

A Product-Line Requirements Approach to Safe Reuse in Multi-Agent Systems

Josh Dehlinger

Dept. of Computer Science, Iowa State University

226 Atanasoff Hall, Ames, IA 50011

+1 (515) 294-2735

dehlinge@cs.iastate.edu

Robyn R. Lutz

Dept. of Computer Science, Iowa State University

226 Atanasoff Hall, Ames, IA 50011

and Jet Propulsion Laboratory/Caltech

+1 (515) 294-3654

rlutz@cs.iastate.edu

ABSTRACT

The dynamic nature of highly autonomous agents within distributed systems is difficult to specify with existing requirements techniques. However, capturing the possibly shifting configurations of agents in the requirements specification is essential for safe reuse of agents. The contribution of this work is an extensible agent-oriented requirements specification pattern for distributed systems that supports safe reuse. We make two basic claims for this idea. First, by adopting a product-line-like approach, it exploits component reuse during system evolution. Second, the template allows ready integration with an existing tool-supported, safety analysis technique sensitive to dynamic variations within the components (i.e., agents) of a system. To illustrate these claims, we apply the requirements specification pattern and safety analysis to a real-world context-aware, distributed satellite system.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications – *methodologies*.

General Terms

Design

Keywords

Agent-based Development; Software Reuse; Product-Line Engineering; Software Engineering.

1. INTRODUCTION

Software reuse technologies have been a driving force in significantly reducing both the time and cost of software requirements analysis, development, maintenance and evolution.

The agent-oriented software based approach has provided a powerful and natural high-level abstraction in which software

developers can understand, model and develop complex, distributed systems. Yet, the realization of agent-oriented software development partially depends upon whether agent-based software systems can achieve reductions in development cost and time similar to other reuse-conscious software development methods.

The object-oriented approach provided mechanisms promoting reuse principles within the requirements and development of large software systems. More recently, the product-line engineering approach has been adopted to take advantage of core commonalities shared among a set of closely related software applications to maximize the potential for reuse [2].

Agent-oriented software engineering (AOSE) [18] methodologies surfaced in the mid-90's to provide tools and techniques for modeling, analyzing and designing agent-based software systems early in the development lifecycle [14]. From its onset, one of the main goals of AOSE was to provide methodologies for reusing and maintaining agent-based software systems [16]. 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 [8]. A few attempts, including [8] and [9], 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. Here, we present an approach to capture the reuse potential of distributed, agent-based software systems in the requirements stage.

This paper offers an approach for mission-critical, agent-based distributed software systems to capture requirements in such a way that they can be easily and safely reused during system evolution. By mission-critical, we mean any system in which a hazard can degrade or prevent the successful completion of an intended operation [7]. Our approach uses a software product-line perspective to promote reuse in agent-based, software systems, and is rooted within the requirements engineering phase. Our approach maintains consistency with the widely used Gaia methodology [18].

The contribution of this paper is that it provides a requirements specification pattern so that the dynamically changing configurations of agents can be captured and reused for future similar systems. By adopting a product-line-like view of an agent-based, software system, the approach presented here maps readily into an existing tool-supported safety analysis technique [4]. This gives greater assurance to developers that the requirements reuse

is indeed safe and will not compromise the system via unsafe agent (e.g., feature) interaction. Specifically, the requirements specification pattern and safety analysis presented in this work perform particularly well in analyzing requirements for safe reuse. We motivate and illustrate this work through a specific application, a phased deployment of agent-based, distributed systems such as satellite constellations [13, 15].

The remainder of the paper is organized as follows. Section 2 reviews related research in AOSE, product-line engineering, software safety analysis and the satellite application. Section 3 presents our approach to defining the requirements of an agent-based, distributed system. Section 4 provides step-by-step guidance for capturing the requirements of an agent-based system using the requirements specification pattern presented in this work. Section 5 describes how to use the requirements specification detailed in Section 3 for requirements reuse during system evolution. Section 6 details how the requirements specification pattern is used in safety analysis to ensure safe requirements reuse. Finally, Section 7 provides concluding remarks and future research directions.

2. RELATED WORK

The research presented here reaches into three distinct areas of software engineering: agent-oriented software engineering (AOSE), software product-line engineering and software safety.

In recent years, numerous AOSE methodologies have been proposed for various agent-based application domains. Specifically, this work builds upon the Gaia AOSE methodology [18]. Gaia provides a comprehensive analysis and design framework for generic agent-based software systems supplying several schemas, models and diagrams to capture the requirements of an agent-based system. Although Gaia provides a mechanism to allow the role an agent takes on to change dynamically, it is unclear how to document agent requirements during the analysis and design phases when an agent can ascend or descend roles in an organizational view (i.e., gain or lose intelligence). Additionally, the Gaia methodology fails to provide a mechanism in which the requirements specification templates developed during the analysis phase can be reused during system evolution.

We illustrate how this approach can be used through its application to portions of an agent-based implementation of the TechSat21 (Technology Satellite of the 21st Century) requirements [13]. TechSat21 is a mission designed to explore the benefits of a distributed approach to satellites employing agents [1, 15]. TechSat21 is a constellation (i.e., cluster) of context-aware microsatellites (weighing under 100 kilograms) in which new microsatellites will be deployed to the constellation in phases with new microsatellites potentially having additional capabilities not found in previously deployed microsatellites while sacrificing functionality found in other microsatellites [1, 15]. Within the TechSat21 constellation, each microsatellite must know its context to meet some safety requirements placed upon the constellation (for example, each microsatellite must know its position in relation to others to avoid collisions). Similarly, microsatellites within the constellation must cooperate to meet mission requirements which may be safety-related.

Schetter, Campbell and Surka have applied this work to a specific, planned constellation of satellite systems [13]. They proposed a multi-agent architecture that would better support fault tolerance and upgradability. Chien et al. have similarly proposed a high degree of agent autonomy for a constellation of satellites [1].

This paper extends these previous efforts by focusing specifically on how software safety can be enhanced in these multi-agent systems. Constellations of satellites are large, distributed multi-agent systems. The software in these systems is safety-critical to the extent that it can cause, or can prevent, a hazardous situation from occurring [10]. Loss of mission, as well as loss of a satellite, is a hazard of concern. Since safety is a system property, the development of safety-critical software requires a clear understanding of the agent's role in, and interaction with, the system

We follow Northrup 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 [12]. Weiss and Lai have defined a two-phase software engineering approach: the domain engineering phase and the application engineering phase [17]. The domain engineering phase defines the product line requirements; the application engineering phase reuses these to develop new product-line members.

Recent work by the authors and others has since shown that standard safety analysis techniques are extensible to such product lines [6, 11]. The safety analyses themselves thus become reusable artifacts of the multi-agent distributed system. Ascertaining that safety properties continue to hold as the system grows or evolves (e.g., as new features or satellites are added) is thus much easier and less costly [4, 5].

3. APPROACH

To illustrate our approach, we will use the agent implementation of the TechSat21 microsatellite constellations detailed in [13]. Like them, we define an agent as a major onboard subsystem of a microsatellite or the microsatellite itself [3, 13].

As in the Gaia methodology [18], we define the requirements of an agent's role in terms of the following characteristics: protocols, activities, permissions and responsibilities. *Protocols* define the way an agent can interact with other agents. *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" [18]. 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.

In order to be able to reason about reuse and system evolution, we define here *variation points* for each role. Variation points give a classification of the different levels of intelligence that the role can adopt during its lifetime. For example, TechSat21 roles [13] display the following variation points:

- I4: receive/execute commands
- I3: local planning and receive/execute commands

Role Schema: Cluster Allocation Planning Agent	Schemata 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.	

Figure 1. Role Schemata for cluster allocation planning agent of TechSat21

- 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 would occur when a role is demoted from a higher intelligence level to a lower intelligence level (from I2 to I3, for example).

The variation points indicated in this example will not be universal to all agent-based, distributed systems. Variation points may be particular to each application. For example, other variation points could include active, passive; hot-spare, warm-spare, cold-spare; etc. Further, for every variation point identified, we also 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 TechSat21, the binding time for all variation points identified is at run-time.

Variation points are added with the Gaian characteristics of a role. 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 a few variations. Identifying the variation points to which a role may dynamically switch allows us to classify at which variation points protocols, activities, permissions and/or responsibilities are introduced to the role. Partitioning the requirements (i.e., the protocols, activities, permissions and responsibilities) of an agent in this manner will allow us to reuse the requirements for future systems. 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. This situation is commonly found during system evolution. Section 5 gives a fuller description of how requirements can be reused.

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

4. CAPTURING THE REQUIREMENTS DURING ANALYSIS AND DEVELOPMENT

Requirements are documented in two schematas. The Role Schemata, shown in Figure 1, defines a role and the variation points that the role can assume during its lifetime. The Role Variation Point Schemata, shown in Figure 2, captures the requirements of a role variation point's capabilities. Both schematas are slightly modified adaptations of Gaia's Role Schemata [18].

During the initial development of a distributed system (the product-line domain engineering phase [17]), the focus must be primarily on identifying the overall requirements of the system. It is later (during the product-line application engineering phase [17]) that actual members of the distributed system can be instantiated with some or all of the requirements established earlier. We consider those initial requirements in the Role Schemata and the Role Variation Point Schemata.

To capture the requirements and document them in the two schematas, we follow the following procedure:

1. Identify the roles within the system. Each role will constitute a new Role Schemata 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 Schemata.
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 Schemata 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 Schemata. For each Role Variation Point Schemata:
 - 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 Schemata. Indicate the variation point identification

Role Schema: Cluster Allocation Planning Agent	Schemata 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, <u>AssignCluster</u> , <u>AcceptDeltaVBids</u> , <u>RequestDeltaVBids</u> , <u>SendMoveNewPosMsg</u> , <u>SendDeOrbitMsg</u>	
Permissions: Reads - <i>position</i> // current position of the microsatellite <i>velocityIncrement</i> // current velocity increment of the microsatellite supplied <i>microsatelliteID</i> // identification number of a microsatellite supplied <i>velocityIncrment</i> // velocity increment of a microsatellite Changes - <i>position</i> // current position of the microsatellite <i>velocityIncrement</i> // current velocity increment of the microsatellite Generates - <i>newPositionList</i> // 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.	

Figure 2. Role Variation Point Schemata for the I1 variation point of the cluster allocation planning agent of TechSat21

tag (corresponding to the variation point identification in Step 3) in the appropriate field in the Role Variation Point Schemata.

- 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 following with the Gaian conventions, activities are distinguished from protocols by being underlined).

These steps result in a set of Role Schematas that have an associated set of Role Variation Point Schematas. Additionally, these steps conform with the domain engineering phase of product-line development [17]. Figure 1 illustrates a Role Schemata example and Figure 2 gives an example of a Role Variation Point Schemata, both derived from the TechSat21 agent specifications given in [13].

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 schematas, we instantiate a new member to be added to the distributed system by specifying each new member to be deployed in the Agent Deployment Schemata. An example is shown in Figure 3.

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 Agent Deployment Schemata and:
 - a. Provide the role's name, a unique system(s) identification and a brief description of the role specific to this deployment in the appropriate fields of the Agent Deployment Schemata. The system(s) unique identification, to be place in the System ID field, shall identify the specific member(s) of the distributed system to be deployed that have the role configuration described in the particular Agent Deployment Schemata. For example, if members #2,3, 8-10 all are to employ the Cluster Allocation Planning Agent in which only variation points I2 and I3 are possible, we denote this in the System(s) ID field 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 Schemata was developed for the particular role.

Agent Deployment Schema: Cluster Allocation Planning Agent	System(s) ID: 2,3
<p>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.</p>	
<p>Variation Points:</p> <p>I4: receive/execute commands [F32-I4]</p> <p>I3: local planning and receive/execute commands [F32-I3]</p>	

Figure 3. Agent Deployment Schemata for a member of the TechSat21 Constellation

- c. 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 Schemata was developed for the particular role.
- d. Identify the variation points in which the role will be deployed and denote it in the Agent Deployment Schemata. 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 Agent Deployment Schematas describing how different members of the distributed system that are to be deployed shall be instantiated. These steps conform to the application engineering phase of product-line engineering [17]).

5. MULTI-AGENT REQUIREMENTS REUSE

Requirements reuse is the ability to safely use previously defined requirements for an earlier product and apply them to a new, slightly different product. Increasing the amount of requirements reuse for any given product will, in turn, reduce the production time and cost of a product [2]. Requirements reuse for agent-based, distributed systems is simplified in our approach by taking advantage of how the requirements for an agent's role were partitioned and documented based on their variation points (i.e., various intelligence levels). This section describes how, using the requirements specifications detailed in Section 4, requirements can be reused during the initial deployment of a distributed system as well as during system evolution.

5.1 Requirements Reuse During Development

The members of a distributed system often will be heterogeneous in their functional capabilities. For example, some microsatellites of the TechSat21 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. That is, several member of the distributed system will have the same role but at different levels of intelligence.

Requirements reuse can be exploited during the initial development and deployment of the members of a distributed system using the Agent Deployment Schemata, illustrated in Figure 3 and described in Section 4. Rather than repeatedly defining the requirements of a role for any given agent, the Agent Deployment Schemata allows us simply to define the intelligence levels it can assume. This reuse is possible because the requirements for each of the levels of intelligence were documented in detail in the Role Variation Point Schematas. 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 Schematas.

5.2 Requirements Reuse During System Evolution

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., additional requirements) [15]. Examples of this include improved sensors, new scientific software, new communication devices, etc. 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.

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 Schemata for the new intelligence level (i.e., variation point) giving the roles name, variation point's name and a unique variation point identification in the appropriate fields.
2. Document a description of 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 Schemata in which the new variation point corresponds to and add the new variation point, along with a description and schemata reference identification, to the Variation Points section.

These steps will produce a new variation point for a role as well as the accompanying Role Variation Point Schemata for future versions of members of the agent-based distributed system.

6. APPLICATION OF REQUIREMENTS TO SAFETY ANALYSIS

A challenge to safety analysis of multi-agent distributed systems, such as constellations of satellites, is the ability of agent-based software systems to dynamically alter their configurations (for example, from active to passive). In addition, we would like to reuse some safety analysis results while ensuring the maintenance of safety. That is, a tradeoff of higher reuse potential for less safety in the final product is not acceptable.

To both address the need for a software safety analysis mechanism and to exploit the reuse potential within the requirements specification pattern, detailed in Section 4, we utilize an existing, tool-supported software safety analysis technique. In previous work [4, 5], we have developed the PLFaultCAT software tool, a partially automated safety analysis tool, based on software fault tree analysis (SFTA) [10], to exploit reuse of safety analysis of requirements for software product lines. This technique has proven to be useful in assuring maintenance of safety throughout requirements reuse.

The PLFaultCAT tool can also accommodate the requirements of an agent-based, distributed system documented in the Role Schemata and the Role Variation Point Schemata(s). The Safety Properties (found under the Responsibilities section of the Role Variation Point Schemata) help guide the safety analysis.

A fault tree is a directed AND/OR graph that represents a hazard and its contributing causes. Each node is an event or condition that can contribute to the occurrence of the hazard.

To build a software fault tree (SFT) for any given role and its associated variation points, the following steps should be taken:

1. Determine the root node of the SFT. The root node is a hazard of concern in the system. It may come directly from a negation of one of the safety properties (listed in the Safety Properties section of one of the Role Variation Point Schemata) or from a previously determined domain specific hazards list.
2. Repeatedly generate a list of causes for each failure event starting at the root node. This process continues until the desired granularity is achieved. This process heavily relies

on domain knowledge, previous experience and in-depth requirements analysis. The causes for each failure may come from requirements of any, all or a combination of a role's variation points.

3. From the list of failures and causes generated in the previous step, construct a tree connecting the causes of a failure to a failure by logical AND or OR gates. This tree should now resemble a traditional SFT.
4. Input the constructed SFT to the PLFaultCAT tool.
5. For each of the leaf nodes of the resulting SFT in PLFaultCAT, consider which role's variation points is the source of the fault and tag the node accordingly. To tag the leaf node so that it is associated with a variation point, we use a circular node in PLFaultCAT and document the name of the variation point that can cause the leaf-node failure. Note that it is possible that more than one variation point is tagged to one leaf-node failure. In this case, the tags representing a variation point should be connected to the leaf node via an OR gate (since for any given role only one variation point can be active at any given time).

These steps yield a SFT in which every leaf node is associated with one or more of the role's variation points. For a more thorough treatment on how this type of SFT is constructed, see [3, 4]. Note that although the original tool considered variabilities of a product line, we have found that variation points of an agent's role can be usefully viewed in the same way.

After constructing and inputting a SFT of an agent-based system using the Role Schemata and Role Variation Point Schemata requirements specification pattern, we can automatically generate a SFT for a particular role regardless of which variation points it contains for any given member of a distributed system. Using PLFaultCAT, we specify which variation points a member has and PLFaultCAT automatically trims the SFT to produce a SFT specific to the combination of variation points selected.

This mechanism of SFT construction and PLFaultCAT utilization provides an initial safety analysis of an agent's role (including its variation points). This approach also allows for reuse of some of the safety analysis artifacts in that the SFT can then be automatically derived for any agent employing the same role and a combination of the role's possible variation points.

This safety analysis process can be reused and extended during system evolution. For example, if another variation point is added to a role, leading to the creation and documentation of a new Role Variation Point Schemata, the following process is adequate to update the SFT originally input to PLFaultCAT:

1. Determine how the new variation point can contribute to the root node hazard and each non-leaf node of the SFT.
2. Repeatedly generate a list of causes for each new failure event created from the previous step.
3. From the new list of failures and causes, add the new nodes in PLFaultCAT to the previously constructed SFT.
4. For each leaf node of the updated SFT, consider whether the new variation point can contribute to the fault. Those leaf nodes that can be caused by the new variation point are tagged in a similar manner as when the SFT was originally created.

This process produces an updated SFT within PLFaultCAT such that SFTs can be automatically generated using the new variation point and the previously documented variation points.

The use of a SFT in this manner provides software engineers some assurance that the system requirements are safe (i.e., will not contribute to the hazards). In the TechSat21 example, a SFTA for the possibility of the failure "Collision during cluster reconfiguration" for the role "Cluster Allocation Planning Agent" for the variation points (I1, I2, I3 and I4) provided some assurance that the mission-critical system is not vulnerable to this single-point failure. Using PLFaultCAT as described above, designers can quickly generate SFTs for all variation point combinations of the Cluster Allocation Planning Agent after the initial construction of the original SFT. This is both more efficient and more effective than serially constructing all the trees from scratch for the power set of the variation points (I1, I2, I3 and I4) for the Cluster Allocation Planning Agent.

7. CONCLUDING REMARKS

This paper adapted portions of the Gaia agent-oriented methodology and integrated them with a product-line-like approach to support the safe reuse of the derived requirements of an agent-based, distributed system. The requirements specification template is constructed in such a way that varying dynamic software configurations of an agent are separated. The benefit is that the agent's configurations can then be reused during system evolution. Future deployments of members of the agent-based, distributed system will incorporate behavior similar to the previously deployed members but also include additional capabilities. Section 5 shows how these variation points or configurations can be readily incorporated into the requirements specifications using the requirements specification template(s).

This paper also described how the requirements specification template can be used with an existing, tool-supported, extensible, safety analysis technique that we previously developed. This approach provides a safety analysis technique that is sensitive to the dynamic configurations within the components (i.e., agents) of a system and capable of supporting safe reuse. Planned future work includes an investigation of how feature dependencies can be better handled within the established framework. It is hoped that this approach will provide the means to support the safe reuse of requirements for mission-critical, agent-based, distributed software systems.

8. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants 0204139 and 0205588, and by the Iowa Space Grant Consortium. We thank Barbara Nsiah for the description of product-line commonalities and variabilities in constellations. Her work was supported by the National Science Foundation REU program under grant 0311876 and by the Iowa Space Grant Consortium.

9. REFERENCES

- [1] Chien, S. et al., "The Techsat-21 Autonomous Space Science Agent", *Proc. 1st International Conference on Autonomous Agents*, pp. 570-577, 2002.
- [2] Clements, P. and Northrop, L., *Software Product Lines*, Addison-Wesley, Reading, MA, 2002.
- [3] Das, S., Krikorian, R. and Truszkowski, W., "Distributed Planning and Scheduling for Enhancing Spacecraft Autonomy", *Proc. 3rd Conference on Autonomous Agents*, pp. 422-423, 1999.
- [4] Dehlinger, J. and Lutz, R. R., "PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool", *The Automated Software Engineering Journal*, to appear.
- [5] Dehlinger, J. and Lutz, R. R., "Software Fault Tree Analysis for Product Lines", *Proc. 8th IEEE Symposium on High Assurance Systems Engineering*, pp. 12-21, 2004.
- [6] Feng, Q and Lutz, R. R., "Bi-Directional Safety Analysis of Product Lines", *Journal of Systems and Software*, to appear.
- [7] Fowler, K., "Mission-Critical and Safety Critical Development", *IEEE Instrumentation & Measurement Magazine*, vol. 7, no. 4, Dec. 2004, pp. 52-59.
- [8] Girardi, R., "Reuse in Agent-based Application Development", *Proc. 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, 2002.
- [9] Hara, H., Fujita, S. and Sugawara, K., "Reusable Software Components Based on an Agent Model", *Proc. Workshop on Parallel and Distributed Systems*, 2000.
- [10] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, Reading, MA, 1995.
- [11] Lutz, R. R., "Extending the Product Family Approach to Support Safe Reuse," *Journal of Systems and Software*, 53(3):207-217, 2000.
- [12] Northrup, L. et al., "A Framework for Product Line Practice", *Software Engineering Institute*, <http://www.sei.cmu.edu/plp/framework.html>, (current February 2005).
- [13] Schetter, T., Campbell, M. and Surka, D., "Multiple Agent-Based Autonomy for Satellite Constellations", *Proc. 2nd International Symposium on Agent Systems and Applications*, 2000.
- [14] Sutandiyo, W., Chhetri, M. B., Krishnaswamy, S. and Loke, S. W., "Experiences with Software Engineering of Mobile Agent Applications", *Proc. 2004 Australian Software Engineering Conference*, pp. 339-349, 2004.
- [15] "TechSat21 - Space Missions Using Satellite Clusters", *Space Vehicles Factsheets*, <http://www.cs.afri.af.mil/Factsheets/techsat21.html>, (current February 2005).
- [16] Tveit, A., "A Survey of Agent-Oriented Software Engineering", *NTNU Computer Science Graduate Student Conference*, 2001.
- [17] Weiss, D. M. and Lai, C. T. R., *Software Product-Line Engineering*, Addison-Wesley, Reading, MA, 1999.
- [18] 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.