SADAAM: Software Agent Development An Agile Methodology

Neil Clynch[1] and Rem Collier[1]

School of Computer Science and Informatics, University College Dublin, Ireland

Abstract. This paper presents SADAAM, an agent development methodology that utilises agile techniques to facilitate the development and implementation of multi-agent systems. We illustrate SADAAM through a worked example from the Supply Chain Management domain, and implement a partial system using the Agent Factory Framework.

1 Introduction

Agile Development [1] [2] methods are currently being successfully employed across organisations of all sizes, and are increasingly becoming integral to enterprise, mainstream deployments within industry. Agile approaches typically focus on delivering working software in small increments, providing minimal documentation, and minimizing risk by developing software in shorter time iterations that contain every task required to successfully release a small increment of additional functionality. Generally, agile methods provide benefits in an environment where requirements are evolving and changing quickly; but are widely considered unsuitable for larger projects, and projects that necessitate critical, reliable and strict safety requirements.[1]

The last decade has witnessed an emergence of numerous Agent-Oriented Software Engineering (AOSE) [22] [21] methodologies [8] [27] [29] based on an assortment of conceptual frameworks, notions, techniques, and methodological steps. Although many of these methodologies typically employ more traditional approaches to development, in the context of agent-based systems, we believe it can (in some cases) be too excessive because the complexity of current industrial strength applications necessitates a more resource-intensive approach. Subsequently, there is an urgent requirement to provide a more efficient methodology for the development of software agent systems. Since the complex interaction scenarios and emerging behaviours between agents make pre-planning very difficult, agile practices appear to be a better choice than conventional engineering approaches for multi-agent system (MAS) development. By combining AOSE and agile techniques, it is possible to inject greater flexibility into the development cycle; thereby promoting an iterative, incremental, test driven approach to MAS development.

[1] [9] offer a risk analysis approach based on criteria that they label 'home-ground' for each end of an adaptive-predictive continuum, to help determine the risk of using either an agile or plan-driven method.

Addressing this issue provides a key motivation for the work presented here, namely the formualation of *SADAAM*, an agile methodology, based loosely on Agent UML [5], that supports the development of multi-agent systems. Specifically, section 2 outlines other existing approaches to agile agent development. Following this, section 3 presents details of SADAAM together with a worked example that is adapted from the Supply Chain Management domain. Finally, section 4 presents some concluding remarks.

2 Related Work

Agent-oriented methodologies can be categorized in terms of how software engineering paradigms influenced their evolution [18]. The majority of these methodologies primarily focus on design and analysis [4] [26] and to some extent implementation.

Tropos [10] provides an exception to this trait by handling the entire software development process; MaSE [28] attempts to facilitate a more complete development cycle, providing support from initial system specification through verification and implementation of the agent system; and MAS-CommonKADS [20] does define some explicit verification process based on model checking that is intended to support inter-agent communication [13]. Further, all of these approaches support implementation through the provision of varying levels of tool support. However, work that includes testing of MAS is limited [11].

More recently there are signs of a growth of interest in the potential of using agile methodologies for the development of agent-based systems. For instance, [23] presents an extreme programming approach that was successfully applied for the development of a prototypical MAS for clinical information logistics. Additionally, [12] introduces Agile PASSI, an agile process that aims to provide a more versatile approach to the development of agents for robotic applications. Lack of maturity, insufficient testing methods, and lack of support were identified as some of the main disadvantages of these approaches. In particular, because they are still in their infancy and thus somewhat limited, they fail to fully embrace the concept of agile development, and eschew important factors such as providing sufficient support for testing and documentation.

[13] presents a unit-testing approach toward MAS development, which allows agents to be tested individually through the use of *"Mock Agents"* that direct the design and implementation of agent unit test cases. Each Mock Agent is responsible for testing a single role of an agent under successful and exceptional scenarios. Aspect-oriented techniques are employed to manage the asynchronous execution of agents under test. One of the main drawbacks of this approach is that each Mock Agent is responsible for testing just one role of an agent. This *"Role-Driven Unit Testing"* limits the testing and development process, because in reality, an agent may adopt a number of roles (e.g. Agent Neil may adopt the role of Developer, Student, Husband, Father, etc). Additionally, each Mock Agent represents a single unit test on a given scenario, and reports a single test result. Thus, a large number of Mock Agents may need to be created when testing

agents that reside in more complex domains, rendering it inefficient, mainly because a significant amount of time and effort is expended designing and coding Mock Agents that are deemed redundant once the specific testing is complete. Furthermore, this approach appears to be limited to a single agent platform; and it fails to allow testing on deployed (live) applications. Finally, application code needs to be recompiled resulting in significant compliation overhead.

3 Methodology

SADAAM is an agent development methodology that utilises techniques derived from a variety of Agile Development methods including some of those found in the Agile Manifesto [2], Agile Modelling [3], SCRUM [25], Extreme Programming (XP) [6], Test-Driven Development (TDD) [7] and Refactoring [17] to facilitate the development and implementation of software agents into MAS. Central to

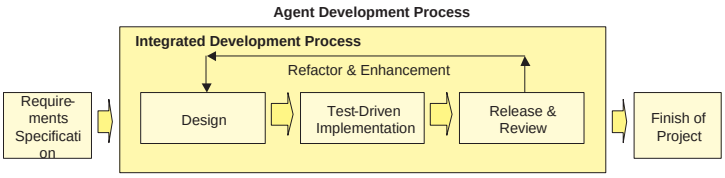

Fig. 1. Agent Development Process (ADP).

SADAAM is the Agent Development Process (ADP), which provides the core agile agent development process. The ADP implements an Integrated Development Process (IDP) that consists of four key phases: *Design*, *Test-Driven Implementation*, *Release and Review*, and *Refactor and Enhancement*, that are applied iteratively until a finished state is reached (Figure 1 presents a single iteration through the ADP). The ADP supports a bottom-up approach that increases flexibility and enables the development team to focus on the rapid delivery of working code, and to respond quickly to changes in requirements. SADAAM also includes some provision for requirements specification[2], and project completion (space constraints prevent a more detailed review here).

3.1 Design Phase

The Design phase for provides an incremental, iterative process for the analysis and design of autonomous agents (as illustrated in figure 2). Its purpose is to translate system requirements into design decisions. In particular, it defines and documents details of the business scenario design solution. Specifically, it

[2] SADAAM adopts a minimal approach to requirements capture, acknowledging that upfront specification typically leads to a large amount of wastage (approx. 64% [3]).

identifies system behaviours and the roles performed by associated agents whilst engaging in these system behaviours; it defines agents, their relationships, interactions, and activities; and defines agent classes and methods that form the solution. These attributes are realized via a minimal set of design artifacts that are based on Agent-UML[3]

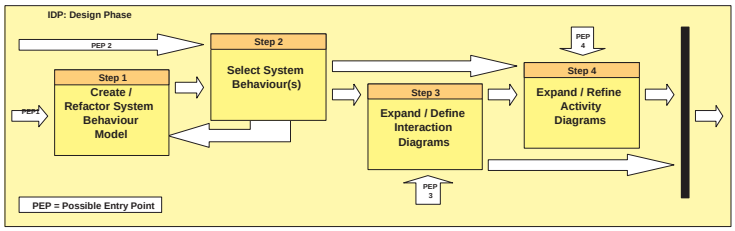

Fig. 2. IDP: Design Phase.

At the project level, SADAAM applies an adaptation of the Agile Model Driven Development process [3]) to our Design Phase (see figure 3a). Significant features of this model include: *Scope and Requirements Analysis*, *Initial Requirements Modeling*, *Expand on Initial Models*, and *SADAAM Model Storming sessions*, prior to implementation via the TDI phase.

Scope and Requirements Analysis helps determine the projects requirements; and includes an agile approach to Requirements Management that employs techniques such as implementation of requirements with highest priority first, flexibility in the management of new or removed requirements and priorities, and efficient allocation of resources to meet requirement tasks.

The Model Initial Requirements phase remains more of a high-level definition of the scope and requirements analysis, intended to help clarify any ambiguities in the requirements specification and provide the basis for dividing implementation into manageable steps. The level of detail should be simple enough to provide a high-level understanding of the system. The first session should establish enough knowledge to compose an initial System Behaviour Model (SBM) that provides enough information for the team to understand what the system does, how its entities are defined, and interaction between these entities. More detailed requirements are elicited through the expansion of this model.

SADAAM Model Storming Sessions call for the minimum amount of upfront modelling effort to turn requirements into models that can be implemented in a single iteration. These sessions will take place whenever a requirement model needs to be looked at in greater detail, and may involve re-assessing the requirements estimate, and refinement of the initial diagrams.

[3] SADAAM inherited these concepts from an earlier methodology [15] which employed Agent-UML as its modeling language.

Expanding on the initial models involves depicting the system behaviours in more detail. SADAAM's approach to Requirements Modelling (Figure 3b) employs the SBM to help identify entities involved in the system, and how participants operate in the system; and requires development of Organisational Unit and System Behaviour diagrams that identify the systems basic entities and relationships between them. The SBM is expanded via the Interaction and Activity models. Further refinement (of the models) and allocation of resources are performed as necessary. Finally, Review and Enhancement provides a further opportunity review and enhance the models.

This approach enables the development team to work iteratively through the design process, delivering working models in small increments; and it supports the rapid development and delivery of working code for release and review. This approach provides a number of benefits. In particular, it offers design flexibility, streamlines the development effort, and allows the development team to accommodate changes in requirements efficiently.

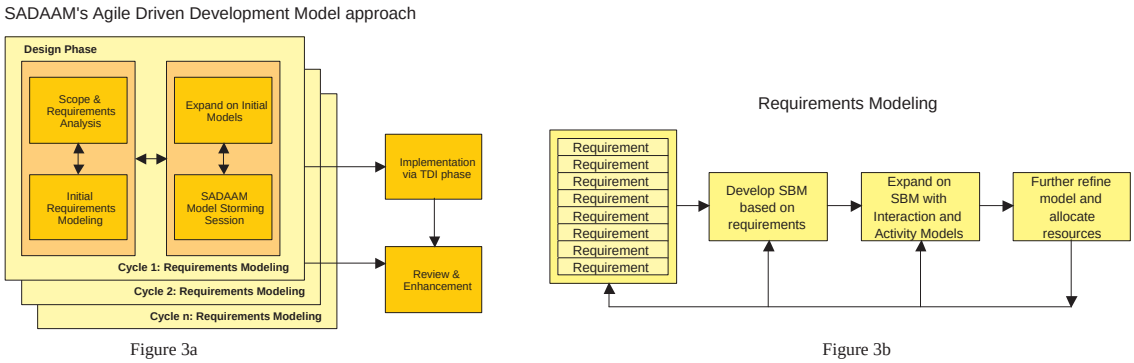

Fig. 3. Sample of the ADDM techniques applied in SADAAM to handle requests.

The principal design steps are as follows:

1. **Create/Refactor System Behaviour Model.** The *System Behaviour Model (SBM)* identifies key system behaviours for the given scenario. System behaviour is a clear set of activities and/or interactions that occur at some stage in the system operation [16]; and is represented as modified UML Use Case diagrams, where actors are stereotyped as agents, and use cases are stereotyped as system behaviours. SADAAM also applies the notion of organisational structure that underlies each given (business) scenario. Specifically, the SBM consists of an *Organisational Unit Diagram (OUD)* that outlines the organisational structure for the target system; and a set of *System Behaviour Diagrams (SBDs')* that outline the key system behaviours.
   Organisational structure is defined by breaking down the (business) scenario into an organisational hierarchy type model. Specifically, the OUD defines

organisations as *Root Organisations*, and sub-organisations as *Organisational Units* based on the business functions; and is responsible for the assignment of agents to specific organisational units (OUs). The OUD may also act as a roadmap to drive the creation of corresponding SBDs'.
Development of the OUD is a 3-step process that defines both agents and OUs in the scenario. This process is explained briefly via a worked example taken from the Supply Chain domain[4]:

(a) *Review of system requirements and business scenario:* In this example, the Retailer procures goods from the Supplier, which are subsequently delivered via an independent Haulage Firm. The scenario begins with the Retailer's Buyer requesting a quote for goods from the Supplier. Once the quote is received back from the Supplier, the Retailer Buyer places an order for goods. The Supplier's Warehouse Manager then contacts the Haulage Firm's Transport Manager to request their Services. Finally, the Haulage Firm collects the goods from the Supplier and delivers them to the Retailer's Inventory Department.

(b) *Define Organisation Structure and OUs in the environment you are attempting to model:* The OUD views our scenario as a structured environment where 3 individual OUs operate within clearly defined boundries, and cooperate systematically to conduct a transaction. Figure 4 formalises this view with an OUD that defines the following OUs: Retailer(complete with sub-organisational units Purchasing and Inventory); Supplier (comprising of Inventory and Sales sub-oranisational units); and Haulage Firm; that each reside under the Supply Chain Scenario root.

(c) *Assign Agents within the Organisational Structure:* This step associates agents to specific OUs, allowing the clear identification of agents required in the given scenario. In our example, a *Buyer* agent is assigned to the Retailer.Purchasing OU and a *WarehouseManager* agent to the Retailer.Inventory OU. Similar assignments are made for the Retailer and Haulage Firm OUs (as illustrated in figure 4a) [5].

Although SBDs' are adaptations of UML use case diagrams; unlike [16], SADAAM associates a SBD with each unit of the OUD, allowing each SBD to identify all system behaviours undertaken by agents that are part of that particular unit or one of its sub-units. Where a unit is itself comprised of sub-units, the SBD shows only inter-sub unit system behaviours (intra-sub unit behaviours are specified in the SBD for that sub-unit). This helps to reduce the complexity of the SBM, and allows designers to focus on inter-organisational unit behaviours seperately from intra-organisational unit behaviours. Dotted lines are introduced to the SBD to clarify the sub-organisational boundaries relevant to that unit.

[4] Please note: this supply chain scenario has been simplified for demontration purposes and is not meant to represent a complete system

[5] Because organisational hierarchies are somewhat constrained by their commercial environment, the OU model expects that agents will fall into a definitive organisational structure.

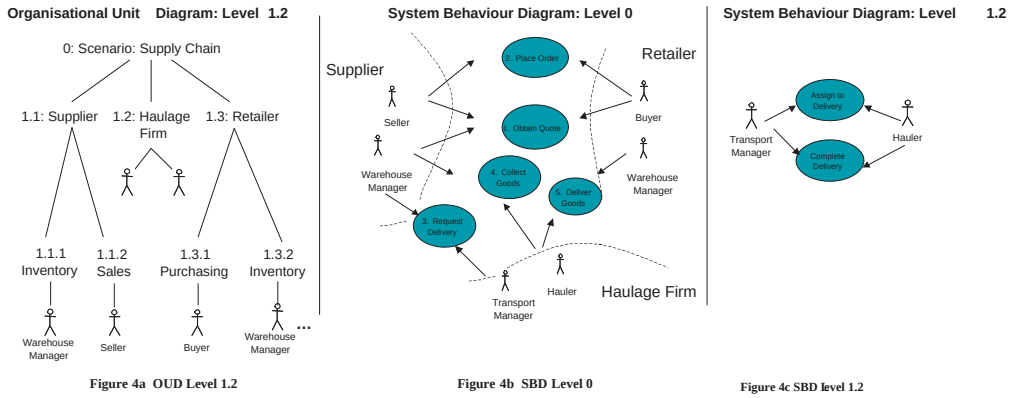

Figure 4a OUD Level 1.2 Figure 4b SBD Level 0 Figure 4c SBD level 1.2

Fig. 4. Supply Chain Scenario: System Behaviour Model Level 0 and level 1.2.

Figure 4b presents an SBD for the overall Supply Chain scenario. This diagram identifies a number of key system behaviours related to the procurement process that are realised by agents associated with various sub-units of the scenario. For example, the Supplier.Seller agent and the Retailer.Buyer agents engage in two key behaviours: obtaining a quote, and placing an order. The value of introducing the OUD is illustrated by figure 4c, which shows the SBD for the Haulage Firm sub-unit. This diagram presents only those system behaviours that are internal to that unit - all external system behaviours are specified in SBDs that relate to higher-level OUs.

2. **Select System Behaviours.** The second step involves selection of a sub-set of the overall system behaviours for further analysis. This step is essentially a scoping operation that identifies which parts of the system the development team is now going to work on, and which parts will be left until later. SADAAM appreciates that finding a suitable subset of behaviours is not trivial. As a result, the approach SADAAM applies is much less restrictive, and includes techniques that facilitate flexible modeling and deliverables, and frequent reviews coupled with team meetings that implement SADAAM Model Storming Sessions (introduced in section 3.1). This approach enables developers to initiate model development at any point, and administer changes to models at any given time.
3. **Expand/Define Interaction Model.** The *Interaction Model (IM)* defines interactions that occur between agents whilst playing roles during a specific system behaviour. The IM consists of a set of Agent UML Protocol Diagrams [19], that are associated with individual system behaviours (see Figure 5a). A given system behaviour may have zero (if no communication occurs), one (typically), or multiple associated protocol diagram (if the overall interaction cannot be represented in one diagram). Typically, a protocol diagram includes the expected interactions underlying the be-

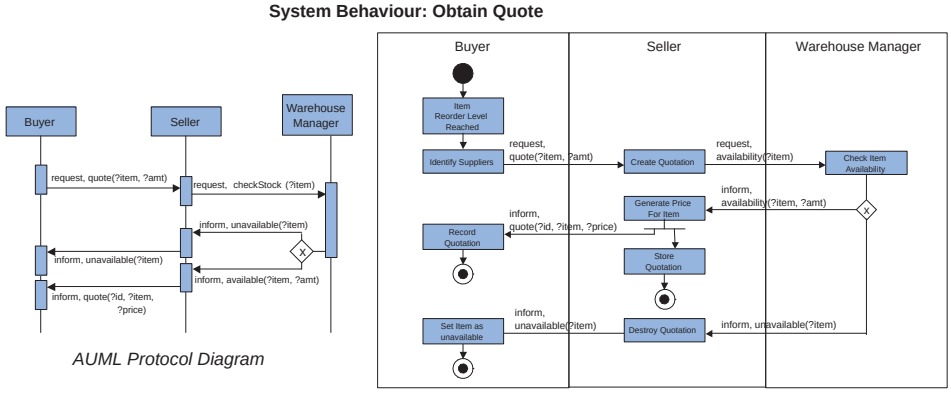

Figure 5a IM Diagram

Figure 5b ActM Diagram

Fig. 5. Supply Chain Scenario IM and ActM: ObtainQuote.

haviour (normally the successful case); and all (known) variant interactions (i.e. the unsuccessful cases). The example, shown in figure 5a, shows an interaction scenario in which the Retailer.Buyer requests a quote from the Supplier.Seller agent by sending a message e.g. `request, quote(?item, ?amt)`. This agent, in turn carries out a stock check by sending a message to the Supplier.WarehouseManager agent. Two variants ensue: if there is sufficient stock, then the Seller generates and returns a quote to the Buyer, otherwise, the Seller informs the buyer that the order cannot be met.

4. **Expand on IM with Activity Model.** *The Activity Model (AM)* identifies the set of activities carried out by agents in order to realise a given system behaviour [16]. This model employs customised UML Activity diagrams, where: boxes represent individual activities; associations place an ordering on the performance of the activities; and labelled swim lanes are introduced to associate agents with activities. Associations between activities in different swim lanes is interpreted as interaction between the agents (i.e. messages). Parallel activities can be represented via similar extensions to those used for Interaction diagrams. Figure 5b presents the Activity diagram for the Obtain Quote system behaviour. Here, the succesful completion the behaviour requires the completion of a number of activities (e.g. Item Reorder Level Reached, and Identify Suppliers, etc).

In summary, the Design phase builds on a number of agile techniques to provide an incremental, iterative design process, that supports customer involvement, improves flexibility, and facilitates continuous analysis and feedback in terms of behaviour, error and success. It provides the ability to adapt to changing situations and allows for the developer to learn and improve on the previous release through continuous iteration and improvement of the design. It facilitates prioritising of tasks and efficient allocation of resources (i.e. developers). Finally, it can be seamlessly incorporated into any agent development framework.

3.2 Test-Driven Implementation

Designed to help manage and control agent development, the Test-Driven Implementation (TDI) phase employs agile techniques to support the creation, testing and implementation of agents. For simplicity, the TDI phase is divided into 5 managable steps (see Figure 6) that can be implemented easily on a variety of frameworks. The following description combines our worked example from Supply Chain Management, which is implemented using Agent Factory (AF)[15][6]

At the centre of the TDI approach are the concepts of *Test Agents* and *Application Agents Under Test (AAUT)*. The Test Agent (TA) is an agent whose implementation encodes a set of *Test Cases* that must be satisfied by the corresponding AAUT (the agent whose behaviour is under test). Each Test Case represents a set of tests that are carried out on the AAUT in order to evaluate whether it satisfies a given requirement. Specifically, these tests check that the AAUT responds in an expected way as is specified by the corresponding aspects of the Interaction Model. A Test Case may contain one or more agent tests. It may require many test cases to establish that a requirement has been fully satisfied.

Individual TAs are executed via a purpose-built test environment (*Test Suite*) that provides an agile-perspective system architecture comprised of *Test*, *Implementation*, and *Integration* platforms (The TestSuite Application Structure is illustrated in Figure 6). Test Platform handles the creation and validation of TAs. The Implementation platform handles both the creation of application agents, and the implementation of tests against those agents. The Integration platform handles the integration of application agents for deployment in a MAS. Dividing the TDI process in this way enables the developer to accurately manage development and integration of tests and application code.

Agent Factory supports TestSuite through a combination of a purpose-built TestSuite Platform Service together with a *TestAgent Base Class* that provides support for coding TAs. Support is realised through a partial AFAPL agent program (*TestAgent*) which provides a generic test agent implementation that can be imported into the actual TAs. Specifically, it facilitates the addition of tests, and the indication of the failure or passing of tests. Support for visualisation of the debugging process is realised through the *SADAAM Monitor* - a Java-based GUI that displays the current status of the test - allowing developers to monitor and manage the progress of tests across platforms. It provides useful information such as: agent identifiers, test identifiers, number of iterations completed, any optional comments, and status of specific tests associated with the agent.

A key feature of SADAAM is the way in which our process utilises the TAs as building blocks for the implementation of the application agents. This is explained in more detail in step 2 below:

[6] AF (http://www.agentfactory.com) is a cohesive framework that delivers structured support for the implementation of intentional agents via the purpose-built Agent Factory Agent Programming Language (AFAPL) [14]).

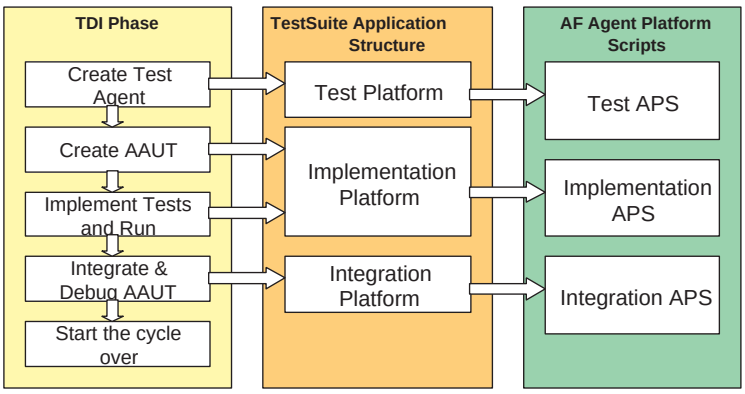

Fig. 6. How the TestSuite Application Structure facilitates TDI, and AF APS.

1. **Create Test Agent (TA)** The TDI process begins with the development of Test Agents to support testing of the application agents. Creating a Test Agent (TA) involves a number of steps. It commences with a review of the Design phase to determine which AAUT's to develop first. Our example involves two agents: the TransportManager and Hauler. Next, the requirements for each TA are determined through the required Test Cases for each given scenario. Each TA is given the same name as the agent it is replacing (in the interaction scenario), but is prefixed by the word "test". In the example, the TransportManager Agent informs the Hauler Agent of required delivery. So, testing the Hauler Agent involves creating a *testHauler* TA that implements the functionality of the Hauler Agent e.g. testHauler receives a message e.g. `inform, assignDelivery(?delID)` and performs the appropriate test. Likewise, a testTransportManager Agent is created to test the behaviour of the TransportManager Agent. The TAs are then developed based on the determined requirements. The worked example is implemented using AFAPL as follows:
   (a) Each TA is created in AFAPL format. Specifically, all agents are created with the `.afapl2` file extension (e.g. `testHauler.afapl2`).
   (b) Import TestAgent base class to provide necessary test methods.
   (c) Include belief/commitment criteria along with overridden tests (as illustrated in the testHauler.afapl2 code snippet below).

   Each TA is subsequently compiled to a .agent file (e.g. `testHauler.agent`), and an AF platform configuration file (CFG) (e.g. `sadaam.cfg`) is then created that includes relevant configuration information used by each agent platform (e.g. TestSuite platform services and GUI). Next, a test deployment is specified via an Agent Platform Script file (aps). This file, conveniently named `test.aps`, stores details of the TAs. Specification involves declaring the TA and its association with other (test) agents. In our example, testHauler and testTransportManager agents are created and then

associated with each other, enabling interaction between the agents. Specifically, the APS file specifies the creation of the TAs (e.g. `CREATE AGENT testHauler testHauler.agent`), whilst the add belief statement (`ADD_BELIEF testTransportManager ALWAYS(BELIEF(( testHauler, addresses(local:test.ucd.ie))))`) links testTransportManager to testHauler (see test.aps code snippet below). This mapping of TAs to one another allows for easy validation of TAs prior to implementation with the AAUT. Validation verifies that the TAs are collaborating and functioning correctly; underpinning our test-driven methodology through the realization of a solid foundation to run tests, and allowing us to proceed with confidence to the next step of the IDP. Finally, the test deployment is loaded onto a configured agent platform and the TAs are executed. The outcome of individual tests are monitored via the TestSuite GUI. To summarize, this section identified, created, and tested TAs ready for integration with applications agents.

testHauler.afapl2

```
IMPORT agent.TestAgent;
BELIEF(testSuite(testSuite));
COMMIT(?self, ?now, BELIEF(true),
SEQ(AWAIT(BELIEF(testSuiteSetUpCompleted)),
FOREACH (BELIEF(TransportManager(?name, ?addr)),
ATTEMPT (inform(agentID(?name, ?addr), documentID(0)),
passTest(testSuite, sendDocumentID),
failTest(testSuite, sendDocumentID)))));
```

test.aps

```
CREATE_AGENT testHauler testHauler.agent
CREATE_AGENT testTransportManager testTransportManager.agent
ADD_ BELIEF testTransportManager
ALWAYS(BELIEF((testHauler, addresses(local:test.ucd.ie))))
```

2. **Create Application Agent under Test (AAUT)** AAUTs represent the entities that are responsible for deploying processes in the MAS. In SADAAM, programming and registering an AAUT follows a "framework specific" approach. Thus creating an AAUT in AFAPL involves:
   (a) *Review Design phase to determine application agent requirement:* e.g. this example requires: TransportManager and Hauler agents.
   (b) *Update Implementation APS file:* AAUTs are created and registered in an Implementation APS file conveniently named `implementation.aps`.
   (c) *Associate AAUT with TA and integrate:* The `implementation.aps` file associates each AAUT with its relevant TA for testing (e.g. associate testHauler with Hauler); and it stores details of both AAUTs and TAs.
3. **Implement tests on Application Agent(s).** This step tests the AAUT prior to implementation into MAS, and involves:
   (a) Implementing tests to establish whether the AAUT is running correctly, or if some debugging is required. The TestSuite GUI displays both the

AAUTs and TAs along with the status of tests that reside within them (indicated in the Status column).

(b) Review, Refactor and start the cycle over. A review of the code my reveal potential refactoring, and or a need to further improve the model. At this point the developer may choose to start the cycle over or move forward to the next step: Integrate and Debug Application Agents.

4. **Integrate and Debug Application Agent(s).** Integration and debugging of Application Agent(s) for deployment into the MAS involves:
   (a) *Create/Revise Integration APS file:* Tested Application Agents are registered (ready for MAS deployment) in an Integration file `integration.aps`.
   (b) *Integrate application agents ready for deployment:* Integration is achieved using standard AFAPL similar to that for APS files described above.
   (c) *Run Integration tests (as necessary):* This should establish whether the application is running correctly, or if some further debugging is required.
5. **Review, Refactor, and Start the cycle over.** Review, Refactor and Start the cycle over as necessary. A review of the code my reveal potential refactoring, and or a need to further improve the model. At this point the developer may choose to start the cycle over or move forward to the next phase of the TDI: Release and Review.

3.3 Definition and Structure of the SADAAM Test Agent

Within SADAAM, tests can be derived directly from Interaction Models[7], and formally defined using a Test Definition Model (TDM). When writing a test we are looking to check that the application code is functioning correctly, and this test definition forms an important part of the development process. Currently, TAs focus on communication scenarios (the sending and receiving of messages), and these models help formally define tests that facilitate expected and unexpected behaviour. Typically, a TDM will define a successful case (i.e. expected

[7] SADAAM is currently working on automating this process; and has already successfully derived TAs from Agent-UML Protocol Diagrams based on earlier work [24].

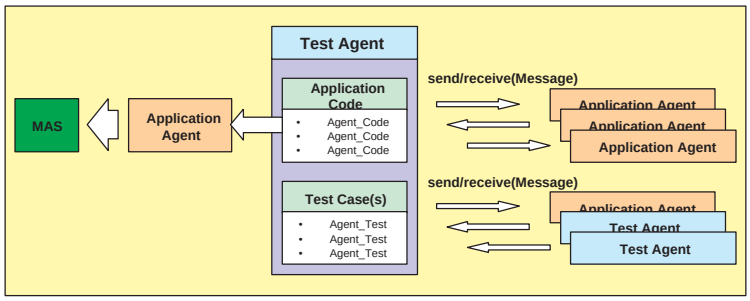

Fig. 7. SADAAM Application Agent Testing.

interactions underlying the behaviour); and it will also include a set of unsuccessful cases (to test various alternative scenarios representing occurrence that deviate from the expected scenario). Results of these test definitions are formally defined in the TestSuite.

Figure 7 illustrates the structure of a typical SADAAM Test Agent. Each TA includes one or more test-cases associated with the AAUT. Each test-case may contain one or more tests, and may facilitate numerous scenarios (expected and alternative). Tests may be deployed to evaluate communication between 2 or more agents [8], facilitating the testing of messages sent and received. Each TA allows an AAUT to be tested while communicating with multiple agents simultaneously. Each TA also contains application code that forms the initial building blocks of the deployable Application Agent.

This structure serves three main purposes. Firstly, it allows the developer to validate the TA. Secondly, it determines that the AAUT is functioning correctly prior to deployment in a MAS. Thirdly, it provides the building blocks for an initial implementation of the deployable Application Agent (each TA represents an initial implementation of the AAUT). This approach is a direct improvement on other approaches because it allows code to be reused for the development of application agents, thus driving the development [9]. Additionally, each TA can hold any number of test-cases, representing a variety of scenarios and agent tests. Furthermore, agents can be tested in isolation or as part of a MAS; and tests are neatly packaged because a TA houses all the tests relating to the specific AAUT. In summary, the SADAAM TA improves flexibility and is more productive because it allows TA code to be reused not only for subsequent development, and maintenance; but more importantly, for development of deployable agents.

3.4 Release and Review

This phase involves delivering working code to the customer for review. Its main purpose is to release the application in a live environment, and ultimately provide closure to the project. During this phase, the customer is given an opportunity to test and review each new piece of working software to determine whether it meets their requirements, before it is signed off as complete. This process may highlight required enhancements, potential refactoring, and / or a need to further improve the application; and is a continuous process that is maintained until the application fulfils the customers requirements. Advantages of this approach include an improvement in flexibility and efficiency, and increased responsiveness to changes in requirements. Upon completion of the Implementation Phase, a release is created. Releases may be internal, for development purposes only, or external, for evaluation by the client.

1. Release: To create a release, all references to the test suites are removed from the agent code; and an initial deployment is created, possibly spanning N agent platforms. 2. Review: Finally, a comprehensive review is carried out after each release.

[8] A TA can be run against one or more: AAUT, TAs, or a combination of the two.
[9] SADAAM is currently working on automating this process.

3.5 Refactoring and Enhancement

Refactoring and Enhancement is not strictly a phase; instead it represents a continuous process that involves applying improvements and enhancements to the finished code. Typically, refactoring entails applying changes to the internal structure of the application software to make it more efficient and easier to comprehend, without altering the external behaviour. This process may necessitate: removal of duplicated code; the simplification of complex logic; and clarifying of ambiguous code. This continuous analysis of code helps enhance the design and implementation of the application (refactoring in small steps helps prevent the introduction of defects). Enhancements include new or improved functionality that may further improve the application.

This process, along with the review, highlights outstanding issues and motivates any required enhancements to the process.

4 Conclusions

Primarily, SADAAM presents an agile methodology for the development of agents into multi-agent systems. In particular, by linking AOSE practices (e.g. [8] [21] [29]) to techniques derived from a variety of agile development methods (see for example [1]), it is possible to combine the benefits of the most useful and established techniques from both fields; thereby promoting the iterative, incremental, testing and development of agent-based systems; and injecting flexibility into the development cycle. This approach significantly increases productivity and reduces time to benefits while facilitating adaptive, empirical systems development. Other potential competitive advantages of this approach include: stronger and more efficient mechanisms delivered through intelligent processes, and reductions in development time and costs, especially in dynamic environments.

This paper demonstrated how SADAAM can be employed to develop, test and implement an agent-based application; and included an evaluation based on a simplified Supply Chain scenario consisting of a Seller, Retailer and Haulage Firm.

SADAAM's approach to agent development focuses on delivery of working code rather than documentation and detailed upfront design, and systematically guides and supports developers through the various stages of system development. SADAAM provides a number of contributions. In particular, it improves on other approaches because SADAAM Test Agents consists of a real implementation of the AAUT. Importantly, these Test Agents form the initial building blocks of the deployable application agent, and TA code is re-used for application agent development, promoting code-reuse. This approach is much more economical, it improves flexibility, and is more productive, because it allows test agent code to be reused not only for subsequent development, and maintenance; but more importantly, for development of deployable application agents. Secondly, Test Agents are able to communicate with one or many Application Agents and /

or other Test Agents (Conversely, Mock Agents communicate with just one agent, the AUT), allowing for an agent to perform multiple roles; and multiple agents can be tested simultaneously. Thirdly, increased flexibility facilitates a better allocation of resources, and therefore requires less initial investment. Furthermore, supporting development of highest-priority requirements first, with early delivery of quality-valued working software, is much more efficient than the serial approach (where the entire system is often developed upfront), and provides a better rate of return on investment. Use of these simple tools and modelling techniques is intuitive because you are creating a common understanding of the requirements and enabling the agile modelling practice to be performed by any number of developers. Additionally, this approach allows the development team to be assigned to specific tasks depending on a variety of criteria including: time scale, skills, or the priority of requirements. Flexible allocation of resources provides greater control over developing whatever is required in the time allocated to build; this level of control maximises developer investment because it allows the development team to retain control over how they invest their time and effort, and allows for the most efficient use of their resources. Thus, SADAAM allows you to develop software that is both high quality and high-value.

Finally, SADAAM is an agile methodology that supports the development and deployment of agent-oriented applications. It fulfills the requirement for an easy-to-use methodology that provides the more rapid development and implementation of agents into agent based systems; it advocates minimal documentation; provides a cohesive repertoire of interoperable techniques that support agile development; and provides a flexible methodology that improves efficiency and time to market.

References

1. Agile Alliance., "What is Agile Software Development?", *web: http://www.agilealliance.org/*
2. AGILE Manifesto available at http://agilemanifesto.org
3. Ambler, S., "Agile Modeling: The Official Agile Modelling (AM) Site." *Agile Modelling Home Page: http://www.agilemodeling.com/*
4. Amor, M., Fuentes, L. and Vallecillo, A., "Bridging the Gap between AO Design and Implementation Using MDA" , *AOSE 2004. LNCS 3382, pp 93-108.*
5. Bauer, B., Mueller, J., and Odell, J., "Agent UML: A Formalisation for Specifying Multi-Agent Software Systems.", *In P. Ciancarini and M. Wooldridge, ed, 1st Int. Workshop on Agent-Oriented Software Eng. (AOSE-2000)*, Limerick, Eire, 2000.
6. Beck, K., "Extreme Programming Explained: Embrace Change.", *AW Pub. 1999.*
7. Beck, K., "Test-Driven Development by Example.", *Addison Wesley, 2003.*
8. Bergenti, F. and Poggi, A., "Exploiting UML in the design of Multi-Agent Systems.", *In Proc. of the ECOOP Workshop on Engineering Societies in the Agents' World 2000 (ESAW'00)*, pages 96-103, 2000.
9. Boehm, B. and Turner, R., "Balancing Agility and Discipline: A Guide for the Perplexed.", *Boston, MA: Addison-Wesley, 2004*, ISBN 0-321-18612-5, pp 165-194.
10. Bresciani, P., Giorgini, P., Giunchiglia, F. Mylopoulos, J. and Perini, A., "TROPOS: An Agent-Oriented Software Development Methodology.", *JAAMAS, 2003.*

11. Cernuzzi, L., Cossentino, M., Zambonelli, F., Process Models for Agent-based Development, *Journal. Eng. Applications of AI*, 18(2), 2005.
12. Chella, A. Cossentino, M., Sabatucci, L., and Seidita, V., "Agile PASSI: An Agile Process for Designing Agents", *International Journal of Computer Science and Eng. Special Issue on "Software Eng. for Multi-Agent Systems"*, May 2006
13. Coelho, R., Kulesza, U., von Staa, A., and Lucena, C., "Unit testing in multi-agent systems using mock agents and aspects, *Procs of the 2006 Int. workshop on Software engineering for large-scale multi-agent systems, Shanghai, China*
14. Collier, R. W. "Agent Factory: A Framework for the Engineering of Agent-Oriented Applications.", *PhD thesis*, Dept. of Computer Science, UCD, Dublin, 2002.
15. Collier, R.W., O'Hare, G.M.P., Lowen, T.D. and Rooney, C.F.B., "Beyond Prototyping in the Factory of Agents.", *3rd Int. Central and Eastern European Conf. on Multi-Agent Systems, CEEMAS 2003*, Prague, Czech Republic, 16-18th June, Lecture Notes in Computer Science (LNCS), Springer Verlag, 2003.
16. Collier, R. W., Rooney, C. and O'Hare, G. M. P., "A UML-based Software Engineering Methodology for Agent Factory.", *Proceeding of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE-2004)*, Banff, Alberta, Canada, 20-25th June, 2004.
17. Fowler, M., "Refactoring: Improving the Design of Existing Code.", Addison Wesley Longman, 1999.
18. P. Giorgini, eds, "Agent-Oriented Methodologies", *IdeaGroup*, B. Henderson-Sellers 2005.
19. Huget, M-P, Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., Zhu, Hong, Interaction Diagram Specification, FIPA AUML Website, http://www.auml.org
20. Iglesias, C., Garijo, M., Gonzalez, J., Velasco, J., "Analysis and Design of Multi-Agent Systems using MAS-CommonKADS.", Springer,LNCS 1365, p489, 2004
21. Jennings, N.R., "Building Complex Software Systems: The Case for an Agent-Oriented Approach.", *Communications of the ACM*, 2001.
22. Jennings, N.R. and Wooldridge, M.; "Agent-Oriented Software Engineering." *in Handbook of Agent Technology, (ed. J. Bradshaw)*, AAAI/MIT Press, 2000.
23. Knublauch, H., Koeth, H, and Rose, T., "Agile Development of a Multi-Agent System: An Extreme Programming Case Study.", *Appears in: Proc of the 3rd Int. Conf. On Extreme Programming and Agile Processes in Software Engineering (XP2002), Alghero, Sardinia, Italy.*
24. Rooney, C F.B., Collier, R.W., O'Hare, G.P., "VIPER: VIsual Protocol EditoR", *in 6th Int. Conf. on Coordination Languages and Models (Coordination 2004)*, Pisa, February 24-27, 2004.
25. Schwaber, K., and Beedle, M., "Agile Software Development with Scrum.", *Prentice Hall, 2001.*
26. Sommerville, I., "Software Engineering.", *Addison-Wesley 6th Ed., 2000.*
27. Tveit, A., "A survey of Agent-Oriented Software Engineering." *In Proceedings of the First NTNU Computer Science Graduate Student Conference*, Norwegian University of Science and Technology, May 2001.
28. Wood, M.F. and DeLoach, S.A., "An Overview of Multi-Agent Systems Engineering Methodology." *AOSE-2000, 1st Int. Workshop on AOSE*, Limerick, Eire, 2000.
29. Zambonelli, F., Jennings, N.R., Omicini, A., and Wooldridge, M., "Coordination of Internet Agents: Models, Technologies and Applications", *Agent-Oriented Software Engineering for Internet Applications*, Springer-verlag, 2000