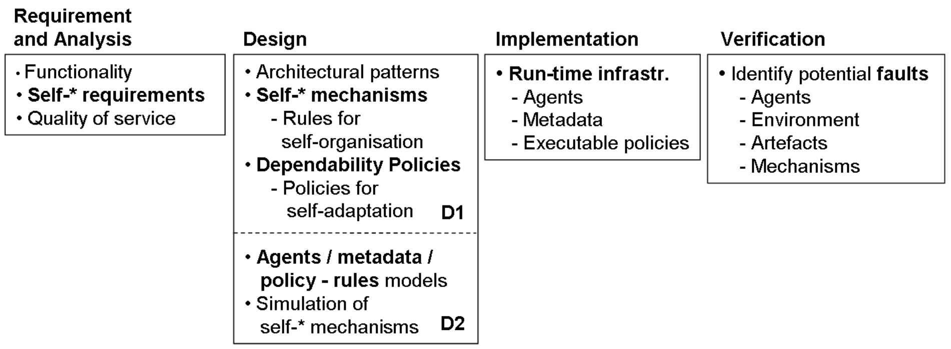


Fig. 10. MetaSelf Development Process

The MetaSelf development process has been applied to dynamically resilient Web services [Di Marzo Serugendo et al. 2008] and to self-organising industrial assembly systems [Frei et al. 2008].

Characteristic self-organisation features:

(1) Architectural patterns
(2) Self-* mechanism
(3) Run-time infrastructure

7.1. Fragments

The most relevant MetaSelf fragments are:

(1) Architectural Patterns Fragment: (Figure 11a). **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.

(2) Identification of Software Architecture Fragment: (Figure 11b). **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. **GL:** define design phase entities.

(3) Run Time Infrastructure Definition Fragment: (Figure 12). **DL:** the final version of the agent model, the executable policies, the metadata repository, and adaptation / coordination services repository. **PC:** the adaptation and coordination mechanism

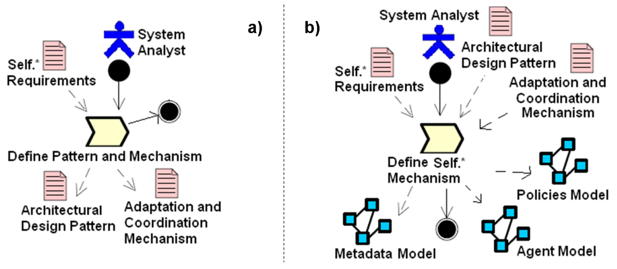

a)
System Analyst
Self.* Requirements
Define Pattern and Mechanism
Architectural Design Pattern
Adaptation and Coordination Mechanism
b)
System Analyst
Architectural Design Pattern
Self.* Requirements
Adaptation and Coordination Mechanism
Define Self.* Mechanism
Policies Model
Metadata Model
Agent Model

Fig. 11. (a) Architectural Patterns Fragment, (b) Identification of Software Architecture Fragment

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.

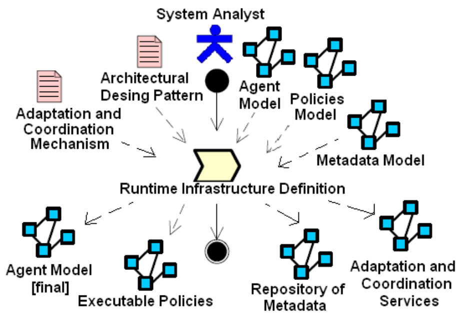

System Analyst
Architectural Desing Pattern
Adaptation and Coordination Mechanism
Agent Model
Policies Model
Metadata Model
Runtime Infrastructure Definition
Agent Model [final]
Executable Policies
Repository of Metadata
Adaptation and Coordination Services

Fig. 12. Run Time Infrastructure Definition Fragment

7.2. Case study 1: traffic lights control

We determine the following **components:** TLs are active (agents), and cars are passive entities (artefacts that change their state only due to environmental change).

(1) Architectural Patterns: **Self-* requirements:** Traffic flow is optimised as a result of self-organisation. **Architectural Patterns:** The generic observer/controller architecture is chosen [Schoeler and Mueller-Schloer 2005]. Each TL comes with its observer and controller components.

(2) Self-* mechanism: **Self-organisation 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.

(3) Run-time infrastructure: **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.

7.3. Case study 2: self-organising displays

We identify the **components**: users and displays are agents.

(1) Architectural Patterns: **Self-* requirements:** Every time a user moves in the environment, the self-organising process creates a new configuration of displays. This includes self-selection of displays and self-composition to form a composite device.

(2) Self-* mechanism: **Self-organisation mechanism:** The mechanism chosen for letting the different displays self-organise when a user arrives at a new location is self-assembly. Components progressively attach to each other following matching rules and according to the current configuration of the structure to build. **Coordination mechanism:** Self-assembled displays coordinate their tasks, e.g. audio and video synchronisation.

(3) Run-time infrastructure: **Guiding Policies:** User agents request service functions F. Display agents need to fulfil and optimise F; they need to find compatible displays for collaboration; and they compose when other displays are: in range, compatible, available and selected. **Display coordination policies:** Coordination of tasks: audio and visual information need to be synchronised. **Bounding policies:** A display connects to local displays only. **Sensing/Monitoring policies:** If a display changes its status, it has to upload this status to remain connected. If a display is near the user location and can help fulfil the requested function, it adds itself to the user's list. A display with a low power capability searches for an equivalent display to replace itself. **Self-description metadata:** For each user agent: position and the list of personal displays. For each display: its position, its status (connected or not), and possibilities of composition with other displays.

8. A GENERAL METHODOLOGY

The *General Methodology* [Gershenson 2007] 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 13.

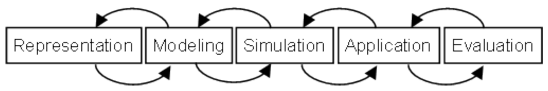


Fig. 13. Methodology steps, adapted from [Gershenson 2007]

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. The representation of the system should consider different level of abstractions.

In the **Modeling phase**, a control mechanism is defined, which should be internal and distributed to 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-organising 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 maximise 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 synchronise the agents to minimise waiting times.

In the **Simulation phase**, the developed model(s) are implemented and different scenarios and mediator strategies are tested. Simulation development proceeds in stages: from abstract to particular. The models are progressively simulated, and based on the results, the models are refined and simulated again. 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 previous ones.

This methodology was applied to traffic lights, self-organising bureaucracies and self-organising artefacts [Gershenson 2007].

Characteristic self-organisation features:

(1) Control mechanism and friction reduction

8.1. Fragments

We extracted the following fragments:

(1) Control Mechanism Definition Fragment (Figure 14). Creates a communication model based on how to optimise the system. **DL:** a 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.

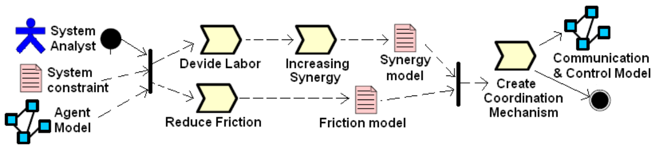


Fig. 14. Control Mechanism Definition Fragment

8.2. Case study 1: traffic lights control

Define **requirements:** Develop a feasible and efficient traffic light control system.

(1) Control mechanism and friction reduction: **Representation phase:** The system can be modelled on two levels: *car level* and *TL level*. The cars' goal is to maximise their satisfaction σ (travelling freely and without stopping at intersections). $\sigma = 0$ corresponds to a car stopping indefinitely. The TL system's goal is to maximise the system's satisfaction σ_{system} (all cars travel as fast as possible, and are able to flow through the city without stopping). $\sigma_{system} = 0$ corresponds to a traffic jam where all cars stop indefinitely. **Modeling phase (1):** Find a mechanism to coordinate TLs so that these mediate between cars, to reduce their friction (try to prevent them from arriving at the same time at the crossing). This will maximise the satisfactions of the cars and the TLs. As all vehicles contribute equally to σ_{system}, frictions are minimised through compromise. **Modeling phase (2):** Each TL keeps a count (κ_i) of the number of cars time steps ($c{*}ts$) approaching *only* the red light, from a distance ρ. κ_i can be seen as the integral of waiting/approaching cars over time. When it reaches a threshold θ, 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. **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 ψ_{min}. Once $\psi_i \geq \psi_{min}$, the lights will change when $\kappa_i \geq \theta$.

For the complete result of possible simulations related to these and other traffic light control models with more constraints, see [Gershenson 2007].

8.3. Case study 2: self-organising displays

Requirements: Optimising a function F representing a user requested service.

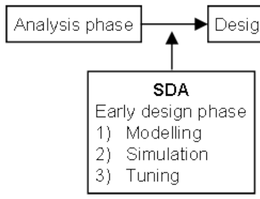
(1) Control mechanism and friction reduction: **Representation phase:** The system can be modelled at the level of the user and the displays. The user's goal is to obtain a service: the satisfaction σ is maximal when the function F is satisfied; the device is accessible. $\sigma = 0$ corresponds to a situation with no available displays. The displays' goal is to collaborate with each other in order to optimise a given function F. The system satisfaction σ_{system} is maximal when the quality of service requested by the user is reached. **Modeling phase:** Find a mechanism which can coordinate the displays by reducing their friction (try to prevent them from disconnecting one from the others). Taking the function F and the list L of active user and environment devices, each device creates a list L_{temp} with all the devices that can be used to satisfy F. Knowing how many displays are necessary to fulfil F, for each display $D_i \in L_{temp}$ the system has to find possible compositions. The best composition found will be kept in a temporary variable f. Friction minimisation happens through compromise.

9. A SIMULATION DRIVEN APPROACH

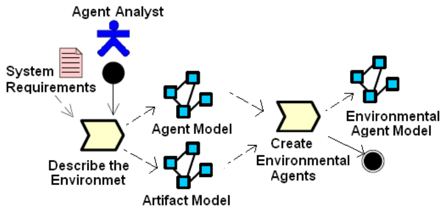
The *Simulation Driven Approach (SDA)* [Gardelli et al. 2008] to build self-organising systems is not a complete methodology, but rather a way of integrating a middle phase into existing methodologies. To describe the environment, suitable abstractions for environmental entities are necessary: the *Agent & Artefact metamodel* [Ricci et al. 2005] considers agents as autonomous and proactive entities driven by their internal goal/task. Artefacts are passive, reactive entities providing services and functionalities to be exploited by agents through a user interface. To overcome many methodologies' limitations regarding the environment, *environmental agents* are introduced. They are responsible for managing artefacts to achieve the targeted self-* properties. Environmental agents are different form standard agents (*user agents*), which 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 15(a)). It assumes that system requirements have just been collected and the analysis has been performed, identifying the services to be provided by environmental agents. To design environmental agents, a model of agents and environmental resources is needed. 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 **Modeling phase**, strategies are formulated to make the system behaviours explicit. To enable further automatic elaborations and reduce ambiguity, these descriptions should be provided in a formal language (not specified in [Gardelli et al. 2008]). 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. In the **Tuning phase**, the model has to be *tuned* until the desired qualitative dynamics is reached, which depends on initial conditions. The tuning process may provide unrealistic parameter values, or may not reach the required behaviour. This means that the chosen



(a) Design phases. Adapted from [Gardelli et al. 2008].



(b) Describe the Environment Fragment

Fig. 15.

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 and to plain diffusion [Gardelli et al. 2008].

Characteristic self-organisation features:

(1) Environmental agents and artefacts

9.1. Fragments

(1) Describe the Environment Fragment (Figure 15(b)). It is relevant because 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. **PC:** the system requirements. **GL:** first describe the environment by extracting agent and artefacts, then create the environmental agents which manage artefacts to achieve the system's self-* properties.

9.2. Case study 1: traffic lights control

This approach is very similar to the one described in section 8.2. They are both based on modelling, simulation and testing (here called tuning). In the follow we only detail the modelling phase; the simulation phase is up to the developer experiences with simulation tools.

(1) Environmental agents and artefacts: We consider four environmental agents (TL). Each TL performs a partial system observation of the space: traffic flow and cars (near and far). The light itself can be considered as an artefact, managed by the TL to achieve the target self-* properties. Each agent has the goal of optimising the traffic flow. Cars are also considered as artefacts: passive entities that provide a service (the traffic flow) and mediate TL interactions (by providing traffic flow information).

The TL calculates the number of cars C that are present (in both directions) in its neighbourhoods. This information is useful to calculate the local traffic flow, and to decide which light to turn green or red. Moving cars modify the traffic flow and thus influence the TL switching. Passing from one TL to another, cars transmit traffic flow information between TLs.

9.3. Case study 2: self-organising displays

(1) Environmental agents and artefacts: Users are considered as environmental agents. Each agent manages its device information to achieve the global goal: optimising a service requested by the user. Displays are considered as artefacts, with their

properties, such as position and type of device. Each user has a list L of active devices in the surroundings (with position and type). When the user changes her position, the agent updates L. Then, starting from the user's list of artefacts, the devices try to compose themselves in order to optimise the service requested by the user.

10. METHODOLOGIES COMPARISONS

After studying methodologies for creating self-organising systems and their important fragments, we shed light on advantages and weaknesses of these methodologies. Figure 17 (in the electronic appendix) shows the main characteristics of each methodology as identified by the fragments (a *tick* indicates the presence of the specified phase). In this way it can be easier to understand what a developer may need for his/her purpose, to improve his/her methodology for self-organising systems.

10.1. Adelfe

Adelfe is the only methodology that focuses on specific self-organisation characteristics. However, 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 without any kind of scientific test. It is for instance difficult to determine if the system is linear or not. It is also difficult to find and select the essential NCS. Adelfe forces the use of cooperation between the agents, which may be beneficial for many systems, but not for all. In this methodology there are several supporting tools that will help developers build their system.

10.2. The Customised 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. CUP also provides a formal technique for the evaluation of 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 directly given.

10.3. MetaSelf

MetaSelf focuses on self-organising mechanisms and dependability support, which provide respectively low-level and high-level control. MetaSelf relies on loosely coupled components, metadata and executable dynamically changing policies. It only provides a guidance for modeling agents, environments, artefacts, self-organising mechanisms, and for identifying dependability issues. There is no development or verification tool.

10.4. A General Methodology

In this methodology the designer receives assistance for understanding how to develop a system, but none for actual development. The system model needs to be chosen, and simulation attempts realised. No guidance for simulation program choice is given. 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 (left to the user).

10.5. A Simulation Driven Approach

This approach considers the environment, agents and artefacts as first class entities. It is not a complete methodology *per se*, it must be combined with other methodologies, which do not necessarily support elements like environmental agents and artefacts.

11. AUGMENTING A METHOD OR METHOD PROCESS

The above proposed methodologies, except for Adelfe, are rather like guidelines for the developers of self-organising systems. This is why we extracted the most relevant fragments regarding self-organisation. These fragments help developers who want to create their own method process: they may combine existing fragments to create a new or customised development method or integrate them into existing methodologies.

To this end, we propose to use the *Prioritisation algorithm* [Seidita et al. 2010] to define the fragments' order in a methodology. This algorithm starts from the elements presented in the source-methodology meta-model (i.e. the fragments). It identifies the relations which every element can have in the meta-model and, depending on the number of dependencies, we determine the right position of each fragment. This approach is very useful when we start from scratch and only have a preliminary meta-model to work with. On the other hand, when we have an existing methodology which we want to adapt, we will not use the entire approach but only follow the strategy to integrate a fragment with the chosen methodology. We study how the outputs of the initial fragments can be used as inputs to the new one, and vice versa for the outputs.

It is unfortunately impossible to have fragments for simulation and implementation, as these activities are not 'guidelines' for design phases. Every methodology can have a simulation phase that may be based either on the user's knowledge (free from methodology guidelines) or on a specific tool that will be supported by a specific platform. The same applies to the implementation phase; it is not possible to create a fragment general enough for every kind of implementation because this is based on a specific platform. It would be possible to create a fragment for implementation starting from a specific methodology and for a specific implementation platform; but this is beyond the scope of this article.

We subsequently present, as an example, the integration of some fragments in an existing methodology. We show how to augment the PASSI2 methodology [Cossentino and Seidita 2009] with some important self-organisation fragments for the development of the traffic lights system. PASSI2 is an agent-based method which does not consider self-organisation. SPEM fragments of this methodology are already available.

Firstly, we evaluate the PASSI2 methodology and what is necessary to develop the self-organising system we need. Naturally, not all the self-organising fragments we have analysed in this article are necessary for a specific application. Here, we find that the concepts of *environment*, *locality* and *information flow* do not exist in PASSI2, but would be very useful for this kind of systems. Afterwards, we study PASSI2 fragments and the fragments which define the previously mentioned concepts, which we have extracted from the 'self-organisation methodologies'.

11.1. The PASSI2 methodology

PASSI2 [Cossentino and Seidita 2009] is based on PASSI (Process for Agent Societies Specification and Implementation). It is a step-by-step requirement-to-code methodology, for the design and development of multi-agent societies, using the UML notation. It aims at using standards whenever it is possible: its chosen target environment is the standard Foundation for Intelligent Physical Agents (FIPA) architecture.

PASSI2, like PASSI, is based on a meta-model describing the elements that constitute the system to be designed (agents, tasks, communications, roles) and the relationships among them. PASSI2 has been designed with the aim of developing systems that can be: (i) highly distributed, (ii) subject to a (relatively) low rate of requirements changes, and (iii) openness (external systems and agents that are unknown at design time will interact with the system to be built at runtime).

It is composed of three models or process components, which include different phases or work definitions [Cossentino and Seidita 2009]:

(1) The *System Requirement Model* models the system requirements in terms of agency and purpose. It consists of a functional description of the system: system requirements are defined in term of use case diagrams. There is a separation of responsibility concerns: the agents' responsibilities are represented through role-specific scenarios; the agents' structure is defined in terms of tasks required for accomplishing the agents' functionalities; the agents' capabilities are specified separately.

(2) The *Agent Society Model* models the social interactions and dependencies between agents involved in the solution. It describes domain categories (concepts), agents' communications in terms of the referred ontology, as well as interaction protocol and message content language. It shows distinct roles played by agents, the tasks involved in the roles, communication capabilities and inter-agent dependencies in terms of services. It also depicts the agents' structure and their behaviour at the social level of abstraction.

(3) The *Implementation Model* models the solution architecture in terms of methods, classes, deployment configuration, code and testing directives. There are two levels of abstraction: multi-agent level and single-agent level. The agents' structure and their behaviour are described at the implementation level of abstraction.

11.2. Fragment integration

If PASSI2 was not previously decomposed using SPEM, we would now have to do that, using the *prioritization algorithm*: we could start from the methodology meta-model, then add the new entities and while decomposing PASSI2, directly integrate the new fragments. In our case, because we have all the necessary fragments, we study the process and highlight where to insert the fragments.

Starting from the **System Requirements Model**, we find that the entity *environment* will help define the *domain requirements*, so we can add the *Environment Description fragment* from Adelfe before the PASSI2 *Domain Requirements Description fragment*. We notice that the entity *scenario* that was originally given externally (e.g by the user or system developer) in PASSI2; however, it can now be defined in the *Environment Description fragment* and reused in the *Domain Requirements Description fragment*. Then, the *problem statement* used in PASSI2 can be seen as an input instead of the *requirements set* of Adelfe, because the meaning of the two entities is very similar. Moreover, to complete the process, it is better to take the *environment definition* and the new *scenario* as inputs for the *Domain Description fragment*, because they are more complete and at the same time they give the information previously used in the fragment. To better understand how the fragments integrate, see Figures 18(a) and 18(b) in the electronic appendix which show the entire system requirements model.

Afterwards, we proceed with PASSI2's **Agent Society Model**. Studying this phase, we find that the concept of *locality* is useful to describe *roles* and the *information flow* will help, together with the PASSI2 *communication ontology*, to build the *communication protocol*. Thus we decide to introduce the *Locality Identification fragment* and the *Information Flow Definition fragment* from CUP.

In Figures 19(a) and 19(b) (in the electronic appendix), we see which fragments we add and how. During the composition we notice that the *Task Specification Diagram*, which is developed within PASSI2, represents the same output as the *Decompose behaviour activity* in the CUP fragment. Therefore, we exchange the original fragment (from CUP) with the Task Specification Diagram as we can see in Figure 16.

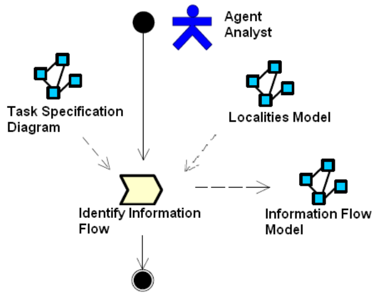


Fig. 16. Information Flow Definition fragment (modified)

Then we introduce the concept of *locality* through the *information flow*, as an input to the *Multi-Agent Behaviour fragment* because it helps define the agents' behaviour in a self-organising system. The last Model will remain the same.

The new methodology is now complete: we have modified PASSI2 to introduce features that permit to develop a self-organising system.

12. CONCLUSIONS

The developers of self-organising systems need the support of customised methodologies. In this article we suggest how designers could extract method fragments and reuse them. To compose customised methodologies, ad hoc fragments, that will underline specific features of the starting methodology (or method), can be used in combination with readily available fragments of different methods or methodologies. These differences need to be considered while building the system and more fragments have to be added or modified, according to the developer's experience and needs. The example proposed in section 11 is intended to clarify how fragments can be added to an existing methodology to develop a self-organising system, starting from an agent-oriented methodology which is not focused on self-organisation.

Composing methodologies is not a simple activity but can be very useful when the existing methodologies do not provide features for specific systems. In this case the use of fragments is sensible and gives promising results. Fragments can be also created from scratch if we know how to work up a specific feature we need in our process; but this is a very difficult work (if we do not have a methodology or method at the base), and it is not the aim of this article to discuss it. We regret that to be flexible enough for fragments reuse, it is not possible to create quite general fragments for simulation and implementation. An increasing number of researchers, particularly in self-organising systems, nowadays consider simulation as an integral part of a methodology.

REFERENCES

BERNON, C., CAMPS, V., GLEIZES, M.-P., AND PICARD, G. 2005. Engineering adaptive multi-agent systems: The ADELFE methodology. In *Agent-oriented Methodologies,*, B. Henderson-Sellers and P. Giorgini, Eds. Idea Group Pub., Hershey, PA, USA, 172–202.

BONABEAU, E., DORIGO, M., AND THÉRAULAZ, G. 1999. *Swarm intelligence*. Oxford University Press, New York, USA.

BRINKKEMPER, S., SAEKI, M., AND HARMSEN, F. 1998. Assembly techniques for method engineering. *Lecture Notes in Computer Science 1413*, 381–400.

CAMAZINE, S., DENEUBOURG, J.-L., FRANKS, N., SNEYD, J., THERAULAZ, G., AND BONABEAU, E. 2001. *Self-organization in biological systems*. Princeton University Press, Princeton, NJ, USA.

CAPERA, D., PICARD, G., AND GLEIZES, M.-P. 2004. Applying ADELFE methodology to a mechanism design problem. In *Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*. Vol. 3. New York, USA, 1508–1509.

COSSENTINO, M. AND SEIDITA, V. 2009. Passi2-going towards maturity of the passi process. Tech. rep., ICAR-CNR Technical Report.

DE WOLF, T. 2007. Analysing and engineering self-organising emergent applications. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

DE WOLF, T. AND HOLVOET, T. 2005. Emergence versus self-organisation: Different concepts but promising when combined. In *Engineering Self-Organising Systems*, S. A. Brueckner, G. Di Marzo Serugendo, A. Karageorgos, and R. Nagpal, Eds. LNAI Series, vol. 3464. Springer, Berlin Heidelberg, 1–15.

DI MARZO SERUGENDO, G. 2009. Robustness and dependability of self-organising systems - a safety engineering perspective. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems(SSS)*. LNCS Series, vol. 5873. Springer, Berlin Heidelberg, Lyon, France, 254–268.

DI MARZO SERUGENDO, G., FITZGERALD, J., AND ROMANOVSKY, A. 2010. MetaSelf - an architecture and development method for dependable self-* systems. In *Symp. on Applied Computing (SAC)*. Sion, Switzerland, 457–461.

DI MARZO SERUGENDO, G., FITZGERALD, J., ROMANOVSKY, A., AND GUELFI, N. 2008. MetaSelf - a framework for designing and controlling self-adaptive and self-organising systems. Tech. rep., BBKCS-08-08, School of Computer Science and Information Systems, Birkbeck College, London, UK.

DI MARZO SERUGENDO, G., GLEIZES, M., AND KARAGEORGOS, A. 2005. Self-organization in multi-agent systems. *Knowledge Engineering Review 20,* 2, 165–189.

FIRESMITH, D. AND HENDERSON-SELLERS, B. 2002. *The OPEN Process Framework*. Addison-Wesley, Harlow, UK.

FREI, R., DI MARZO SERUGENDO, G., AND BARATA, J. 2008. Designing self-organization for evolvable assembly systems. In *IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO)*. Venice, Italy, 97–106.

GARDELLI, L., VIROLI, M., CASADEI, M., AND OMICINI, A. 2008. Designing self-organising environments with agents and artifacts: A simulation-driven approach. *Int. Journal of Agent-Oriented Software Engineering 2,* 2, 171–195.

GERSHENSON, C. 2007. Design and control of self-organizing systems. Ph.D. thesis, Faculty of Science and Center Leo Apostel for Interdisciplinary Studies, Vrije Universiteit, Brussels, Belgium.

HEYLIGHEN, F. 1999. Self-organisation. In *Principia cybernetica web*, F. Heylighen, C. Joslyn, and V. Turchin, Eds. Brussels.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The unified software development process*. Addison Wesley, Reading, MA, USA.

LEVESON, N. G. 1995. *Safeware: System Safety and Computers*. Addison Wesley, Reading, MA, USA.

MAMEI, M., ZAMBONELLI, F., AND LEONARDI, L. 2004. Cofields: a physically inspired approach to motion coordination. *IEEE Pervasive Computing 3,* 2, 52–61.

OBJECT MANAGEMENT GROUP, O. 2002. Meta object facility (MOF), specification.

ODELL, J., PARUNAK, H., AND BAUER, B. 2001. Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, Berlin, 121–140.

PICARD, G. AND GLEIZES, M.-P. 2004. The ADELFE methodology-designing adaptive cooperative multi-agent systems. In *Methodologies and Software Engineering for Agent Systems*, F. Bergenti, M.-P. Gleizes, and F. Zambonelli, Eds. Kluwer Academic Publishers, Norwell, MA, USA, 157–175.

PUVIANI, M., DI MARZO SERUGENDO, G., FREI, R., AND CABRI, G. 2009. Methodologies for self-organising systems: a SPEM approach. Tech. rep., BBKCS-09-05, School of Computer Science and Information Systems, Birbeck College, London, UK.

RALYTÉ, J. AND ROLLAND, C. 2001. An approach for method reengineering. In *Conceptual Modeling*, H. Kunii, S. Jajodia, and A. Solvbe, Eds. LNCS Series, vol. 2224. Springer Berlin Heidelberg, 471–484.

RICCI, A., VIROLI, M., AND OMICINI, A. 2005. Programming MAS with artefacts. In *Programming Multi-Agent Systems*, R. P. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds. Vol. LNAI 3862. Springer, Berlin Heidelberg, 206–221.

SCHOELER, T. AND MUELLER-SCHLOER, C. 2005. An observer/controller architecture for adaptive reconfigurable stacks. In *Systems Aspects in Organic and Pervasive Computing*. LNCS Series, vol. 3432. Springer Berlin Heidelberg, 139–153.

SEIDITA, V., COSSENTINO, M., HILAIRE, V., GAUD, N., GALLAND, S., KOUKAM, A., AND GAGLIO, S. 2010. The metamodel: a starting point for design processes construction. *Int. Journal of Software Engineering and Knowledge Engineering 20,* 4, 575–608.

WOOLDRIDGE, M. 2002. *An Introduction to Multiagent Systems*. J. Wiley, New York, USA.

Received February 2007; revised March 2009; accepted June 2009

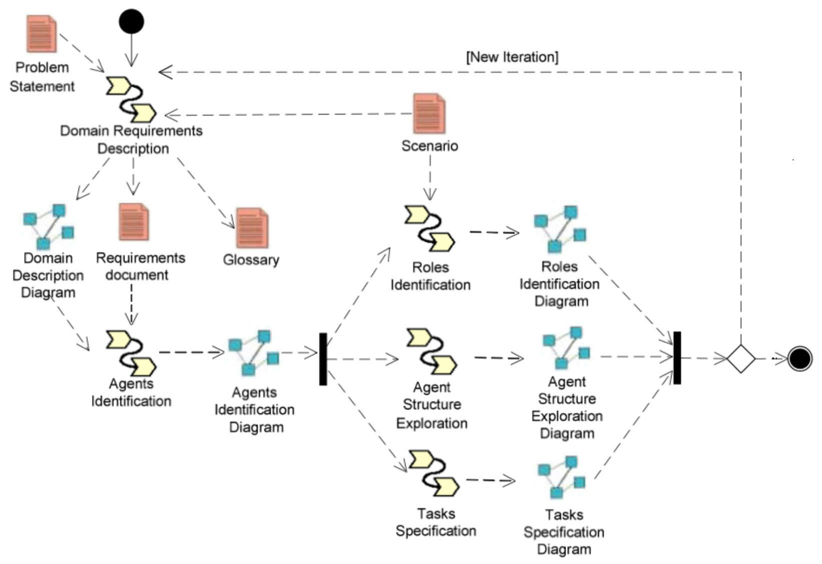
Online Appendix to: A method fragments approach to methodologies for engineering self-organising systems

MARIACHIARA PUVIANI, University of Modena and Reggio Emilia, Italy
GIOVANNA DI MARZO SERUGENDO, University of Geneva, Switzerland
REGINA FREI, Imperial College London, United Kingdom
GIACOMO CABRI, University of Modena and Reggio Emilia, Italy

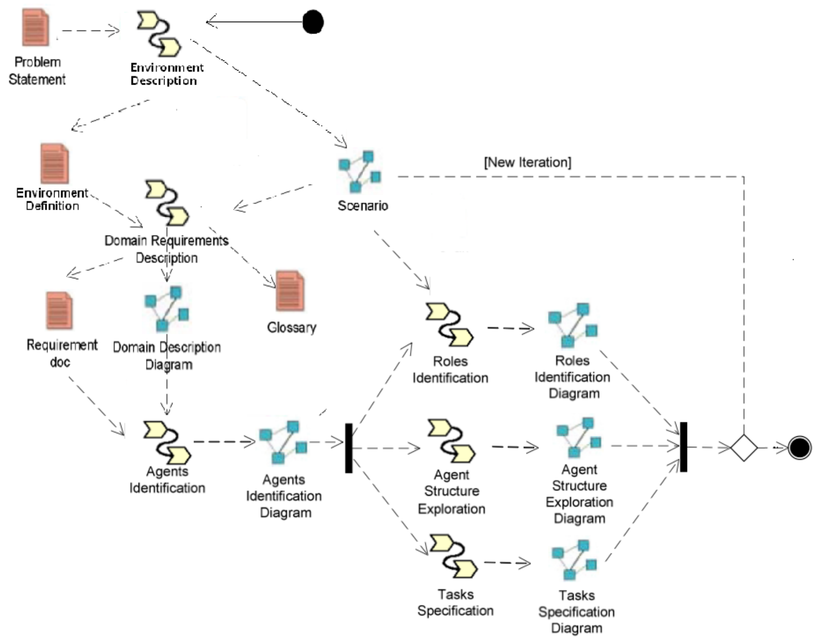
© 2010 ACM 1539-9087/2010/03-ART39 $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

	Requirements		Design / Model			Implementation	Simulation	Test / evaluation
	Environment	*Self-* properties*	*Architecture*	*Self-organising mechanism*	*Model*			
Adelfe	Environment and its characteristics			NCS	Agent	NCS rules		Tools
CUP				Patterns	Information flow; localities			Equation-free macroscopic analysis
MetaSelf	Coordination space environment	Self-* requirements	Patterns	Patterns, mechanisms for self-org. and dependability	Environment Agents Metadata Rules, policies	Rules Policies Metadata	Design of rules for self-organisation	Identifying potential faults (FMEA)
GM			Abstraction levels	Friction Synergy	Agent		Refined design	
SimDriven	Environmental agents, artefacts				Agents, environmental agents, artefacts		Design	

Fig. 17. Methodologies comparison

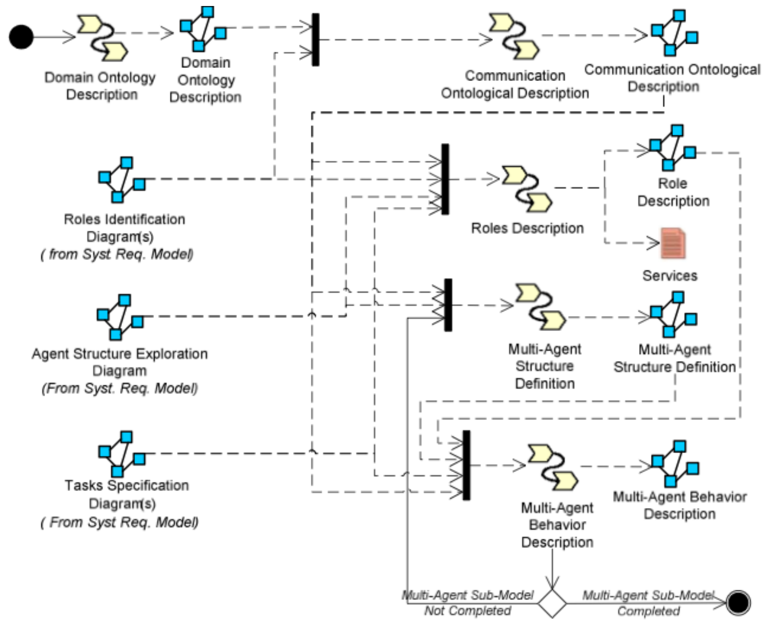

Problem Statement
[New Iteration]
Domain Requirements Description
Scenario
Domain Description Diagram
Requirements document
Glossary
Roles Identification
Roles Identification Diagram
Agents Identification
Agents Identification Diagram
Agent Structure Exploration
Agent Structure Exploration Diagram
Tasks Specification
Tasks Specification Diagram

(a) System Requirements phase (original)

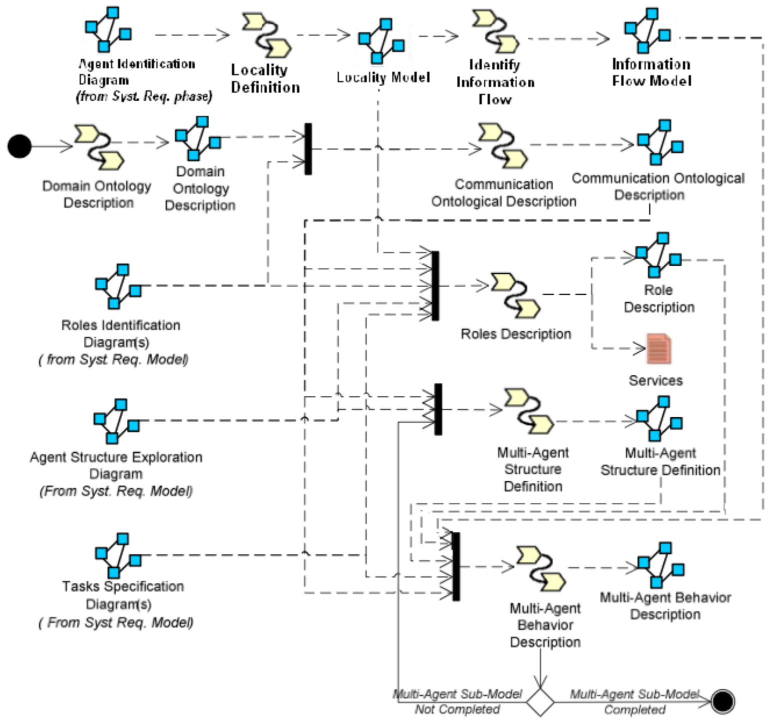

Problem Statement
Environment Description
Environment Definition
Scenario
[New Iteration]
Domain Requirements Description
Requirement doc
Domain Description Diagram
Glossary
Roles Identification
Roles Identification Diagram
Agents Identification
Agents Identification Diagram
Agent Structure Exploration
Agent Structure Exploration Diagram
Tasks Specification
Tasks Specification Diagram

(b) System Requirements phase (modified)

Fig. 18. PASSI2: System Requirements Phase


Domain Ontology Description
Domain Ontology Description
Communication Ontological Description
Communication Ontological Description
Roles Identification Diagram(s) (from Syst Req. Model)
Roles Description
Role Description
Services
Agent Structure Exploration Diagram (From Syst. Req. Model)
Multi-Agent Structure Definition
Multi-Agent Structure Definition
Tasks Specification Diagram(s) (From Syst Req. Model)
Multi-Agent Behavior Description
Multi-Agent Behavior Description
Multi-Agent Sub-Model Not Completed
Multi-Agent Sub-Model Completed

(a) Agent Society phase (original)


Agent Identification Diagram (from Syst. Req. phase)
Locality Definition
Locality Model
Identify Information Flow
Information Flow Model
Domain Ontology Description
Domain Ontology Description
Communication Ontological Description
Communication Ontological Description
Roles Identification Diagram(s) (from Syst Req. Model)
Roles Description
Role Description
Services
Agent Structure Exploration Diagram (From Syst. Req. Model)
Multi-Agent Structure Definition
Multi-Agent Structure Definition
Tasks Specification Diagram(s) (From Syst Req. Model)
Multi-Agent Behavior Description
Multi-Agent Behavior Description
Multi-Agent Sub-Model Not Completed
Multi-Agent Sub-Model Completed

(b) Agent Society phase (modified)

Fig. 19. PASSI2: Agent Society Phase