A Product-Line Approach to Promote Asset Reuse in Multi-agent Systems

Josh Dehlinger[1] and Robyn R. Lutz[1,2]

[1] Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, USA
dehlinge@cs.iastate.edu
http://www.cs.iastate.edu/~dehlinge
[2] Jet Propulsion Laboratory / Caltech
rlutz@cs.iastate.edu
http://www.cs.iastate.edu/~rlutz

Abstract. Software reuse technologies have been a driving force in significantly reducing both the time and cost of software specification, development, maintenance and evolution. However, the dynamic nature of highly autonomous agents in distributed systems is difficult to specify with existing requirements analysis and specification techniques. This paper offers an approach for open, agent-based distributed software systems to capture requirements specifications in such a way that they can be easily reused during the initial requirements phase as well as later if the software needs to be updated. The contribution of this paper is to provide a reusable requirements specification pattern to capture the dynamically changing design configurations of agents and reuse them for future similar systems. This is achieved by adopting a product-line approach for agent-based software engineering. We motivate and illustrate this work through a specific application, a phased deployment of an agent-based, distributed microsatellite constellation.

1 Introduction

Software reuse technologies have been a driving force in significantly reducing both the time and cost of software requirements specification, development, maintenance and evolution. Industry's continuous demand for shorter software development cycles and lower software costs encourages software development methodologies to exploit software reuse principles whenever possible.

Agent-oriented, software-based approaches have provided powerful and natural high-level abstractions in which software developers can understand, model and develop complex, distributed systems [5]. Yet, the realization of agent-oriented software development partially depends upon whether agent-based software systems can achieve reductions in development time and cost similar to other reuse-conscious software development methods such as object-oriented design, service-oriented architectures and component based systems.

In recent years, several agent-oriented software engineering (AOSE) methodologies have been proposed for various agent-based application domains. The Gaia methodology [28], in particular, offers a comprehensive analysis and design

A. Garcia et al. (Eds.): SELMAS 2005, LNCS 3914, pp. 161 – 178, 2006.

framework based on organizational abstractions by supplying schemas, models and diagrams to capture the requirements of an agent-based software system.

However, Gaia has two limitations. First, although Gaia provides a mechanism to allow the role of an agent to change dynamically, it is unclear how to document agent requirements specifications during the analysis and design phases when an agent must be updated to include new functionality. Second, the Gaia methodology fails to provide a mechanism by which the requirements specification templates developed during the analysis phase can be reused to be incorporated into the current system or to build a new, similar but slightly different system.

This paper offers an approach for open, agent-based distributed software systems to capture requirements specifications that can be easily reused during the initial requirements phase as well as later if the software needs to be updated. Our approach uses a product-line perspective to promote reuse in agent-based, software systems early in the development lifecycle so that software assets can be reused in the development lifecycle and during system evolution. We define a *product-line asset* as a software engineering output (including, but not limited to, architecture, reusable software components, domain models, requirements statements, documentation and specifications, and test cases) that forms, along with other product-line assets, the core for the development of a software product line [8]. We define *system evolution* as either the updating of an existing agent(s) in a deployed system or the inclusion of additional agents in the system.

The contribution of this paper is to provide a requirements specification pattern to capture the dynamically changing design configurations of agents and reuse the requirement specifications for future similar systems. This is achieved by adopting a product-line approach into AOSE. Requirements specifications reuse is the ability to easily use previously defined requirements specifications from an earlier system and apply them to a new, slightly different system.

The integration of product-line concepts into AOSE expands the techniques and tools available to developers of multi-agent systems. For example, software safety analysis tools [10, 18] and techniques [10, 12, 16] have been developed by the authors to assure developers that the reuse of requirements specifications is safe and will not compromise the system via incompatible features interacting in such a way as to cause unsafe conditions. We motivate and illustrate this work through a specific application, a phased deployment of an agent-based, distributed microsatellite constellation [19, 22]. A constellation is a group of semi- or fully autonomous satellites working together to fulfill complex mission objectives such as monitoring ocean levels or the spread of wildfires.

The remainder of the paper is organized as follows. Section 2 reviews related research in AOSE, product-line software engineering and a microsatellite application. Section 3 presents our approach to define the requirements specification of an agent-based, distributed system using the case study. Section 4 provides step-by-step guidance for documenting the requirements specifications of a distributed, multi-agent-based system using the requirements specification pattern presented in this work in a product-line-like way. Section 5 describes how to use the requirements specification detailed in Section 4 for reuse during system changes and updates. Finally, Section 6 provides concluding remarks and future research directions.

2 Related Work

This work is based on two different areas of software engineering: agent-oriented software engineering (AOSE) and software product-line engineering. This section discusses background information and related work in each of these areas as well as the application we use to illustrate our approach in this work.

2.1 Agent-Oriented Software Engineering

Agent-oriented software engineering (AOSE) [26] methodologies surfaced in the late-90's to provide tools and techniques for abstracting, modeling, analyzing and designing agent-based software systems early in the development lifecycle [21]. Different methodologies, such as Gaia [4, 27, 28], Tropos [2] and MaSE [11] for example, use different abstractions and models for agent-oriented software development. Although Tropos and MaSE are not the focus of this paper, an investigation of integrating a product-line approach into these AOSE methodologies is planned future work.

From its onset, one of the goals of AOSE has been to provide methodologies for reusing and maintaining agent-based software systems [23]. In spite of this goal, AOSE methodologies have failed to adequately capture the reuse potential, since many of the developed methodologies center on the development of specific software applications [13]. A few attempts, including [13] and [14], have been proposed for reuse in an agent-oriented development environment. However, in each case, reuse is positioned in the later stages of design and development. In [13], the Multi-Agent Application Engineering (MaAE) work exploits reuse during the design phase of a multi-agent software system. Likewise, [14] utilizes reuse principles from component-based development to reuse components from a previously developed agent-based component repository. The work described here differs from previous work in that we present an approach to capture the reuse potential of distributed, agent-based software systems in the requirements analysis and specification stage.

As in Gaia, our approach follows an early requirements engineering phase that focuses on analyzing the "characteristics to be exhibited and the goals to be achieved by the system-to-be" [27]. Our approach utilizes the output produced from the requirements engineering phase which may include goals and sub-goals, detailed requirements and partial requirements specifications [24].

2.2 Product-Line Engineering

We follow Northrop et al. in defining a software product line as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission [17]. The common, managed set of features shared by all members of a product line is called its commonalities. For example, a commonality of a planner role for a microsatellite would be that it should be able to know (i.e., read) its current position. The members of a particular product line may differ from each other via a set of allowed variabilities. In our application, an important variability is the *level of intelligence* of the agent or member. For example, a particular intelligence level of a planner role for a microsatellite would be that it has

the ability to command other microsatellites to move to another position. Additional examples of commonalities and variabilities are given in Figure 2. The benefits of the product-line approach come from the reuse of the commonalities in developing a new product-line member [20]. We define a member as a single instance or system of the product line. In the application used in this work, a member is a single microsatellite within the constellation.

Weiss and Lai defined the Family-Oriented Abstraction, Specification and Translation (FAST) software engineering model to analyze and design software product lines [25]. This model employs a two-phase software engineering approach: the domain engineering phase and the application engineering phase. The *domain engineering* phase defines the product line requirements specifications and design. The second phase, the *application engineering* phase, reuses these product-line assets (i.e., the product line requirement specifications and design) to develop the requirements and design of new product-line members.

Product-line requirements are often specified through a Commonality and Variability Analysis (CVA) [1, 25]. The CVA, developed during the domain engineering phase of FAST, provides a comprehensive definition of the product line requirements including the commonalities (i.e., requirements of the entire product line) and the variabilities (i.e., specific features not contained in every member of the product line) of the product line.

In this paper we use product-line techniques to transfer agent roles in the Gaia methodology into reusable assets. We demonstrate how requirements detailed in a CVA can be easily mapped into the Role Schema and the Role Variation Point Schema of the Gaia analysis phase using a product-line approach. Doing so helps link the requirements specification patterns for defining roles within the Gaia methodology to earlier requirements engineering assets. Our approach maintains consistency with the widely published Gaia methodology [28] of AOSE as well as the FAST methodology [25] of software product-line engineering.

2.3 Application

We illustrate how our approach can be used by applying it to portions of an agent-based implementation of a constellation, loosely based on the requirements for the TechSat21 (Technology Satellite of the 21st Century) [19]. TechSat21 was a proposed mission, originally scheduled to launch in January 2006 but cancelled in late 2003 with much of the software reused on a subsequent mission [6]. It was designed to explore the benefits of a distributed, cooperative approach to satellites employing agents [7, 22]. TechSat21 is a constellation (i.e., cluster) of context-aware microsatellites (weighing less than 100 kilograms). New microsatellites will be deployed to the constellation in multiple, planned phases with the new microsatellites potentially having additional capabilities not found in previously deployed microsatellites while sacrificing functionality found in other microsatellites [7, 22]. For example, after the initial cluster of microsatellites is deployed, some additional microsatellites may be deployed with extra functionality for the Cluster Allocation Planner role that the initially deployed microsatellites will lack.

Within the constellation, each microsatellite must know its context in order to meet certain functional or non-functional requirements placed upon the constellation. For

example, a context-based, functional requirement placed upon the constellation is to perform earth science (i.e., taking sensor readings, photographs, etc.). Thus, each microsatellite needs to know its context in relation to Earth. A context-based, non-functional requirement placed upon TechSat21 microsatellites is that each microsatellite must know its position in relation to others to avoid collisions. Similarly, microsatellites within the constellation must cooperate to meet mission requirements.

Schetter, Campbell and Surka have investigated several possible agent-based, organizational architectures for the TechSat21 constellation. They evaluated each agent-based, organizational architecture through a simulation tool to derive a multi-agent architecture that would offer good support for fault tolerance and upgradeability [19]. Separately, Chien et al. have similarly proposed a high degree of agent autonomy for a constellation of satellites. In their work, they demonstrate how continuous planning, model-based mode identification and reconfiguration and other artificial intelligence technologies can be used in a hybrid, multi-layer control architecture to facilitate a virtual spacecraft agent [7]. We partially use the agent specifications for the TechSat21 microsatellite constellation detailed in [7] and [19] for the requirements specification presented here as well as the notion of an agent possibly having different levels of intelligence described in [19].

3 Approach

To illustrate the adaptation of a product-line approach to multi-agent system development, we use the agent-based implementation of the microsatellite constellation (i.e., distributed system) detailed in [19]. Like them, we define an *agent* as a major onboard subsystem of a microsatellite or the microsatellite itself [9, 19].

3.1 Adopting Product-Line Concepts into the Gaia Methodology

The work presented here ties some of the analysis steps performed in Gaia to earlier requirements engineering outputs. Gaia, however, does not explicitly handle the requirements capturing and modeling of early requirements engineering. To address this, we link the Commonality and Variability Analysis (CVA) [1], performed within the domain engineering phase of the Family-Oriented Abstraction, Specification and Translation (FAST) product-line methodology (detailed in Section 2.2) [25] to the analysis and design of roles in a distributed, multi-agent system. Doing so, we are able to use a product-line-like approach to specify the requirements of a distributed, multi-agent system where differing intelligence is a design consideration. Note that the use of a CVA is not necessary. Developers may utilize other requirements modeling techniques such as goal-oriented [3] or feature-oriented [15] approaches. We discuss the advantages of using a CVA over these approaches in Section 4.1.

We first give an overview of how we integrate product-line concepts into multi-agent system development in order to build reusable patterns. Our approach, shown in Figure 1, partially encompasses three phases of multi-agent system development. Figure 1 illustrates how we incorporate elements of our approach into pieces of the Gaia methodology. The requirements collection and documentation phase, a part of FAST's domain engineering phase [25], partitions the requirements for the proposed

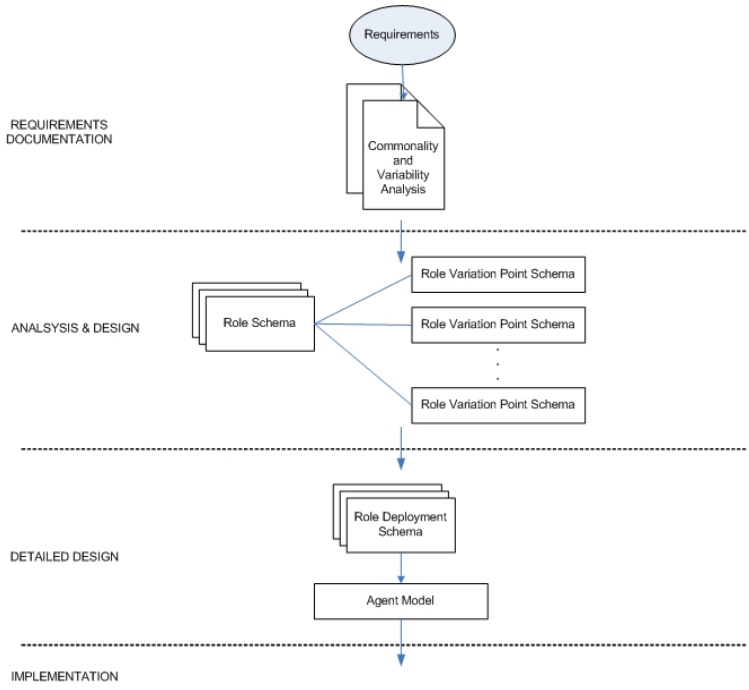

Fig. 1. An overview of the software engineering assets used in our Gaia-based product-line approach during requirements documentation, analysis and design and detailed design phases

system into commonalities and variabilities in the CVA. The analysis and design phase, equivalent to FAST's domain engineering phase [28] and loosely corresponding to Gaia's Preliminary Role Model and Role Model steps [30], entails documenting a role's requirements specifications within both the Role Schema and the Role Variation Point Schema. The Role Schema provides a description of the role and defines the variation points possible, described below, for the role. For each variation point, a Role Variation Point Schema details the requirement specifications for a role at a specific intelligence level.

We claim that the Role Schema and Role Variation Point Schema fall into the analysis and design phase, rather than solely the analysis phase, since the determination of variation points may influence, or be influenced by, the organizational architecture chosen (a product of the design phase in Gaia) for the multi-agent system being developed. For example, the decision to use an active/passive or master/slave type of variation point indicates that the architecture will display a top-down or a central coordination scheme. On the other hand, the intelligence-based design described below allows for a distributed or fully distributed coordination scheme. This process is described in Section 4.

In the detailed design phase of our approach, we first instantiate a role for a specific member of the distributed, multi-agent system by selecting the variation points (i.e., intelligence levels) it is capable in the Role Deployment Schema. It is in this phase that we are able to take advantage of the reuse principles inherent in the

FAST product-line methodology's application engineering phase [25]. Secondly, we define an agent using an Agent Model by declaring which roles, along with their variation points, it is responsible for fulfilling. We provide a full description of how the product-line concepts utilized in the analysis and design phase are taken advantage of to exploit the reuse potential in the detailed design phase in Section 5.

As in the Gaia methodology [27], we define the requirements specification of an agent's role, documented in the Role Variation Point Schema, in terms of the following characteristics: protocols, activities, permissions and responsibilities. *Protocols* define the way agents interact. *Activities* are the computations associated with the role that can be executed without interacting with other agents. *Permissions* are the information resource rights a role has to read, change and generate resources. *Responsibilities* define the functionality of a role and are divided into liveness properties and safety properties. Liveness properties refer to the "state of affairs that an agent must bring about, given certain environmental conditions" [27]. Safety properties refer to that subset of the non-functional requirements that the agent must maintain throughout the duration of the agent's life to prevent and handle hazards.

3.2 Using Variation Points in Multi-agent Systems

Product-line engineering uses *variation points* to capture the allowed differences amongst members belonging to the same family. We define the variation points for a specific role as the differing protocols, activities, permissions and responsibilities available to that role. Variation points typically stem from the grouping of the variabilities defined in the Commonality and Variability Analysis (CVA) documented as part of the output of the requirements engineering phase.

An important way to classify variation points for a satellite constellation is intelligence levels. In this work, we focus on these variation points. Here, we adopt the notion of variation points to model a multi-agent system as a product line. The variation point notion is important because it aids in capturing the different arrangements of agents and promotes reuse. For example, TechSat21 roles [19], ordered in terms of increasing intelligence levels, I4 through I1, display the following variation points:

- I4: receive/execute commands
- I3: local planning and receive/execute commands
- I2: local planning, interaction, partial cluster-knowledge and receive/execute commands
- I1: cluster-level planning, interaction, full cluster-knowledge and receive/execute commands

Thus, as a role is promoted to a higher intelligence level (from I3 to I2, for example) the configuration of the agent dynamically changes by incorporating additional protocols, activities, permissions and/or responsibilities. The reverse occurs when a role is demoted from a higher intelligence level to a lower intelligence level (from I2 to I3, for example). Using this construct, our notion of *an agent has one or more roles where each role may have one or more variation points*.

The variation points will initially be fixed upon deployment based upon the software and hardware facilities available as well as the role's goal. At deployment a

default intelligence level for each role is set. During execution, a role may change its variation point (i.e., intelligence level) based upon its internal state, commands from external sources or the environment. Alternatively, within a distributed, agent-based system, it is not likely that the same set of intelligence levels will be included in any given role throughout the entire distributed system [19]. Thus, from a product-line engineering perspective, we can view the set of roles containing different role/variation point combinations as a product line. The set of roles and dynamic variation points an agent contains is its *configuration*.

The variation points in the constellation will not be universal to all agent-based, distributed systems. Variation points are particular to each application and, indeed, particular to each role. For example, other variation points could include active, passive; hot-spare, cold-spare; etc. For every variation point identified, we associate a *binding time* which defines the time at which the variation point could be assumed by a role. Potential binding times include specification-time, configuration-time and run-time. In the case of the constellation, the binding time for all the variation points described here is at run-time. The actual decision as to which features to group together and how to classify each variation point (e.g., I1, I2, etc., versus hot-spare, warm-spare, cold-spare) is likely domain and/or application specific and is not covered in this work. Rather, we assume that domain experts group the variabilities listed into variation points so that they can be used during the analysis phase of Gaia.

Variation points are added with the Gaia characteristics of a role [27]. This allows us to leverage a product-line-like perspective to maximize reuse among software products that share a great many similarities amongst each other and differ by only a few variations. For example, it might be the case that all the microsatellites in a constellation responsible for monitoring volcano eruptions will be very similar while other microsatellites in the same constellation responsible for capturing Earth atmosphere pictures will greatly differ from the previously mentioned microsatellites but will be similar amongst each other.

Identifying the variation points to which a role may dynamically switch allows us to classify at which variation points the protocols, activities, permissions and/or responsibilities are introduced to the role. Partitioning the requirements specifications (i.e., the protocols, activities, permissions and responsibilities) of an agent in this manner will allow us to reuse the requirement specifications for future systems. Thus, future agents within a domain such as Earth-orbiting microsatellites can more readily utilize assets that have been specified in such a way. These future systems employ roles comprising some of the variation points previously defined as well as new capabilities not found in any of the previous systems. Section 5 gives a more complete description of how requirements specifications can be reused.

Lastly, we here introduce identification numbers to all schemas for traceability, organization and management purposes. The schemas serve as a requirements specification pattern in which requirements can be defined and documented.

4 Documenting the Requirements Specifications Using a Product-Line Approach

This section describes the documentation of requirements, analysis and design, and the detailed design phases, illustrated in Figure 1, incorporating a product-line

approach into multi-agent system development. For each phase, we describe the documentation process and how each document will later contribute to the ease of reuse, discussed in Section 5.

4.1 Requirements Documentation Phase

In the development of a software product line, requirements are collected and then documented in a Commonality and Variability Analysis (CVA) [1, 25]. Using the CVA, requirements can be further refined and detailed requirements can be derived during the analysis and design phases so that requirements specifications can be documented. An excerpt of a CVA detailing the commonality and variability requirements for the constellation is shown in Figure 2.

A developed and documented CVA during the requirements collection phase may give developers an insight into what roles might be appropriate for the multi-agent system to be developed. In terms of multi-agent system development, a CVA may assist in the identification of possible roles since it partitions those requirements that will be found in every future instantiation of a particular role from those requirements that will only be found in some instantiations of a particular role. That is, only the commonality should be investigated for potential agent roles since the variability can be thought of as features that will not be found in every role.

Commonalities

C_1. An agent capable of performing cluster allocation planning shall be able to receive/execute commands from other known members of the constellation at all times (i.e., during times of degraded system functionality).

C_2. An agent capable of performing cluster allocation planning shall be able to know its current position and velocity increment.

C_3. An agent capable of performing cluster allocation planning shall be able to change its current position and velocity increment.

C_4. An agent capable of performing cluster allocation planning shall be able to move to a new position.

Variabilities

V_1. An agent capable of performing cluster allocation planning may be able to have partial cluster knowledge.

V_2. An agent capable of performing cluster allocation planning may be able to have full cluster knowledge.

V_3. An agent capable of performing cluster allocation planning may be able to receive and accept change in velocity bids from other members during cluster reconfiguration.

V_4. An agent capable of performing cluster allocation planning may be able to issue a request to members of the cluster for change in velocity bids.

V_5. An agent capable of performing cluster allocation planning may be able to issue a move to new position message to members of the cluster during cluster reconfiguration.

V_6. An agent capable of performing cluster allocation planning may be able to receive a velocity increment calculation from other members of the cluster.

V_7. An agent capable of performing cluster allocation planning may be responsible to optimize the fuel use of all members of the cluster.

Fig. 2. A sample Commonality and Variability Analysis of the requirements for the Cluster Allocation Planner role

The actual identification of appropriate roles for a multi-agent system is not discussed here. Gaia proposes to identify roles through an inspection of the problem (via the division of a system into organizations and sub-organizations) [28]. Rather, we only claim that documenting a multi-agent system's requirements in a CVA may aid in confirming the role definition and help in the preliminary role model(s).
The variabilities of the CVA will help define the variation points of the product-line, multi-agent system. Partitioning the variabilities into similar groups (e.g., by similar required intelligence level) provides the initial requirements for the variation points of a system. For example, from Figure 2 we can assign variabilities V_1, V_3 and V_4 as belonging to intelligence level I2 since each indicates that it would at least require the intelligence level of I2 to occur.

Alternative approaches to the CVA in documenting product-line requirements and performing variability analysis include the goal-oriented [3] or the feature-oriented [15] approach. Alternatively, the use of domain or application expertise may also suffice in this process. This work exclusively used the CVA as the medium for variability documentation and analysis because of our use of the FAST methodology (in which a CVA is exclusively utilized to document and analyze variabilities). In terms of reuse, CVA is superior to either goal-oriented or feature-oriented approaches since it clearly defines those requirements that will be found in every member of a product line (i.e., commonalities) and those requirements that will only be found in a subset of the members of a product line (i.e., variabilities).

4.2 Analysis and Design Phase

Requirements specifications are documented in two schemas. The Role Schema, shown in Figure 3, defines a role and the variation points that the role can assume during its lifetime (e.g., whether it only implements the assignments it receives or it can also assign positions). The Role Variation Point Schema, shown in Figure 4, captures the requirements of a role variation point's capabilities. Both schemas are slightly modified adaptations of Gaia's Role Schema [28]. The Role Schema and the Role Variation Point Schema are both needed to capture the different levels of intelligence possible in a role throughout a large, distributed, multi-agent system.

Role Schema: Cluster Allocation Planner	**Schema ID:** F32
Description: Assigns a new cluster configuration by assigning new microsatellite positions within the cluster. This is done to equalize fuel use across the cluster.	
Variation Points: I4: receive/execute commands [F32-I4] I3: local planning and receive/execute commands [F32-I3] I2: local planning, interaction, partial cluster-knowledge and receive / execute commands [F32-I2] I1: cluster-level planning, interaction, full cluster-knowledge and receive/execute commands [F32-I1]	
Binding Times: All binding time for the variation points are at run-time.	

Fig. 3. Sample Role Schema for the Cluster Allocation Planner role of TechSat21

Role Schema: Cluster Allocation Planner **Schema ID:** F32-I1

Variation Point: I1

Description:

Assigns a new cluster configuration by assigning new microsatellite positions within the cluster. This is done to equalize fuel use across the cluster. With the I1 intelligence level, it is able to send cluster assignments to other microsatellites (i.e., spacecraft level agents) in order to arrange a new cluster configuration. This may occur when a new microsatellite is added or in the case of a failure of a microsatellite.

Protocols and Activities:

CalculateDeltaV, UpdateClusterInformation, MoveNewPos, DeOrbit, AssignCluster, AcceptDeltaVBids, RequestDeltaVBids, SendMoveNewPosMsg, SendDeOrbitMsg

Permissions:

Reads -

position	// current microsatellite position
velocityIncrement	// current microsatellite velocity increment
supplied *microsatelliteID*	// microsatellite identification number
supplied *velocityIncrment*	// microsatellite velocity increment

Changes -

position	// current microsatellite position
velocityIncrement	// current microsatellite velocity increment

Generates -

newPositionList	// new position list to assign to the // microsatellites within the cluster

Responsibilities:

Liveness -

Optimize the fuel use across the cluster.

Safety -

Prevent microsatellite collisions during a new cluster configuration.

Fig. 4. Sample Role Variation Point Schema for the I1 variation point of the Cluster Allocation Planner role of the Cluster Configuration

During the initial development of a distributed system (the product-line domain engineering phase of the Family-Oriented Abstraction, Specification and Translation (FAST) product-line methodology [25]), the focus must be primarily on identifying the overall requirement specifications of the system. It is later (during the product-line application engineering phase of FAST) that actual members of the distributed system can be instantiated with some or all of the requirements established earlier. We consider those initial requirement specifications in the Role Schema and the Role Variation Point Schema.

To capture the requirement specifications and document them in the two schemas, we use the following procedure:

1. Identify the roles within the system. Each role will constitute a new Role Schema to be created.
2. For each role, provide the role's name, a unique identification and a brief description of the role in the appropriate fields of the Role Schema. We partly follow the naming and numbering scheme from [19] for the Cluster Allocation Planner role depicted in Figure 3.

3. For each role, identify and define the differing intelligence levels that the role can adopt during all envisioned execution scenarios of the system. These differing intelligence levels will represent the variation points that the role can adopt. For each variation point, fill in the Variation Points section of the Role Schema by including the name, a brief description of the variation point and a reference identification number to the Role Variation Point Schema that gives the detailed requirements of the variation point (see Step 4a).
4. For each identified variation point (Step 3), create a new Role Variation Point Schema. For each Role Variation Point Schema:
   a. Document the name of the role to which the variation point corresponds as well as the name of the variation points in the appropriate sections of the Role Variation Point Schema. Indicate the variation point identification tag (corresponding to the variation point identification in Step 3) in the appropriate field in the Role Variation Point Schema.
   b. Identify the protocols, activities, permissions and responsibilities that are particular to only that variation point. That is, define the protocols, activities, permissions and responsibilities that are not found in any of the lower intelligence level variation points.
   c. Document and define the identified protocols, activities, permissions and responsibilities in the appropriate sections of the Role Variation Point. (Note, in accordance with the Gaia conventions, activities are distinguished from protocols by being underlined).

These steps result in a set of Role Schemas that have an associated set of Role Variation Point Schemas. Additionally, these steps conform to the domain engineering phase of product-line development [25]. Figure 3 illustrates a Role Schema example, and Figure 4 gives an example of a Role Variation Point Schema, both derived from the TechSat21 agent specifications given in [19].

4.3 Detailed Design Phase

Upon completion of the initial requirements analysis and development of an agent-based, distributed system, it will be necessary to utilize the derived requirements specifications to instantiate a number of members of the distributed system. During this initial deployment of distributed members, it is not necessary that all members be equipped with equal capabilities, intelligence or functionality. Since the prior steps have specified all the possible variation points of the roles in the schemas, we instantiate a new member (i.e., agent) to be added to the distributed system by specifying each new member to be deployed in the Role Deployment Schema. An example is shown in Figure 5.

Thus, the process is as follows:

1. Identify the roles that will constitute the member to be deployed.
2. For each role identified, create a new Role Deployment Schema and:
   a. Provide the role's name, unique system(s) identification and a brief description of the role specific to this deployment in the appropriate fields of the Role Deployment Schema. The system(s) unique identification, to be placed in the

System ID field, identifies the specific member(s) of the distributed system to be deployed that has the role configuration described in the particular Role Deployment Schema. For example, if members 2,3, 8-10 are to employ the Cluster Allocation Planning Agent in which only variation points I3 and I4 are possible, we denote this in the System(s) ID field of the Role Deployment Schema, shown in Figure 5, as 2,3, 8-10. This avoids repetitive manual overhead when designing new members to be deployed in the distributed system.

b. Identify all possible variation points that the role can assume during its lifetime. The set of possible variation points was previously established when the original Role Schema was developed for the particular role.
c. Identify the variation points in which the role will be deployed and specify it in the Role Deployment Schema. This variation point represents the default intelligence level at which the agent will most commonly operate during normal operations.

These steps produce a (set of) completed Role Deployment Schemas describing how different members of the distributed system to be deployed are instantiated.

Role Deployment Schema: Cluster Allocation Planner	**System(s) ID:** 2,3, 8-10
Description: A microsatellite member of the TechSat21 constellation that lacks the intelligence to globally assign new positions to other microsatellites within the cluster during a reconfiguration caused by a new microsatellite joining the cluster or a failure in one of the microsatellites. The sacrifice of this capability was chosen in favor of accommodating additional science instrumentation and software not found in microsatellites that allow I1 and I2 Cluster Allocation Planning Agent intelligence levels.	
Variation Points: I4: receive/execute commands [F32-I4] I3: local planning and receive/execute commands [F32-I3]	

Fig. 5. Sample Role Deployment Schema for the Cluster Allocation Planner role of the Cluster Configuration agent of a member of TechSat21

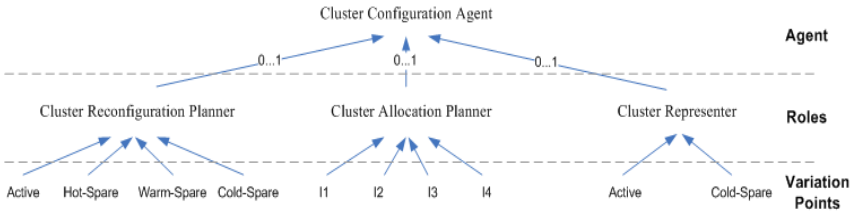

Fig. 6. A sample Agent Model for the Cluster Configuration Agent for a member of TechSat21

We illustrate how an Agent Model, expanded from the Agent Model of Gaia [4], can be derived in Figure 6. The Agent Model graphically illustrates the assignment of roles to agents as well as variation points to roles. The cardinality relationship between agent and role is indicated and all possible variation points are listed for each

role. At runtime, the designer annotates the actual cardinality and the specific possible variation points of an agent instance.

These steps conform to the application engineering phase of FAST [25] and produce the documentation shown in the detailed design phase shown in Figure 1. Documenting the requirements specifications in such a way allows easy reuse when instantiating actual systems. We detail how the documentation created in this section can easily be reused during both initial development and system evolution in the next section.

5 Multi-agent Requirements Specification Reuse

Requirements specification reuse is using previously defined requirements specifications from an earlier system and applying them to a new, slightly different system. Increasing the amount of requirements specification reuse for any given product will reduce the production time and cost of the software system [8].

Requirements specification reuse for agent-based, distributed systems is simplified in our approach by our use of variation points to handle the variabilities in similar software systems. Our approach takes advantage of how the requirement specifications for an agent's role were partitioned and documented in the Role Schema and Role Variation Point Schema based on their variation points. This section describes how the requirements specifications documentation detailed in Section 4 can be reused during the initial deployment of a distributed system as well as during system evolution. We define *system evolution* as the updating of an existing member(s) of a deployed system or the addition of new members to the system.

5.1 Requirements Specification Reuse During Initial Development

The members of a distributed system often will be heterogeneous in their functional capabilities yet mostly similar in structure. For example, some microsatellites of the constellation may have additional scientific imaging software while others may have additional cluster planning and reconfiguration software. Heterogeneity may also arise when resources (such as weight limits, memory size, etc.) are limited and different members of a distributed system must assume different roles. In the case of agent-based, distributed systems, members also may be heterogeneous in terms of their intelligence levels. For example, depending on the level of coordination (centralized, distributed or fully distributed, for example) among agents, not all agents must support roles at the highest level of intelligence. That is, not all agents may be capable of having full cluster-knowledge and/or being capable of making cluster-level decisions. For this reason, initially deployed members of a distributed system will likely contain a role that differs amongst other members in terms of which intelligence levels (i.e., variation points) it is capable of assuming. Several members of the distributed system will have the same role but at different levels of intelligence.

Requirements specification reuse can be exploited during the initial development and deployment of the members of a distributed system using the Role Deployment Schema, illustrated in Figure 5. Rather than repeatedly defining the requirements of a role for any given agent, the Role Deployment Schema allows us to define the intelligence levels it can assume. This reuse is possible because the requirements specifications for each of the levels of intelligence were documented in the Role

Variation Point Schemas, and because the agents of a distributed system will be similar. Thus, to document a particular role for several different heterogeneous members of a distributed system we must only indicate which variation points (i.e., previously defined intelligence levels) it can assume and give the reference number(s) to the Role Variation Point Schemas. After assigning variation points to an instance of a role and a role to an instance of an agent, an Agent Model can be used to illustrate an actual instance of an agent. We provide an example Agent Model in Figure 6.

5.2 Requirements Specification Reuse During System Evolution

Change is inevitable. Hardware failures or altered mission goals in a deployed distributed system typically necessitate software updates to one or more members. For example, a satellite of the constellation may have a malfunctioning planning and control module that could motivate operators to update that particular satellite's software to erase it and replace it with updated mission planning software. Alternatively, technology or mission goals after the initial deployment of a distributed system routinely evolve in such a way that future deployments of members joining the distributed system will require additional functionality (i.e., new features requiring new requirements). In the case of the satellite constellations, designers envision that new microsatellites will be deployed in multiple, planned phases [7, 22]. The new microsatellites will likely contain additional features not found in previously deployed microsatellites. Examples of the types of evolution the constellation may undergo include improved sensors, new scientific software, new communication devices, etc.

When the system evolves, new members may include additional functionality not previously defined in the requirements specifications. The requirements specification pattern detailed in Section 4 is extensible in that it can accommodate this kind of system evolution by being able to include a new set of requirements while still reusing the previously documented requirements.

If the system evolution is an update of a member of the distributed system where the update includes functionality previously defined in the requirements specifications (Role Schema and Role Variation Point Schema), it suffices to modify the Role Deployment Schema and, possibly, the Agent Model to reflect the update.

The addition of a new role within the distributed, agent-based system was described in Section 4.2. Briefly, we create a new Role Schema and a Role Variation Point Schema just as during the initial development of a multi-agent system. Following the creation of a Role Schema and a set of Role Variation Point Schemas, the process for the detailed design phase, outlined in Section 4.3, is used to instantiate a new agent with the new role.

The addition of a new variation point to an existing role, however, requires a modification to existing Role Schema documentation as well as the creation of a new Role Variation Point Schema. For example, the need to add a fifth intelligence level to an existing role would require such modification. For a new intelligence level desired for a particular role in future deployments of members of a distributed system, the following process suffices:

1. Create a new Role Variation Point Schema for the new intelligence level (i.e., variation point) giving the role's name, variation point's name and a unique variation point identifier in the appropriate fields.

2. Document the variation point indicating how the new variation point differs from previously defined variation points in the Description section.
3. Identify the protocols, activities, permissions and responsibilities that are particular to only that variation point. That is, define the protocols, activities, permissions and responsibilities that are not found in any of the lower intelligence level variation points and that are not found in any other variation points.
4. Document and define the identified protocols, activities, permissions and responsibilities in the appropriate sections of the Role Variation Point.
5. Update the Role Schema to which the new variation point corresponds, and add the new variation point, along with a description and schema reference identification, to the Variation Points section.

These steps will produce a new variation point for a role and the accompanying Role Variation Point Schema for future versions of members of the system.

6 Concluding Remarks

This paper incorporates a product-line approach into an agent-oriented software engineering methodology to support the reuse of the derived requirements specifications of an agent-based, distributed system. The requirements specification templates are constructed in such a way that varying dynamic software configureations of an agent are supported. The benefit is that the agent's configurations can then be reused during initial system development and during periods of system changes and updates. This can significantly reduce the software development time and cost.

To allow for the integration of product-line concepts into the Gaia methodology, we modified some of Gaia's schemas to better suit the concepts of software product-line engineering. In this paper we describe how a Role Schema, a Role Variation Point Schema, a Role Deployment Schema and an Agent Model can be created using a product-line approach. Using this approach assists in capturing the shifting configurations of agents/roles during the requirements analysis, design, detailed design and specification phases. Specifically, we describe how requirements specifications reuse can be achieved during initial system development, during periods of system changes and updates and during the addition of new members with previously defined functionality to a deployed, heterogeneous, distributed system.

Although this work was specifically intended for use in distributed multi-agent systems, this work may also be useful for distributed systems that are not necessarily agent-based such as sensor networks, grid-computing applications and peer-to-peer applications. Planned future work includes an application of this approach to a multi-agent system under development to evaluate the scalability of this approach.

Acknowledgements

This research was supported by the National Science Foundation under grants 0204139 and 0205588, and by the Iowa Space Grant Consortium.

References

1. Ardis, M. A. and Weiss, D. M., "Defining Families: The Commonality Analysis", *Proc. 19th Int'l Conf. on Software Engineering*, pp. 649-650, 1997.
2. Bresciani, P., Giorgini, P., Guinchiglia, F. and Perini, A., "TROPOS: An Agent-Oriented Software Development Methodology", *Journal of Autonomous Agents and Multi-Agent Systems*, 8(1):203-236, 2004.
3. Castro, J., Kolp, M. and Myopoulos, J. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project" *Information Systems* 27(6):365-389, 2002.
4. Cernuzzi, L., Juan, T., Sterling, L. and Zambonelli, F., "The Gaia Methodology: Basic Concepts and Extensions", *Methodologies and Software Engineering for Agent Systems.The Agent-Oriented Software Engineering Handbook Series: Multiagent Systems, Artificial Societies, and Simulated Organizations*, 11:69-88, 2004.
5. Chan, K. and Sterling, L., "Specifying Roles within Agent-Oriented Software Engineering", *Proc. 10th Asia-Pacific Software Engineering Conf.*, pp. 390-395, 2003.
6. Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castano, R., Davies, A., Mandl, D., Frye, S., Trout, B., D'Agostino, J., Shulman, S., Boyer, D., Hayden, S., Sweet, A. and Christina, S., "Lessons Learned from Autonomous Sciencecraft Experiment", *Proc. Autonomous Agents and Multi-Agent Systems Conf.*, 2005.
7. Chien, S., Sherwood, R., Rabideau, G., Castano, R., Davies, A., Burl, M., Knight, R., Stough, T., Roden, J., Zetocha, P., Wainwright, R., Klupar, P., Van Gaasbeck, J., Cappelaere, P. and Oswald, D., "The Techsat-21 Autonomous Space Science Agent", *Proc. 1st Int'l Conf. on Autonomous Agents*, pp. 570-577, 2002.
8. Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, Reading, MA, 2002.
9. Das, S., Krikorian, R. and Truszkowski, W., "Distributed Planning and Scheduling for Enhancing Spacecraft Autonomy", *Proc. 3rd Conf. on Autonomous Agents*, pp. 422-423, 1999.
10. Dehlinger, J. and Lutz, R. R., "PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool", *The Automated Software Engineering Journal*, to appear.
11. DeLoach, S. A., "The MaSE Methodology", *Methodologies and Software Engineering for Agent Systems-The Agent-Oriented Software Engineering Handbook Series: Multiagent Systems, Artificial Societies, and Simulated Organizations*, 11:107-125, 2004.
12. Feng, Q and Lutz, R. R., "Bi-Directional Safety Analysis of Product Lines", *Journal of Systems and Software*, to appear.
13. Girardi, R., "Reuse in Agent-based Application Development", *Proc. 1st Int'l Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, 2002.
14. Hara, H., Fujita, S. and Sugawara, K., "Reusable Software Components Based on an Agent Model", *Proc. Workshop on Parallel and Distributed Systems*, 2000.
15. Kang, K. C., Kim, S., Lee, J. and Lee, K., "Feature-Oriented Engineering of PBX Software for Adaptability and Reusability", *Software Practice and Experience*, 29(10):167-177, 1999.
16. Lutz, R. R., "Extending the Product Family Approach to Support Safe Reuse," *Journal of Systems and Software*, 53(3):207-217, 2000.
17. Northrop, L., "A Framework for Product Line Practice", *Software Engineering Institute*, http://www.sei.cmu.edu/productlines/framework.html, (current November 2005).
18. Padmanabhan, P. and Lutz, R. R., "Tool-Supported Verification of Product Line Requirements", *The Automated Software Engineering Journal*, 12(4):447-465, 2005.
19. Schetter, T., Campbell, M. and Surka, D., "Multiple Agent-Based Autonomy for Satellite Constellations", *Proc. 2nd Int'l Symposium on Agent Systems and Applications*, 2000.
20. Sommerville, I., *Software Engineering*, Addison-Wesley, Reading, MA, 2004.

21. Sutandiyo, W., Chhetri, M. B., Krishnaswamy, S. and Loke, S. W., "Experiences with Software Engineering of Mobile Agent Applications", *Proc. 2004 Australian Software Engineering Conf.*, pp. 339-349, 2004.
22. "TechSat21 - Space Missions Using Satellite Clusters", *Space Vehicles Factsheets*, http://www.cs.afrl.af.mil/ Factsheets/techsat21.html, (current February 2005).
23. Tveit, A., "A Survey of Agent-Oriented Software Engineering", *NTNU Computer Science Graduate Student Conf.*, 2001.
24. United States Department of Defense, "Draft DoD Software Technology Strategy", Office of the Director, Defense Research & Engineering, DRAFT: December 1991.
25. Weiss, D. M. and Lai, C. T. R., *Software Product-Line Engineering*, Addison-Wesley, Reading, MA, 1999.
26. Wooldridge, M. and Jennings, N. R., "Agent Theories, Architectures and Languages: A Survey", *Workshop on Agent Theories, Architecture and Languages*, pp. 1-32, 1995.
27. Wooldridge, M., Jennings, N. R. and Kinny, D., "The Gaia Methodology for Agent-Oriented Analysis and Design", *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285-312, 2000.
28. Zambonelli, F., Jennings, N. R. and Wooldridge, M., "Developing Multiagent Systems: The Gaia Methodology", *ACM Transactions on Software Engineering and Methodology*, 12(3):317-370, 2003.