An Iterative Approach for Development of Safety-Critical Software and Safety Arguments

Xiaocheng Ge, Richard F. Paige and John A. McDermid
Department of Computer Science, University of York, UK.
{xchge, paige, jam}@cs.york.ac.uk

***Abstract*—The benefits ascribed to Agile methods are attractive to software engineers working in the safety-critical software domain. There is limited industrial experience and evidence of successful applications of Agile methods in this domain; however, academic research has identified some of the key challenges of their adoption and application, and has started to present feasibility studies. In this paper, we propose an iterative approach for developing safety-critical software, making two novel contributions. Firstly, we address the notion of *up-front design* in safety-critical software development, and describe the characteristics of an up-front design that is minimal from the perspective of achieving safety objectives. Secondly, we identify a key difficulty of using iterative development for building safety-critical software, and present a way to develop both a software system and a safety argument iteratively. We also give details of a proof-of-concept example illustrating the use of the approach.**

***Keywords*-Safety-critical system, Agile method, Iterative development, Safety arguments**

I. Introduction

Software systems in the safety domain are becoming increasingly crucial and complex. Safety-critical systems are those where any failure is likely to result in the loss of human life or the damage to the environment. In the past 30 years, there is substantial evidence that software flaws can contribute to accidents and failures involving safety-critical systems, e.g., Therac-25 [1] and Ariane 5 [2], and more recently Boeing 777-200 (registered 9M-MRG) [3] and the Toyota Prius[1].

The development of non-critical software systems prioritises the satisfaction of customers requirements. In the case of critical systems, the quality of the system is sometimes more important, or at least as important as its functionality. In terms of system quality, the system must comply with certain standards or guidelines in its own application domain. DO-178B [4] is such a guideline, and is accepted as a standard against which avionics software is certified. Software in safety-critical systems is normally developed following specific, precisely defined processes that are influenced by relevant safety standards and guidelines, which mandate either steps to be followed, or evidence to be delivered from the process. Traditionally, the development of safety-critical systems is approached following a rigorous *heavyweight* process – such as the V model [5]. Such a process is characterised by emphasis on *up-front design* and the production of documentation (typically for use by safety engineers or certifying authorities) in each step of the development process. To develop safety-critical systems, organisations are often required to adopt such processes, but their adoption can make it difficult to manage requirements volatility, introduce new and emerging technologies, and can lead to substantial costs in producing and maintaining documentation. Needless to say, Agile methods are very attractive to software engineers and project managers working in the safety domain, while posing difficulties and challenges to safety engineers working in this domain.

Are Agile methods applicable to developing safety-critical software systems? In [6], Boehm performs a comparative study of Agile methods *vs.* plan-driven methods in developing software and asserts that it is important to know which method is applicable to what type of *project*. It was suggested that critical systems require stable requirements, often have a number of inflexible requirements, and that Agile methods might not be best suited for such applications [7]. Research into the suitability and applicability of Agile methods for safety-critical software development is still at an early stage; there is yet to be a successful application of an Agile method to a safety-critical project reported in the literature.

This paper argues that the lightweight and iterative approach taken in Agile methods can improve the development of safety-critical systems. What it does not do is argue that Agile methods are directly applicable to developing safety-critical systems that require certification. The argument comes in three parts. The first part briefly reviews the development processes typically used for safety-critical systems, and contrasts them with Agile processes. As a result, we argue that the iterative and incremental style of agile development is not inherently at-odds with the requirements of safety-critical systems development. The second part of the paper proposes an iterative process for developing safety-critical software system; the novelty of the process is in its definition of up-front design (appropriate for the safety-critical domain) and the incremental development of safety arguments (explained in the sequel). While we do not claim that the process is agile, we do argue that it is more likely that it is Agile than existing processes. The third part of the paper is an overview of an example that demonstrates how a safety-critical system can be developed iteratively and incrementally, while also producing a *safety argument*, which is generally required for certification.

Whether or not applying Agile practices can improve the development of safety-critical systems is not a simple question to answer; we argue that each development activity in existing

[1] See CNN news, http://edition.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complaints/index.html

978-0-7695-4125-9/10 $26.00 © 2010 IEEE
DOI 10.1109/AGILE.2010.10

Agile methods will have its own impact. This paper focuses on the effect of two specific activities in Agile methods: lightening the up-front design of safety-critical system development and iterative development of system and its safety arguments.

II. Background

In this section we review a typical safety-critical development process, and relate it to Agile processes for the purposes of identifying commonalities and areas of incompatibility. This in turn will help us to derive two key challenges that must be overcome to allow safety-critical software development to benefit from the principles and practices of Agile development.

A. Safety-critical software development processes

The typical process of developing a safety-critical software system is generally time-consuming. Most such processes are based on the V-model, which is illustrated diagrammatically in Figure 1. This model identifies the major elements of the development process and indicates the structured, and typically sequential, nature of the development process. The sequential nature of development is generally considered essential for reasons of managing communication and scale, for scheduling different phases and disciplines, for managing traceability (which is mandated by relevant safety standards) and for certification purposes.

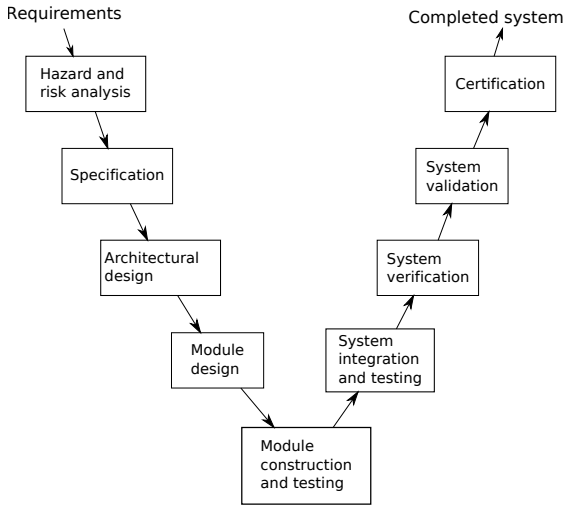

Fig. 1. The V model of safety-critical system development [5]

The last phase in the life-cycle is *certification*, which is often the most important concern of the critical system development process. In order for a safety-critical system to be put in service, it must be tested and evaluated by independent third parties (i.e., a certifying authority) against demanding and standardised criteria, in order to obtain a certificate for release to service; we refer to this as certification although the terminology used may vary between sectors.

The need for assuring the quality of systems in critical applications has led governments and international regulatory organisations into developing *consensus* standards or guidelines, which must be adhered to before any system of a critical nature is commissioned. Those standards and guidelines are means of specifying how critical systems *should be* developed, based on best practices and past experience gathered over many years.

In order to certify the quality of software, different standards or guidelines are applied for different application domains. For example, DO-178B, Software Considerations in Airborne Systems and Equipment Certification, is such a guideline in avionics domain. It is accepted as a means of certifying software by the FAA (Federal Aviation Administration). In general, standards and guidelines have at least two purposes. On one hand, they aim to provide a way for the producers of critical systems to determine whether the system they are acquiring is free (to the greatest technical extent possible) from defects which may result in losses; on the other hand, they provide the developers of critical systems with guidelines on how to develop such systems, so that the resulting products will perform according to the stakeholders' expectations.

Like most safety standards, DO-178B defines a *framework* for the development process, rather than mandating a particular process or process model. DO-178B addresses three separate processes that comprise most life-cycles and describes the interactions between them: a planning process, a technical development process, and an integration process. While DO-178B does not specify a process, it does identify objectives which any process must meet. For each software product to be developed according to DO-178B, a software life-cycle(s) is constructed that includes these three processes.

In practice, heavyweight development processes currently dominate industrial safety critical software development. Such processes are widely accepted, are integrated with certification standards, and are generally based on well-defined principles (e.g., waterfall life-cycle model). Heavyweight processes also share a common view of the phases which make up an ideal software development life-cycle, namely *analysis*, *design*, *implementation*, and *testing and integration*.

B. Overview of Agile processes

Agile processes are in effect a family of development processes. Typical Agile development methods include eXtreme Programming (XP) [8], [9], [10], Feature Driven Development (FDD) [11], [12], [13], and Scrum [14]. Table I briefly summarises the common features of these three Agile methods.

In terms of development process, each Agile method introduces its own life-cycle model. For example, the XP process consists of five phases: **Exploration**, **Planning**, **Iterations to release**, **Productionizing**, **Maintenance** and **Death**. By comparison, a Scrum process has only three: **pre-game**, **game**, and **post-game**.

We will not explore the details of these phases. We observe that, despite different names for these phases in different Agile methods, there is commonality, and it is not difficult to produce a generic software development life-cycle model (SDLC) for Agile methods, which is based on the tasks and

Method	Key Points	Special Features	Identified Shortcomings
XP	Customer driven development, small teams, daily builds.	Refactoring - the ongoing redesign of the system to improve its performance and responsiveness to change.	While individual practices are suitable for many situations, an overall system view and management practices are given less attention.
FDD	Five-step process, Object-oriented component (i.e. feature) based development. Very short iterations: from hours to 2 weeks.	Method simplicity, design and implementation the system by features, object modelling.	FDD focuses only on design and implementation, and needs other supporting approaches.
Scrum	Independent, small, self-organising development teams, 30-day release cycles.	Enforce a paradigm shift from the "defined and repeatable" to the "new product development view of Scrum."	While Scrum details specifically how to manage the 30-day release cycle, integration and acceptance tests are not detailed.

TABLE I
GENERAL FEATURES OF AGILE METHODS (DERIVED FROM [15])

activities in each phase. We illustrate this generic life-cycle model in Figure 2. The generic life-cycle model consists of four phases: *preparation*, *planning*, *short iterations to release*, and *integration*.

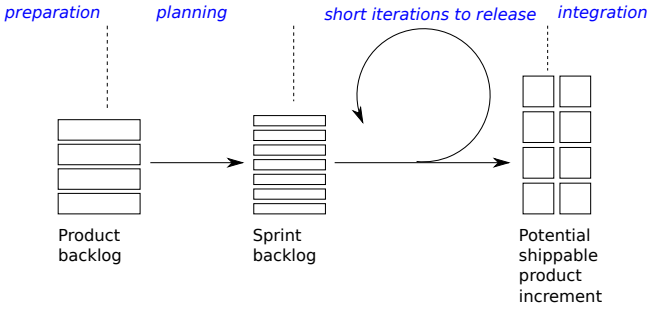

Fig. 2. Generic development lifecycle of Agile methods

Different Agile methods may have different names for the **preparation** phase, For example **Exploration** in XP, **Pre-game** in Scrum[2], and **Develop an overall model** in FDD. The development task in this phase is to understand the blueprint and environment of the system. The product of this phase is a package of current requirements or user stories which provide materials for the next phase, *planning*.

Planning is the stage where the whole system is disassembled into pieces according to an appropriate priority index. For example, in XP the user stories are prioritised and allocated to the development iterations in phase **Planning**; and the planning job is done in **Pre-game** phase of Scrum and the stage **Build a feature list** and **Plan by feature** of FDD.

Once the objective of each development iteration is clear, the system is constructed iteratively through many *short iterations of release*. *Short iterations of release* is a key characteristic of all Agile methods, although it goes by different names. For example in Scrum, it is called the **Game** phase, whereas in XP it is called **iterations to release**. Each development iteration is short, usually 2 to 4 weeks. At the end of every iteration, tests are carried out to verify and validate the release. When the last iteration is completed, the system is ready for *integration*.

[2]The planning work is done in **Pre-Game** phase too.

In the *integration* phase, the system undergoes additional testing and checking of satisfaction of environmental variables, e.g., requirements. This phase is instantiated in phases **Productionizing** and **Maintenance and Death** of XP, and **Post-game** in Scrum.

There are commonalities between the generic agile life-cycle and the steps carried out in a plan-driven process (which consists of four activities: analysis, design, implementation and testing). More specifically, at the beginning and the end of an Agile development process, software engineers have to do the same sort of development activities (i.e., **analyse** and **design** at the beginning, and **test** at the end), and in the iterations the four activities are performed in the same sequence. Figure 3 shows the commonalities and differences between plan-driven and Agile processes.

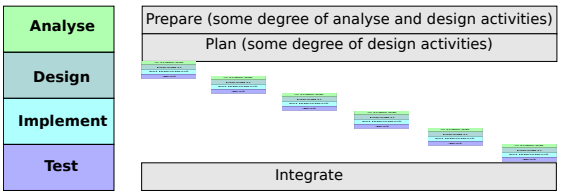

Fig. 3. Plan-driven development *vs.* Agile development

The core differences between Agile and plan-driven processes are thus as follows.

1) Plan-driven processes have an up-front design (including the analysis activity). Instead of planned design, the design in Agile processes is evolutionary. Compared with plan-driven processes, the shape of the system rather than its detail are designed at the beginning of an Agile process, which makes up-front design in an Agile process lighter.
2) In plan-driven processes, implementation is normally a monolithic activity, which also entails a monolithic testing phase. In an Agile process, implementation (and hence testing) are iterative and incremental.

In order to enable the development of safety-critical software systems using Agile processes, there are thus two key requirements:

- *"lighten"* the amount of effort required in up-front design of a safety-critical system,
- support iterative development.

Experience shows that there are few planned development iterations in industrial safety-critical system development projects; these iterations occur typically at later stages of the development process, e.g., coding and testing. In this paper, we discuss the activities which occur at early stage of development, i.e, designing and planning, which seem to be a more constructive way to emphasise the benefits of Agile development. In the next section, we describe our efforts towards achieving these two requirements.

III. Towards an Acceptable Agile Process for Developing Safety-Critical Software Systems

We are aware of companies who say that they apply Agile practices; however these are not well-documented (i.e, in the public domain) and the notion of "incremental certification" [16] remains an aspiration, albeit one which is being actively researched. Although there are as of yet no detailed, evidenced reports of an application of an established Agile process successfully applied to a real-world, safety-critical software project that requires certification, there have been a number of research papers arguing that an Agile process, with some customisation, can be applied to critical systems. This includes work on developing both security- and safety-critical software systems, and generally involve modifications to agile practices (e.g., inclusion of greater use of model-based design and auto-coding) [17], [7], [6], [18], [19], [20], [21], [22]. However, current practice and current methods have been informed by decades of experience on long-lived projects; the ability to change current practice on projects that may take 20-30 years to deliver and deploy is both technically and politically difficult. Thus, direct adoption of *any* new method is difficult, though Agile methods offer their own distinctive challenges.

In principle, Agile software development is intended to lead to a development process that is more responsive than traditional plan-driven methods to customer needs. At the same time, Agile processes aim to create software of better quality[3]. The quality of the target software system is based not only the functionality, but also other aspects, such as reliability and other non-functional properties. In principle, therefore, there is no conflict between the principles of Agile processes and the requirements of safety-critical software development projects.

Earlier, we identified two key challenges in adopting Agile methods for building safety critical software systems: managing up-front design; and carrying out iterative and incremental development. We now discuss each challenge in turn and explain how these challenges can be overcome.

A. Up-front design in safety-critical project

Model with a purpose is a core Agile principle. In [23], Ambler states that *"with respect to modelling, perhaps you need to understand an aspect of your software better, perhaps you need to communicate your approach to senior management to justify your project, or perhaps you need to create documentation that describes your system to the people who will be operating and/or maintaining/evolving it over time. If you cannot identify why and for whom you are creating a model then why are you bothering to work on it all?"*

In Agile projects, the plan for iterative development effectively constitutes a form of up-front design. This plan is normally incomplete and focuses more on the shape (or abstract architecture) of the software system rather than its detailed design; FDD and Scrum both take this approach. However, for safety-critical projects, the up-front design/plan needs to be sufficiently detailed to serve as input to *hazard analysis*, which in turn informs the safety analysis and certification processes later. Checklists in some domains or systematic techniques such as Functional Failure Analysis (FFA), are carried out at the start of the development. As design progresses, more detailed analyses are carried out using techniques such as Fault tree analysis (FTA), Hazards and operability analysis (HAZOP), or Failure modes and effects analysis (FMEA); there is usually iteration between the design and analysis processes, with issues found in analysis leading to design changes. In the early stages of safety critical development, the preliminary hazard analysis is based on an architectural model of the system. Thus, an up-front plan for safety-critical software development must include not only up-front architectural design but also up-front hazard analysis, and as the architecture evolves, the hazard analysis must be extended. The more critical the system (e.g., in terms of safety integrity level), the more effort that may need to be put into hazard analysis and into the software development to demonstrate the integrity of the software. The question we must answer is: for a specific project, how can we lighten its up-front design load, while still achieving acceptable levels of detail in our hazard analysis? Importantly, this gives us a precise notion of sufficiency for our planning and up-front design.

Underpinning this question is a general issue about assessing the *sufficiency* and *adequacy* of software models in general. It is difficult to define general-purpose criteria about whether or not an up-front model is sufficiently accurate and sufficiently detailed in an *agile* safety-critical software project, because different hazard analysis techniques have different information requirements. Table II summarises the information needed for performing three of the most commonly used hazard analysis techniques.

Table II indicates that a description of system structure (i.e., system architecture in some style) is needed by all three techniques. Beside that, to do a FTA on a certain failure, the information of what may cause the failure is needed. The information always comes from the experience of safety engineers or historical data, not directly from the functional design of the system. To do FMEA or HAZOP analysis, engineers not only need to know the system structure, but also the information of the effect of a failure on other components, which sometimes comes from a detailed design of components.

In contrast, in an Agile process, the corresponding up-

[3] www.agilemanifesto.org

Hazard analysis	Information needed for analysis technique
FTA	A system failure (top event) to be analysed for each fault tree; System structure layout; Component (or module) functional descriptions (may not need the details of each function specification if using qualitative analysis).
FMEA	System structure layout; Failure modes and their effect of each component.
HAZOP	System structure layout; A set of potential failure modes of system in terms of key words; Failure behaviours of each component.

TABLE II
INFORMATION NEEDED FOR FTA, FMEA, AND HAZOP ANALYSIS PROCESSES

front 'plan' requires an overview of the system architecture and a user prioritisation of features; for example, a system architecture model and a prioritised feature list are required in FDD. We thus claim that, for an agile safety-critical software project, the up-front design should produce at least a system architecture model as well as functional requirements for each component, along with a priority.

B. Iterative development and planning

Iterative development is perhaps the most important characteristic of Agile methods. Currently, the development of a safety-critical system is not carried out iteratively and incrementally (at least not deliberately at the fine-grained level implied by Agile methods). This is in part because of the mandated requirements of development process in current safety standards and because of the multi-disciplinary nature of safety-critical systems engineering (e.g., software engineers must communicate with system engineers and safety engineers, and this communication must be carefully coordinated). Nevertheless, we can propose an *idealised* iterative process for safety-critical software development, where the release of each iteration should come with evidence and an argument that the release is *acceptably safe*. Here what we mean of iterative development is not the iteration between development and hazard analysis process, or the iterative development of some detailed functional features of the system. We focus on the plan and the development of the entire system. What constitutes acceptable is dictated by the integrity level of the software, the standard that is being targeted for certification, the requirements of the certifying authority, and in some circumstances, the requirement that the risk be reduced as low as reasonably practicable. The point is to iteratively and incrementally construct not only the software, but also the argument that the software is acceptably safe, and to always have an acceptably safe software system with each release. A similar concept, namely evidence-based development[4] has been proposed recently.

As shown in Figure 1, the last phase of the development process of a safety-critical system is *certification*. Certification is a process whereby the system is evaluated by a certifying authority against demanding criteria. To demonstrate that the system is acceptably safe, a document, presenting a detailed *safety argument* of how the safety characteristics of the system are achieved, is normally required. Languages and tools exist for constructing and validating safety arguments [24], [25]. When applying an Agile method to building a safety-critical software system, evidence must still be produced and a safety argument must still be constructed. However, with an Agile method, we must find a way to incrementally build safety arguments, so that we can reuse safety arguments for previous releases in producing a safety argument for the current release. This is difficult because safety arguments are generally monolithic and are constructed for complete software systems [25].

In order to make it easier to construct safety arguments for large systems, the use of modular safety arguments was proposed [25], [26]. This approach allows safety arguments to be developed in terms of modules with argument dependencies. An example of a modular argument structure is shown in Figure 4. In this structure, a safety argument for the whole system *main* contains three sub-arguments for arguing over a particular module in the system. An argument about all the (software) module interactions is made in a separate (argument) module. There are no longer any links between the module arguments as these no longer have the responsibility to reason about their effect on other modules.

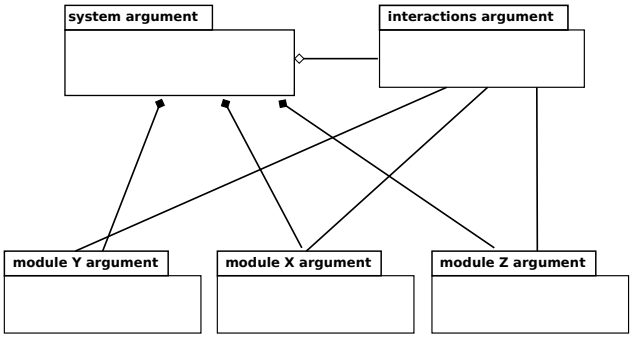

Fig. 4. Modular argument structure

We identified that this technique can be adapted in an agile safety-critical software project as one of references for the plan of iterative development because the plan must take the anticipated safety argument structure into account, i.e., a package of safety arguments and associated evidence should be produced in one iteration. This introduces a potential point of tension. Existing Agile methods suggest that each iteration should be short, normally taking 2-4 weeks. However, the requirement to produce a safety argument structure sometimes will need to override the other requirements of iteration planning in order to ensure that the release of each iteration is acceptably safe. In other words, iterations may need to be extended in duration in order to satisfy requirements for producing a safety argument, since without this argument, the software cannot be deployed.

In each iteration, the software components are developed

[4]See http://www.integrate.biz/services_mi_ebd.html

while in parallel, the safety argument is produced (in typical safety critical software development, the safety argument is not produced by the software development team). Taking the example of Figure 4, the safety argument for the entire system requires four parts: the arguments for individual components X, Y, and Z, and an argument for the interactions between modules. Therefore, the system can be developed in at least three iterations: each iteration produces a module and its safety argument. The safety argument of interactions is produced when the system is integrated at the completion of the third iteration.

IV. Example

In attempting to evaluate our approach, we selected a safety-critical avionics software application. The role played by flight-critical avionics software in aviation safety makes this an ideal test-bed, since avionics software is generally required to possess a high Safety Integrity Level (SIL). The questions addressed by work are whether the analysis of up-front design and the proposed techniques for targeted iterative development are suitable for use in environments that demand extremely low failure rates and adherence to safety standards and regulations. The selected example can help evaluate the lightened up-front design and iterative development to meet the needs of safety-critical software projects.

A. System overview

The software of interest is based on the example in [27]: it constitutes an Integrated Altitude Data Display System (IADDS). It is responsible for providing pilots with accurate and understandable altitude data during flight. It is also responsible for issuing audible and visible warnings to pilots whenever human-specified altitude limits are reached. The system receives altitude data from two distinct sources: a barometric altimeter and a radar altimeter (**Altimeter**). The software processes altitude data via a data processor (**DP**) and displays altitude data through a simple user interface (**GUI**), which provides pilots with controls for selecting the status modes (Status Modes), and enabling or disabling audio warnings or visual warnings (VWF). Figure 5 illustrates the IADDS conceptual architecture.

B. Up-front design and hazard analysis

Along with the functional and safety requirements of the target system, we were given documents which specified hardware failures (e.g., the failure behaviour of the altimeter) and effects. Based on these documents, we produced the system architecture shown in Figure 5. The software system consists of five components: an altimeter, a warning system, a data processor, a status reader, and a GUI. Among them, the warning facility (VWF), status reader, and data processor may be subsystems (and are represented as UML packages), while the altimeter and GUI are components (UML classes).

Based on the hardware failures and effects information, we performed a fault tree analysis on the system architecture; as described earlier, the system architecture was sufficiently detailed (in terms of the behaviour of components, how critical each component is, and what safety countermeasures are to be used for each component) to support FTA. The preliminary hazard analysis report was then fed in to the design stage of the next functional development iteration. We do not include the fault tree here due to space limitations.

C. Iterative development

Based on the system architecture (Figure 5), the safety argument for the system has six parts, which are the arguments about the five components in the system and the interaction between components. Besides these arguments, we also needed to argue for the completeness of the interaction argument. This means that we also needed to demonstrate that unintended interactions between components do not occur. The development of the system consisted of six iterations. The sixth iteration did not deliver new code, but produced the safety argument of interactions and their completeness. The structure for the safety arguments was presented in the Goal Structuring Notation (GSN) [25], and is shown in Figure 6. In the diagram, the safety argument of interactions, module VWF, model Data Processor, and model Status Reader are all packages which have a hierarchical architecture. Once these packages had been designed, the structure of the related safety argument package was developed in more detail.

V. Evaluation

In the example, we found that, even for safety critical software, the up-front design can be as simple as shown in Figure 5; it must be detailed enough to allow the preliminary hazard analysis to be performed. There is also an interesting feedback relationship between the hazard analysis and the development of functional components: the hazard analysis can in turn help to determine where to best spend effort in developing functional components.

The example also shows that a safety-critical software system and its safety argument can be developed incrementally; however, the duration of each iteration may be longer than 2-4 weeks. Development iterations can potentially be shortened if the internal architecture of each component (package) is designed at an early stage and the corresponding safety argument is well-structured, e.g., using the GSN patterns and modular framework presented in [26].

The example allows us to show the feasibility of carrying out iterative development of a safety-critical software system with a minimal up-front design, thus achieving our objectives. However, one problem is that we cannot evaluate the quality of the safety argument we made during the iterations. In other words, we lack measurements that allow us to ensure that the safety argument from each iteration and of the entire system are sufficiently strong. Certain GSN patterns can potentially help us to improve the weakness of the safety arguments. For example, in Figure 6, by applying a pattern, we had to add an extra argument to demonstrate that unintended interactions do not exist. Even so, some arguments may be logically incomplete or logically weak, which is not acceptable.

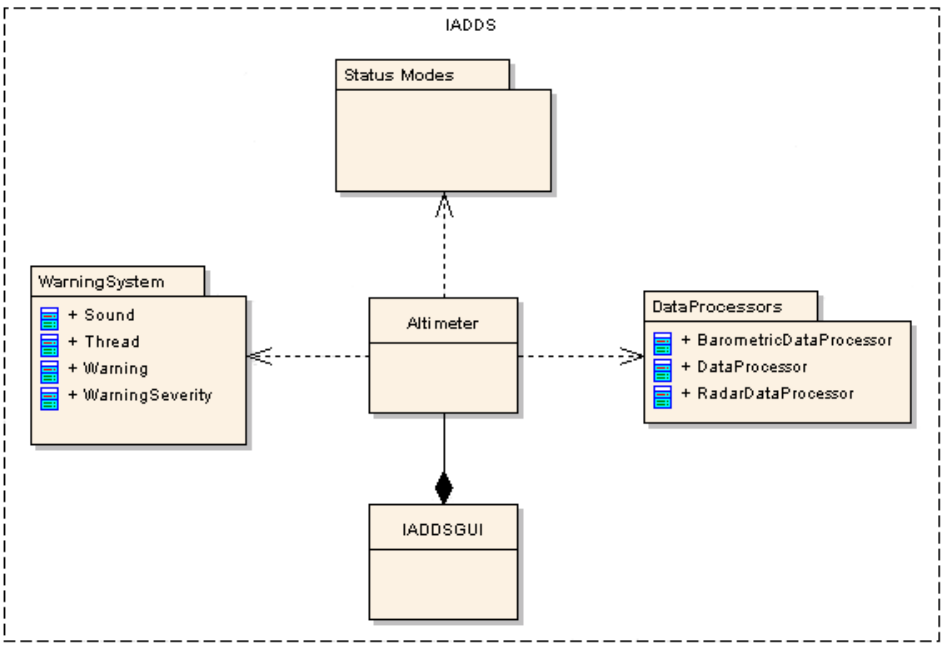

Fig. 5. IADDS Conceptual Architecture

It is difficult to completely and convincingly argue that our approach is an Agile process. In [15], Abrahamsson proposed that an Agile process satisfies four characteristics: *incremental*, *cooperative*, *straightforward*, and *adaptive*. With the exception of the first characteristic, these notions are subjective and difficult to evaluate without substantial user testing; user testing is more difficult in the safety domain since different users from different engineering disciplines (e.g., software, mechanics, electrical) are involved. The only thing that we can conclude is that a process is not Agile if it is not iterative. Therefore, we argue that our process has the *potential* to be an Agile process, providing that it is evaluated as being acceptably cooperative, straightforward and adaptive for practitioners. Our overall conclusion is that a notion of *acceptability* must be inherent in any detailed definition and evaluation of agility.

VI. Conclusion

The advantages of Agile methods, demonstrated in many non-critical software projects, have attracted the attention of researchers and engineers in the safety domain. While there has not been a successful, documented deployment of existing Agile methods on a real industrial safety-critical project, there is room for optimism about their applicability and suitability. Software development practices have been existing in software engineering for many years, those Agile methods have chosen elements which comply with Agile principles from the *"toolbox"* of those practices. What we have been doing in the area of Agile safety engineering is to identify, apply, and evaluate the practices which may have already been existing in safety engineering for many years so that the new adaptation of those practices can enable and improve the agility of safety-critical systems. In the end, the *"Agile"* practices we found may not change the nature of entire development process of safety-critical systems because certification plays a significant influence to the development of these systems. However, those practices can improve the agility of the development bit by bit.

In this paper, we have argued for the potential compatibility between Agile processes and safety-critical software development, and have proposed an iterative process for development. The process in turn precisely captures the notion of *sufficient* up-front design for safety-critical software: it is sufficient to allow a hazard analysis to take place, which will ultimately be used to derive a safety argument for the entire system. We also outlined how to use safety patterns and a modular notion of safety argument to enable the iterative development of safety arguments, and the importance of aligning functional development with development of a partial safety argument (and also the importance of an argument supporting safe interactions of components).

We are developing the work in two ways. Firstly, we are applying the process to other safety-critical software systems, but this time using other forms of hazard analysis and also considering notions of requirements change and their impact on the safety argument. Requirements change in safety critical applications is not as common as in, e.g., the development

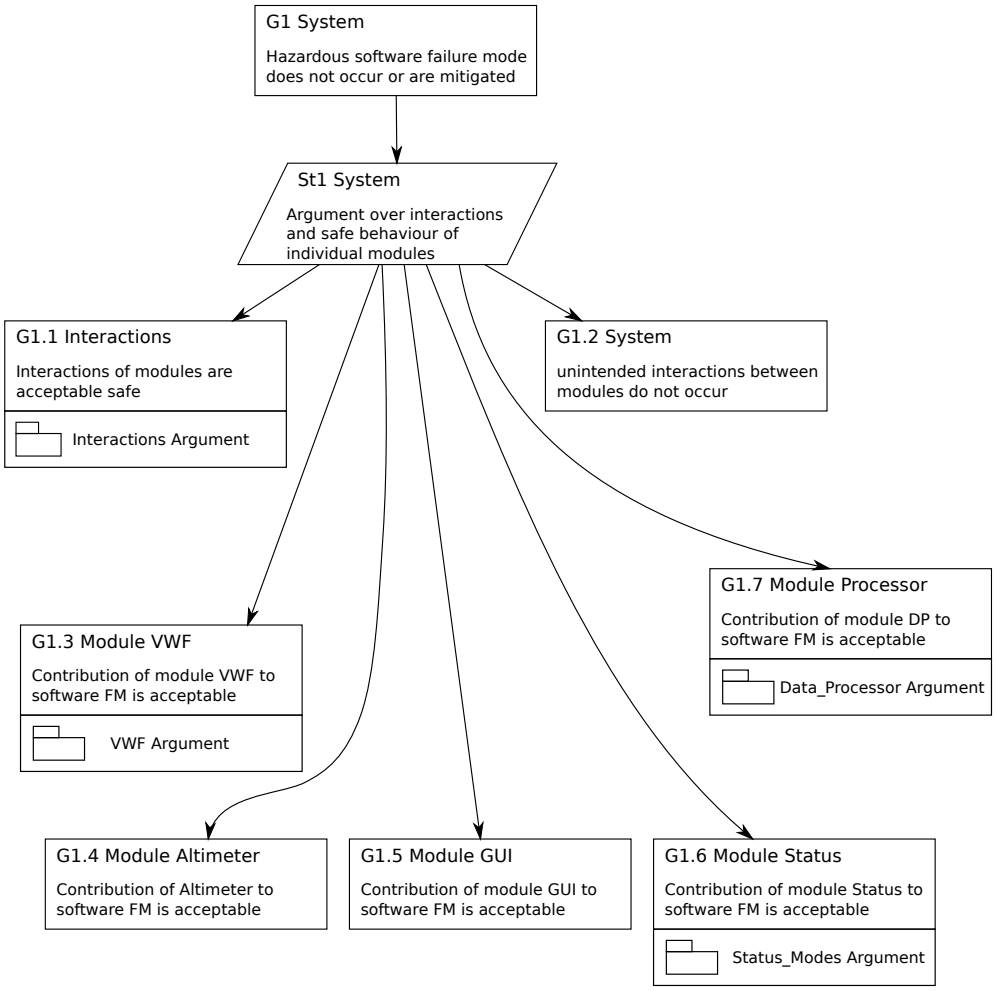

Fig. 6. Safety Argument Structure in GSN

of information systems, but when it does occur it can have substantial impact on software architecture and may invalidate the entire safety argument [7]. We are attempting to identify patterns of requirements change that can lead to more manageable changes to safety arguments, so that (at least) we can potentially predict the impact of requirements change on the safety argument.

Secondly, we will investigate the acceptability of our approach to both safety-critical software engineers as well as agile practitioners. The challenge of adopting new techniques in the safety community is a difficult one to overcome, and we will attempt to carry out structured interviews with practitioners and engineers to ascertain how compatible our approach is with their existing practice.

REFERENCES

[1] N. G. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[2] J. L. Lions, "ARIANE 5: Flight 501 failure," Ariane 5 Inquiry Board Report, Paris, Tech. Rep., 1996.

[3] "Aviation safety investigation report - in-flight upset; 240km nw perth, wa; boeing co 777-200, 9m-mrg," http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx, 2005.

[4] "DO-178B: Software considerations in airborne systems and equipment certification," RTCA Inc. and EUROCAE, December 1992.

[5] N. Storey, *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[6] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

[7] R. F. Paige, H. Chivers, J. A. McDermid, and Z. R. Stephenson, "High-integrity extreme programming," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2005, pp. 1518–1523.

[8] http://extremeprogramming.org/, July 2006.

[9] K. Beck, "Embracing change with extreme programming," *IEEE Computer*, vol. 33, no. 10, pp. 70–77, 1999.

[10] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, November 2004.

[11] S. R. Palmer and J. M. Felsing, *A practical guide to feature-driven development*. Prentice Hall, 2002.

[12] http://featuredrivendevelopment.com/, May 2007.

[13] J. D. Luca, "FDD implementations," http://www.nebulon.com/articles/fdd/fddimplementations.html, May 2007.

[14] http://www.controlchaos.com/, May 2007.

[15] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," Technical Research Centre of Finland, Tech. Rep. ESPOO 2002, VTT Publication 478, 2002.

[16] "Dependable systems enabling agility and networking," http://www.science.mod.uk/strategy/dtplan/sector_helicopters.aspx?ThemeId=ec6ff7da-2f2c-43b3-90f9-edee5aa32ff9&ThemeType=rdo&ActivityId=adec27fc-62d5-4b86-b2a3-7822cdf7bb11&ActivityType=rd&.

[17] G. Boström, J. Wäyrynen, M. Bodén, K. Beznosov, and P. Kruchten, "Extending xp practices to support security requirements engineering," in *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*. New York, NY, USA: ACM, 2006, pp. 11–18.

[18] K. Beznosov, "Extreme security engineering," in *Proc. BizSec*, Fairfax, VA, October 2003.

[19] K. Beznosov and P. Kruchten, "Towards agile security assurance," in *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*. New York, NY, USA: ACM, 2004, pp. 47–54.

[20] J. Grenning, "Extreme programming and embedded software development," in *Proceedings on Embedded System Conference 2002*, Chicago, USA, 2002.

[21] P. Manhart and K. Schneider, "Breaking the ice for agile development of embedded software: An industry experience report," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 378–386.

[22] M. P. E. Heimdahl, "Safety and software intensive systems: Challenges old and new," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 137–152.

[23] S. Ambler, *Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process*. John Wiley & Sons, April 2002.

[24] S. P. Wilson, J. A. McDermid, C. H. Pygott, and D. J. Tombs, "Assessing complex computer based systems using the goal structuring notation," in *ICECCS*, 1996, pp. 498–505.

[25] T. P. Kelly, "Arguing safety a systematic approach to managing safety cases," Ph.D. dissertation, Department of Computer Science, University of York, 1998.

[26] R. D. Hawkins, "Using safety contracts in the development of safety critical object-oriented systems," Ph.D. dissertation, Department of Computer Science, University of York, 2006.

[27] R. F. Paige, R. Charalambous, X. Ge, and P. J. Brooke, "Towards agile engineering of high-integrity systems," in *SAFECOMP*, 2008, pp. 30–43.