Towards an Agent-driven Software Architecture Aligned with User Stories

Yves Wautelet[1], Samedi Heng[2], Manuel Kolp[2] and Christelle Scharff[3]
[1]*Faculty of Economics and Business, KULeuven, Brussels, Belgium*
[2]*Louvain School of Management, Université Catholique de Louvain, Louvain-La-Neuve, Belgium*
[3]*Seidenberg School of Computer Science and Information Systems, Pace University, New York, U.S.A.*

Keywords: Agent Architecture, Agile Development, User Story, Agile Architecture, Multi-agent System.

Abstract: Agile principles have taken an increasing importance in the last decades. Software Architecture (SA) definition is perceived as a non-agile practice as it is executed in a top-down manner, reminding waterfall development, and sometimes imposes heavy documentation. This paper proposes to systematically build an agent-oriented SA from a set of User Stories (US), the core artifact to document requirements in agile methodologies. Previous research has allowed to define a unified US meta-model for the generation of templates relating WHO, WHAT and WHY elements. This meta-model's elements define a syntax issued from practitioners templates associated with semantics from Goal Oriented Requirements Engineering frameworks, more precisely i*. With a set of US following the templates of this previous model, the link between the US and SA concepts is systematically studied and a transformation process is proposed. The SA can decline agent behaviors aligned with requirements and organizational behaviors. Moreover, requirements (thus US) are subject to evolution through agile iterations; the SA can evolve with these changes in a semi-automatic manner. We thus argue that the Agent-SA produced with our transformation process contributes to the overall project agility.

1 INTRODUCTION

(Abrahamsson et al., 2010) highlights the core opposition between *Software Architecture* (*SA*) definition within a software project – seen as a top-down practice implying heavy documentation – and agile development. The Agile Manifesto emphasizes four values in agile development: (i) Individuals and interactions over processes and tools; (ii) Working software over comprehensive documentation; (iii) Customer collaboration over contract negotiation; and (iv) Responding to change over following a plan. Agile principles seem thus opposed to SA design unless the SA can evolve together and parallel to the requirements. Such an evolution would be eased if the SA would follow agile-style requirements definition. In agile methodologies, requirements are expressed using *User Stories* (*US*). US are written in natural language by the customer and they describe user functionalities at an abstract level.

In this work, we consequently suggest to systematically derive part of the SA from the set of US of a project in the form of a multi-agent architecture better aligned with agile principles. At the end, we want to dispose of a SA definition that is not heavy but can be partly automated and changed with the evolution of the US models, only if they are properly structured following the model defined in (Wautelet et al., 2014). This model's elements define a syntax issued from practitioners templates associated with semantics from Goal Oriented Requirements Engineering (GORE) frameworks, more precisely i* (Yu et al., 2011). The implementation of the agent-architecture is outside the scope of this paper, but an implementation model in line with the SA can be found in (Kiv et al., 2012).

2 STRUCTURING USER STORY CONCEPTS

This section exposes the "building blocks" used within this research.

2.1 Macro-level: Ways to Organize User Stories

We first distinguish a US macro-level dealing with the

Wautelet, Y., Heng, S., Kolp, M. and Scharff, C.
Towards an Agent-driven Software Architecture Aligned with User Stories.
DOI: 10.5220/0005706103370345
In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016) - Volume 2*, pages 337-345
ISBN: 978-989-758-172-4

grouping of US into depending (i.e. relating) sets. We thus consider here the US as potential building blocks serving for SA composition/decomposition. Figure 1 presents our meta-model of the US concepts.

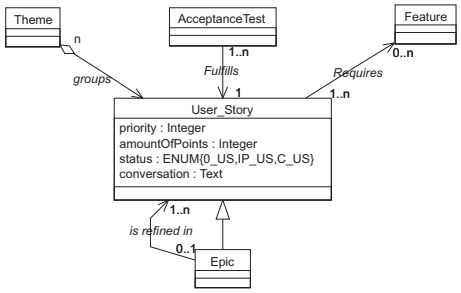

Figure 1: US as Macro-Level Structures: Meta-Model.

The *User_Story* class represents the US as a whole. *US* are written by the customer or product owner at the earliest stages of the project and put in the product backlog with a (implementation) *priority* and an *amountOfPoints* which refers to the number of *User Story Points* (*USP*)[1] (Cohn, 2004; Leffingwell, 2010). As a result, these elements have been added as attributes to the *User_Story* class. Additional attributes required for process management are also included within the *User_Story* class.

The concept of US is by essence granular in the sense that some US need to be refined into other ones since they are too abstract to be estimated, implemented and tested in their initial forms. These US are called *Epics* (Cohn, 2004). Epics are US with a high level of abstraction meaning that they must be refined/decomposed into smaller US to fully describe the requirement they refer to. These US are represented by the class *Epic* that inherits from the US class. As we will see later on, Epic US contain coarse-grained process elements that cannot be transformed at once in the SA (see Section 4.2).

Other concepts evoked in the meta-model of Figure 1 are non-relevant for the present research and will not be discussed.

2.2 Micro-level: Decomposing a User Story in *Descriptive_Concepts*

Within Figure 2, the meta-model of the previous section is enriched with the constituting elements of the US. We refer to this US view as the micro-level.

Rather than using the US as a whole within the requirements analysis process, we suggest, in our research design, to decompose the US on the basis of their *WHO*, *WHAT* and, when available, *WHY* dimensions. To ensure uniformity, these elements are all characterized as *Descriptive_Concepts* (*D_C*). When decomposed into a set of *D_C*, the dependency between *D_C* is intended to be further studied (see Section 2.3). Each element of a US template relating to one of the 3 dimensions is then an instance of the *D_C* class. For the template of a US: *As a <role>, I need a <task> so that <goal>*, we have 3 instances of the *D_C* class: one for *role*, one for *task* and one for *goal*. The *dimension* attribute thus compulsorily takes one of the values *WHO* (for *role*), *WHAT* (for *task*) or *WHY* (for *goal*) and the *syntax* attribute takes the syntax of the concept name (role, task, goal, ...). The *semantic* attribute relates to the definition of the *D_C*. The list of all the possible *D_C* is given in the form of a meta-model allowing to define US templates in Section 2.3.

Finally, since different *D_C* can be linked together, we introduce the *Link* class that represents the possible types of links between two *D_C*. Typically such links can be (at least partially) found in a logical way through agents' behavior at the level of the SA (see Section 6).

2.3 Unified-model of User Stories' *Descriptive_Concepts*

(Wautelet et al., 2014) proposes to build a unified model for designing US templates. The interested reader will refer to the latter reference for research and process related details. Figure 3 represents the meta-model of US templates. A US template can be designed from taking an element of the WHO, WHAT and possibly WHY dimensions. The link between the classes conceptually represents the link from one dimension to the other. Specifically, the unidirectional association from the *Role* to one of the *Capability*, *Task* or *Goal* classes implies that the target class instantiates an element of the WHAT dimension (always tagged as *wants/wants to/needs/can/would like* in the model). Then, the unidirectional association from one of these classes instantiating the WHAT dimension to one of the classes instantiating the WHY dimension (always tagged as *so that* into the model) implies that the target class eventually (since 0 is the minimal cardinality) instantiates an element of the WHY dimension. A US template supported by our model is for instance: *As a <role>, I would like <task> so that <hard-goal>*.

Each concept is associated with a particular syntax (identical to the name of the class in Figure 3) and a semantic. The syntax and semantics of the model are summarized here. As a result of the research

[1]The amount of USP represents the estimated effort required to implement the US.

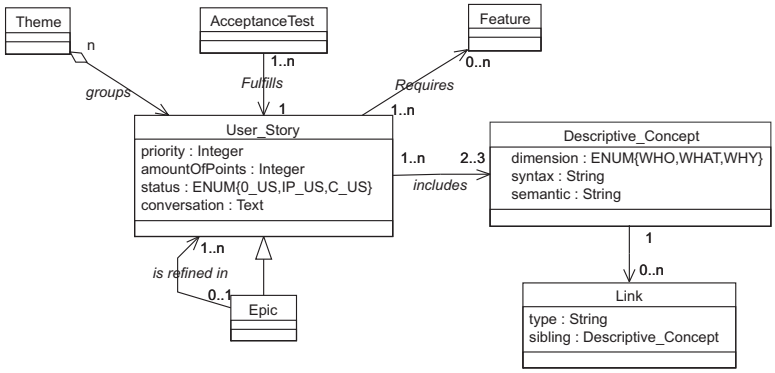

Figure 2: US as Macro and Micro-Level Structures: Meta-Model.

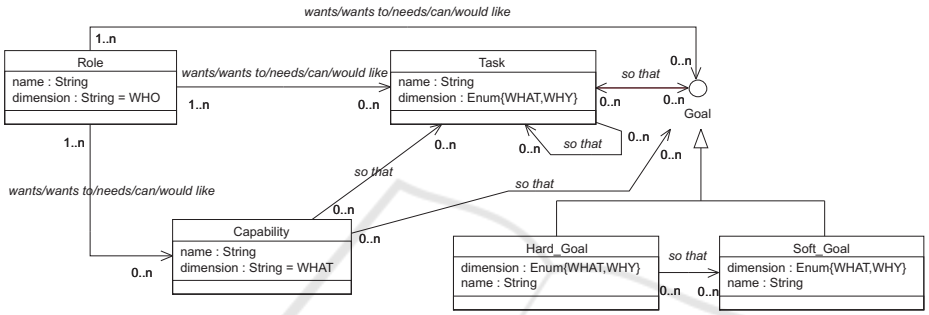

Figure 3: Unified Model for User Story Descriptive Concepts.

conducted in (Wautelet et al., 2014), all these couples syntax/semantic are issued of (Yu et al., 2011). Specifically:

- *A Role is an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor*;
- *A Task specifies a particular way of attaining a goal*;
- *A Capability represents the ability of an actor to define, choose, and execute a plan for the fulfillment of a goal, given certain world conditions and in the presence of a specific event*;
- *A Hard-goal is a condition or state of affairs in the world that the stakeholders would like to achieve*;
- *A Soft-goal is a condition or state of affairs in the world that the actor would like to achieve. But unlike a hard-goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated soft-goal*.

More explanations for distinguishing *Hard-goal*, *Task* and *Capability* elements will be discussed in Section 4.

3 A STRUCTURE FOR DEFINING THE SOFTWARE ARCHITECTURE

This section exposes the main building blocks of the abstract agent-architecture that will be used to map US concepts in order to automatically align part of the SA with the collected requirements. Examples of each of the elements are provided in the illustrative example of Section 6.

3.1 Agent Structure

Figure 4 proposes a generic template for the *Agent* concept. The definition of an Agent is composed of five parts: *Attributes*, *Events*, *Plans*, *Beliefs* and *Methods*.

We typically refer in this work to BDI agent implementation platforms such as JADE, JACK Intelligent Agents or the Jadex BDI Agent System (Pokahr et al., 2005; Pokahr et al., 2013; Pokahr et al., 2014).

The agent class allows to specify:

- the declaration of agent attributes;
- the events (both internal and external) that the agent (i) handles, (ii) can post internally (to be

handled by other plans), or (iii) can send externally to other agents;

- the plans that the agent can execute;
- the beliefs the agent can use and refer to. The beliefs of an agent can be of type *private*, *protected*, or *public*. A *private* access is restricted to the agent to which the belief belongs. *Protected* access is shared with other agents of the same class, while *public* access is unrestricted;
- the declaration of agent methods (e.g., the constructor of an agent).

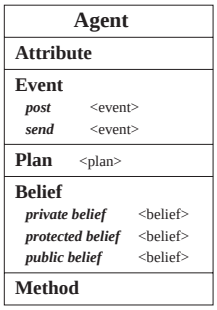

Figure 4: Agent template.

3.2 Event Structure

Events describe stimuli, emitted by agents or automatically generated, in response to which other or the same agents must take action.

Events are the origin of all activity within an agent-oriented system. In the absence of events an agent stays idle. Whenever an event occurs, an agent initiates a task to handle it. This task can be thought of as a thread of activity within the agent. The task causes the agent to choose between the plans it has available, executing a plan or a set of plans until it succeeds or fails.

There are different event types, each with different uses. Events can be described along three dimensions:

- *External* or *internal* event: external events are sent to other agents while internal events are posted by an agent to itself;
- *Normal* or *BDI event*: an agent has a number of alternative plans to respond to a BDI event and only one plan in response to a normal event. Whenever an event occurs, the agent initiates a plan to handle it. If the plan execution fails and if the event is a normal event, then the event is said to have failed. If the event is a BDI event, a set of plans can be selected for execution and these are attempted in turn. If all selected plans fail, the event is also said to have failed;

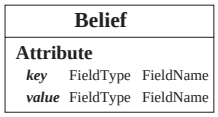

Figure 5: Belief template.

- *Automatic* or *nonautomatic* event: an automatic event is automatically created when certain belief states arise.

3.3 Belief Structure

A *Belief* describes a piece of knowledge that an agent has about itself and its environment.

Figure 5 shows a template of a belief. Every belief that an agent currently has is represented as tuples. It has key and value fields. The `key FieldType Fieldname` declaration describes the key attributes, while the `value FieldType FieldName` declaration describes the data attributes of each belief. A belief may have zero or more key or value field declarations.

3.4 Plan Structure

A *plan* describes a sequence of actions that an agent can take when an event occurs.

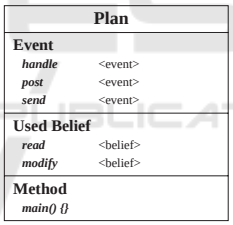

Figure 6: Plan template.

Plans are structured in three parts as shown in Figure 6: the *Event* part, the *Belief* part, and the *Method* part. The *Event* part declares events that the plan handles (i.e., events that trigger the execution of the plan) and events that the plan produces. The latter can be either posted (i.e., sent by an agent only to itself) or sent (i.e., sent to other agents). The *Belief* part declares beliefs that the plan reads and those it modifies. The *Method* part describes the plan itself, that is, the actions performed when the plan is executed. In the following, we study in more detail each part of the plan template.

3.4.1 Events in Plans

```
handle <event>
```

Whenever an agent detects that an event arises, it tries to find a plan to handle the event.

The event the agent can handle is identified by its plans' `handle <event>` declarations. When an instance of a given event arises, the agent may execute one of the plans that can handle this event.

The `handle <event>` declaration is mandatory. Whenever an instance of this event occurs, the agent will consider this plan as a candidate response. Without a `handle <event>` declaration, a plan would never be executed by an agent.

```
post <event>
```

The `post <event>` statement declares that the plan is able to post this event when executed. This may be in one of the methods declared in the Method part. Only events that the agent posts internally are declared after the post statement.

```
send <event>
```

The `send <event>` declaration is similar to the `post <event>` declaration, except that it identifies events that the plan can send to other agents. A plan is able to send an event through one of the methods declared in the Method part.

3.4.2 Used Beliefs in Plans

```
read <belief>
```

The `read <belief>` declaration indicates that the `<belief>` is to be read (and only read) within the plan.

```
modify <belief>
```

The `modify <belief>` statement indicates that the `<belief>` may be read and modified within the plan.

3.4.3 Methods in Plans

All the plan's methods are declared in the method part. Two of these methods are worth to be pointed out: `context()` and `main()`.

An agent may further discriminate plans that handle an event by determining whether a plan is *relevant*. The `context()` method allows the agent to determine which plan to execute when a given event occurs. To be relevant, the plan must declare that it is capable of handling the event that has arisen (via the `handle <event>` declaration) and that it is relevant to the event instance (via the `context()` method).

`context()` is a boolean method. If this method returns true, the plan is relevant to the event. If not, the plan is not relevant to the event.

The `main()` method is executed whenever a plan is executed. The `main()` method is just like the `main()` method in Java – it represents the starting point in a plan's execution.

4 BRIDGING A SET OF US TO A MULTI-AGENT SOFTWARE ARCHITECTURE

The aim of this Section is, starting from the *Descriptive_Concepts* defined into the model presented in Section 2.3, to study the transformation possibilities to the element constituting the agent SA.

4.1 Role

A *Role* within a US is transformed in an intentional Agent in the SA. This way, an organizational role (which is most often a software user) is literally mapped to an intelligent acting entity at run-time into the system.

4.2 Hard-goal

The *Hard-goal* is the most abstract element since it is a condition or state of affairs that must be attained but there is no defined way to attain it or more precisely several ways could be followed in practice contrarily to the task that represents an operational way to attain a *Hard-goal*. An example of a *Hard-goal* could, for example, be to *Graduate for a diploma in Business Information Management*; it can be the *Hard-goal* of a student but there are several ways to attain this *Hard-goal* .

Being highly abstract, Hard-goals are not right away transformed into a functional aspect in the software architecture. US containing *Hard-goals* can be (but not necessarily) Epic US. Their satisfaction is rather supported by one or several *Task(s)* and/or *Capabilities* that will themselves be transformed into the agent architecture (see Section 4.3).

4.3 Task & Capability

The *Task* and the *Capability* represent more concrete and operational elements but these two need to be distinguished. The *Capability* could in fact be modeled as a *Task* but the *Capability* has more properties than the former since it is expressed as a direct intention from a role. In order to avoid ambiguities in interpretation, we use of the *Capability* element only for *an atomic Task* (i.e., a task that is not refined into other

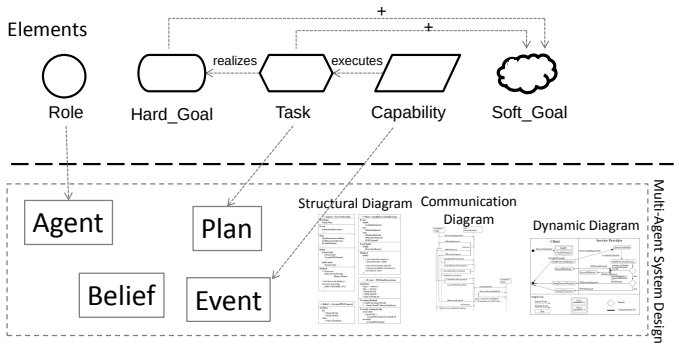

Figure 7: The Forward Engineering Process.

elements but is located at the lowest level of hierarchy). A *Task* could then be *Take the Business Intelligence Methods class* and a *Capability* would be *Fill in the Exam*.

The *Task* representing a whole process composition will be transformed as a *Plan*, while *Capability*, by nature atomic, will be transformed in an *Event*.

4.4 Soft-goals

Soft-goals are not transformed at operational level but other *Hard-goals* and *Tasks* contribute positively or negatively to their satisfaction. These latter *Hard-goals* and *Tasks* are themselves transformed into the agent architecture as evoked above.

5 SOFTWARE TRANSFORMATION PROCESS

Figure 7 graphically illustrates the transformation process defined in the previous section. With respect to the SA:

- A *Role* element is transformed into an *Agent*;
- A *Task* element is transformed into a *Plan*;
- A *Capability* element is transformed into an *Event*.

Let us also note that *Domain Entities* (*DE*) manipulated by the Agents to fulfill the elements evoked above will form the Beliefs in the SA. DE are not expressed as such in US but rather physical or logical resources manipulated by *Roles* in the realization of US. This means they are derived from a careful reading of the US.

On the bottom right side of Figure 7, we can also note three types of diagrams – the *Structural Diagram* (*SD*), the *Communication Diagram* (*CD*) and the *Dynamic Diagram* (*DD*) – representing complementary views of the agents properties and behavior at system level. Due to a lack of space and with the willingness to focus on the transformation process only, we will, in the context of this paper, focus on the SD only.

6 ILLUSTRATIVE EXAMPLE

This section exposes the illustrative example on which we apply the SA transformation process. Note that this illustrative example is taken from a bigger case but that, due to a lack of space, the set of US chosen and their transformation into the SA is limited.

6.1 User Stories

Our proposal will be illustrated using a running example about carpooling. Carpooling deals with sharing car journeys so that more than one person travels within a car. In this context, it becomes increasingly important to save gas, reduce traffic, save driving time and control pollution. *ClubCar* is a multichannel application available as an Android application, SMS service and IVR system. Users of ClubCar are riders and/or drivers that can register by SMS, voice or through an Android app. Roughly speaking, the software allows drivers to propose rides and submit their details with *dates*, *times*, *sources* and *destinations* while riders can search for available rides (Shergill and Scharff, 2012).

As shown in Table 1, we have taken a sample of the US of the ClubCar application to illustrate the research developed in this paper. The first column depicts the *Dimension* of US *D_C*, the second column describes the element itself and the last column gives the type of the *D_C*[2].

[2]Note that, when there were several possibilities, an ad-hoc choice has been made since envisaging the multiple modeling solutions at the level of the unified template model is not the focus of the present research.

Table 1: US sample issued of the ClubCar development.

#	*Dimension*	*Element*	*D_C Type*
1	WHO WHAT	*As a DRIVER* *I want to propose rides to Riders*	Role Hard-goal
2	WHO WHAT	*As a DRIVER* *I want to propose a ride from A to B with the price location and time of departure, and number of seats available*	Role Task
3	WHO WHAT WHY	*As a DRIVER* *I want to log in to the platform* *so that I can register to the service*	Role Capability Task
4	WHO WHAT	*As a DRIVER* *I want to select the ride characteristics*	Role Capability
5	WHO WHAT	*As a DRIVER* *I want to confirm the proposal*	Role Capability
6	WHO WHAT	*As a RIDER* *I want to order a ride*	Role Capability

6.2 Agent Architecture

US1 contains a *Hard-goal* element in the WHAT dimension and constitutes an Epic US so that it is not forwarded as such to the SA but is satisfied through the realization of the *Task* element in US2. Nevertheless we can "manually" identify the presence of a DE which is the `Ride` that will be transformed into a Belief (see hereunder).

As illustrated in Figure 8, the **ProposeRideFromAtoB** Plan is forwarded from the Task element depicted in US2. It is used by the `Driver` Agent to propose a ride to the `Rider` Agent[3]. The Plan is executed when the `CharacteristicsSelected` Event (containing the information about the characteristics of the `Ride` offered by the `Driver`) occurs. The latter Event is forwarded from US4. The **ProposeRideFromAtoB** Plan also creates a `Ride` Belief, i.e. the `Driver` belief storing the proposed `Ride` that can then be proposed to a `Rider` Agent. Later, the `Rider` Agent can subscribe for the proposed `Ride`.

DriverProposalConfirmation is an Event transformed from US5 that is posted automatically whenever the value of the `status` field (of the `Ride` Belief) is set to submitted. It will then invoke a plan to inform the Rider that the Ride is available.

The `RiderOrder` Event (not illustrated in Figure 8) belonging to the `Rider` Agent is sent when the `Rider` Agent subscribed to a `Ride` offered by the `Driver` Agent. This Event is transformed from the Capability element in US6.

As evoked in the previous section, other agent behavior is outside the scope of the this paper that focuses on systematic forward engineering of US to a SA, therefore the full SD as well as CD and DD are not represented here.

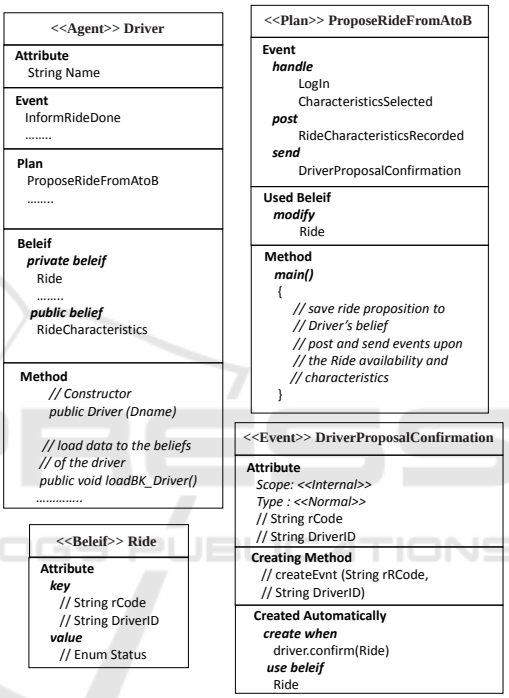

Figure 8: Structural Diagram – ClubCar Example.

7 RELATED WORK

Agile development and SA have been identified as the fourth position among the top ten research questions that should be addressed by the community (Freudenberg and Sharp, 2010). The actual researches on this area are focused on quantity of SA rather than quality (Babar et al., 2013). SA is seen as a human activity based on designers experience meaning that most architectures are ad-hoc and only some follow a defined pattern (Booch, 2008). Many practitioners have argued that model-driven engineering as well as user-center design are beneficial for constructing a SA in the agile environment (Babar et al., 2013). We point to the development of a SA not following patterns but aligned on the structure of requirements.

[3]Due to a lack of space the `Rider` Agent has not been represented in Figure 8.

To the best of our knowledge, there is no work on mapping the US to agent SA. In most existing research, US are generated and decomposed from the SA elements. (Leffingwell, 2010) proposes the scaled agile framework that emphasizes on the importance of building the SA to support development. They however do not furnish any mechanism for building and and tracing the SA. (Perez et al., 2014) propose a similar framework that bridges US to SA elements; US are nevertheless considered as fine-grained elements only.

We can also focus on the use of US in agent methods. In Agile PASSI (Chella et al., 2006), US are used as requirement artifact for communication but its their only usage. US, specifically in MaSE (DeLoach et al., 2002), is one source of requirements for capturing the goal of the agent (Wood and DeLoach, 2001) but without formal transformation. (Gaur and Soni, 2012) use fuzzy theory to provide a systematic and methodical approach to explore various dependencies such as goal, task, resource or soft-goal from US. Again the technique is limited at analysis stage with no transformation. In contrast, (Tenso and Taveter, 2013) adopt agent-oriented modeling in agile development. They provide a method for decomposing a goal into subgoals and link them to US elements. (Carrera et al., 2014) use US as testing mechanism for agent-oriented software development. Only one US template is used and aligned to JBehave (http://jbehave.org).

8 CONCLUSION

The quest for agility has, in the recent years, been conducted in several domains and, behind the buzz word, companies are looking for a high-level of alignment between user requirements and software as well as rapid respond to change. This paper has proposed to map US – *THE* requirements models of agile software development methods like SCRUM and XP – with an agent-based SA in order to implement a software system made of Agents mapping organizational behavior at run-time. The transformation process can be partially automated for rapid response to change and to avoid a heavy documentation process.

Future work includes a full validation on the ideas on a broader case study. A plug-in supporting the automatic transformation of US elements into the proposed SA is under development within the DesCARTES Architect CASE Tool.

REFERENCES

Abrahamsson, P., Babar, M. A., and Kruchten, P. (2010). Agility and architecture: Can they coexist? *IEEE Software*, 27(2):16–22.

Babar, M. A., Brown, A. W., and Mistrik, I. (2013). *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Morgan Kaufmann Publ.

Booch, G. (2008). Architectural organizational patterns. *Software, IEEE*, 25(3):18–19.

Carrera, Á., Iglesias, C. A., and Garijo, M. (2014). Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Info. Syst. Frontiers*, 16(2):169–182.

Chella, A., Cossentino, M., Sabatucci, L., and Seidita, V. (2006). Agile passi: An agile process for designing agents. *International Journal of Computer Systems Science & Engineering*, 21(2):133–144.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison Wesley.

DeLoach, S. A., Matson, E. T., and Li, Y. (2002). Applying agent oriented software engineering to cooperative robotics. In *FLAIRS Conference*, pages 391–396.

Freudenberg, S. and Sharp, H. (2010). The top 10 burning research questions from practitioners. *IEEE Software*, 27(5):8–9.

Gaur, V. and Soni, A. (2012). A novel approach to explore inter agent dependencies from user requirements. *Procedia Technology*, 1:412–419.

Kiv, S., Wautelet, Y., and Kolp, M. (2012). Agent-driven integration architecture for component-based software dvpt. *Trans. Comp. Collect. Intell.*, 8:121–147.

Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional.

Perez, J., Diaz, J., Garbajosa, J., and Yage, A. (2014). Bridging user stories and software architecture: A tailored scrum for agile architecting. In *Agile Software Architecture*, pages 215–241.

Pokahr, A., Braubach, L., Haubeck, C., and Ladiges, J. (2014). Programming BDI agents with pure java. In Müller, J. P., Weyrich, M., and Bazzan, A. L. C., editors, *Multiagent System Technologies - 12th German Conference, MATES 2014, Stuttgart, Germany, September 23-25, 2014. Proceedings*, volume 8732 of *LNCS*, pages 216–233. Springer.

Pokahr, A., Braubach, L., and Jander, K. (2013). The jadex project: Programming model. In Ganzha, M. and Jain, L. C., editors, *MAS and Applications - Volume 1: Practice and Experience*, volume 45 of *Intelligent Systems Reference Library*, pages 21–53. Springer.

Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *MAS, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer.

Shergill, M. P. K. and Scharff, C. (2012). Developing multichannel mobile solutions for a global audience: The

case of a smarter energy solution. *SARNOFF'12, New Jersey*.

Tenso, T. and Taveter, K. (2013). Requirements engineering with agent-oriented models. In *ENASE*, pages 254–259.

Wautelet, Y., Heng, S., Kolp, M., and Mirbel, I. (2014). Unifying and extending user story models. In *CAiSE 2014, Thessaloniki, Greece, June, 2014. Proceedings*, volume 8484 of *LNCS*, pages 211–225. Springer.

Wood, M. F. and DeLoach, S. A. (2001). An overview of the multiagent systems engineering methodology. In *Agent-Oriented Soft. Eng.*, pages 207–221. Springer.

Yu, E., Giorgini, P., Maiden, N., and Mylopoulos, J. (2011). *Social Modeling for Requirements Engineering*. MIT Press.