Chapter 4
Fundamentals of Designing Complex Aerospace Software Systems

Emil Vassev and Mike Hinchey

Abstract. Contemporary aerospace systems are complex conglomerates of components where control software drives rigid hardware to aid such systems meet their standards and safety requirements. The design and development of such systems is an inherently complex task where complex hardware and sophisticated software must exhibit adequate reliability and thus, they need to be carefully designed and thoroughly checked and tested. We discuss some of the best practices in designing complex aerospace systems. Ideally, these practices might be used to form a design strategy directing designers and developers in finding the "right design concept" that can be applied to design a reliable aerospace system meeting important safety requirements. Moreover, the design aspects of a new class of aerospace systems termed "autonomic" is briefly discussed as well.

Keywords: software design, aerospace systems, complexity, autonomic systems.

1 Introduction

Nowadays, IT (information technology) is a key element in the aerospace industry, which relies on software to ensure both safety and efficiency. Aerospace software systems can be exceedingly complex, and consequently extremely difficult to develop. The purpose of aerospace system design is to produce a feasible and reliable aerospace that meets performance objectives. System complexity and stringent regulations drive the development process, where many factors and constraints need to be balanced to find the "right solution". The design

Emil Vassev · Mike Hinchey
Lero—the Irish Software Engineering Research Centre
University of Limerick, Ireland
e-mail: {Mike.Hinchey,Emil.Vassev}@lero.ie

of aerospace software systems is an inherently complex task due to the large number of components to be developed and integrated and the large number of design requirements, rigid constraints and parameters. Moreover, an aerospace design environment must be able to deal with highly risk-driven systems where risk and uncertainty are not that easy to capture or understand. All this makes an aerospace design environment quite unique.

We rely on our experience to reveal some of the key *fundamentals* in designing complex aerospace software systems. We discuss best design practices that ideally form a design strategy that might direct designers in finding the "right design concept" that can be applied to design a reliable aerospace system meeting important safety requirements. We talk about design principles and the application of formal methods in the aerospace industry. Finally, we show the tremendous advantage of using a special class of space systems called autonomic systems, because the latter are capable of self-management, thus saving both money and resources for maintenance and increasing the reliability of the unmanned systems where human intervention is not feasible or impractical.

The rest of this paper is organized as follows: In Section 2, we briefly present the complex nature of the contemporary aerospace systems. In Section 3, we present some of the best practices in designing such complex systems. In Section 4, we introduce a few important design aspects of unmanned space systems and finally, Section 5 provides brief summary remarks.

2 Complexity in Aerospace Software Systems

The domain of aerospace systems covers a broad spectrum of computerized systems dedicated to the aerospace industry. Such systems might be onboard systems controlling contemporary aircraft and spacecraft or ground-control systems assisting the operation and performance of aerospace vehicles. Improving reliability and safety of aerospace systems is one of the main objectives of the whole aerospace industry. The development of aerospace systems from concept to validation is a complex, multidisciplinary activity. Ultimately, such systems should have no post-release faults and failures that may jeopardize the mission or cause loss of life. Contemporary aerospace systems integrate complex hardware and sophisticated software and to exhibit adequate reliability they need to be carefully designed and thoroughly checked and tested. Moreover, aerospace systems have strict dependability and real-time requirements, as well as a need for flexible resource reallocation and reduced size, weight and power consumption. Thus, system engineers must optimize their designs for three key factors: *performance*, *reliability*, and *cost*. As a result, the development process, characterized by numerous iterative design and analysis activities, is lengthy and costly. Moreover, for systems where certification is required prior to operation, the control software must go through rigorous verification and validation.

Contemporary aerospace systems are complex systems designed and implemented as *multi-component* systems where the components are *self-contained* and *reusable*, thus requiring high independency and complex synchronization. Moreover, the components of more sophisticated systems are considered as agents (multi-agent systems) incorporating some degree of intelligence. Note that intelligent agents [1] are considered one of the key concepts needed to realize self-managing systems. The following elements outline the aspects of complexity in designing aerospace systems:

- multi-component systems where inter-component interactions and system-level impact cannot always be modeled;
- elements of artificial intelligence;
- autonomous systems;
- evolving systems;
- high-risk and high-cost systems, often intended to perform missions with significant societal and scientific impacts;
- rigid design constraints;
- often extremely tight feasible design space;
- highly risk-driven systems where risk and uncertainty cannot always be captured or understood.

3 Design of Aerospace Systems – Best Practices

In this section, we present some of the best practices that help us mitigate the complexity in designing aerospace systems.

3.1 Verification-Driven Software Development Process

In general for any software system to be developed, it is very important to choose the appropriate development lifecycle process to the project at hand because all other activities are derived from the process. An aerospace software development process must take into account the fact that aerospace systems need to meet a variety of standards and also have high safety requirements. To cope with these aspects, the development of aerospace systems emphasizes *verification*, *validation*, *certification* and *testing*. The software development process must be technically adequate and cost-effective for managing the design complexity and safety requirements of aerospace systems and for certifying their embedded software. For most modern aerospace software development projects, some kind of spiral-based methodology is used over a waterfall process, where the emphasis is on verification.

As shown in Figure 1, NASA's aerospace development process involves intensive verification, validation, and certification steps to produce sufficiently safe and reliable *control systems*.

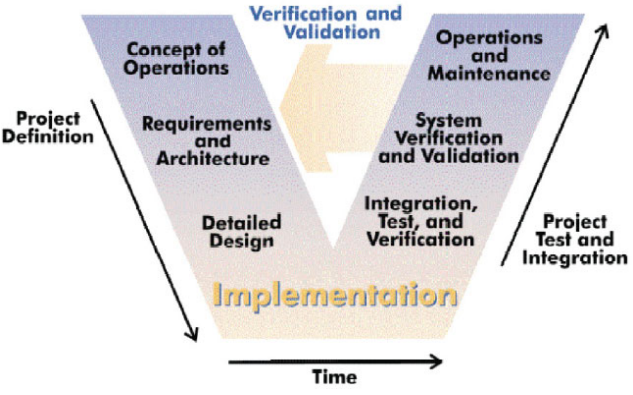

Fig. 1 A common view of the NASA Software Development Process [2]

3.2 Emphasis on Safety

It is necessary to ensure that an adequate level of safety is properly specified, designed and implemented. Software safety can be expressed as a set of features and procedures, those ensuring that the system performs predictably under normal and abnormal conditions. Furthermore, "the likelihood of an unplanned event occurring is minimized and its consequences controlled and contained" [3].

NASA uses two software safety standards [4]. These standards define four qualitative hazard severity levels: *catastrophic*, *critical*, *marginal*, and *negligible*. In addition, four qualitative hazard probability levels are defined: probable, occasional, remote, and improbable. Hazard severity and probability are correlated to derive the risk index (see Table 1). The risk index can be used to determine the priority for resolving certain risks first.

Table 1 NASA Risk Index Determination [4]

Hazard Severity	**Hazard Probability**			
	Probable	Occasional	Remote	Improbable
Catastrophic	1	1	2	3
Critical	1	2	4	4
Marginal	2	3	4	5
Negligible	3	4	5	5

3.3 Formal Methods

Formal methods are a means of providing a computer system development approach where both a formal notation and suitable mature tool support are provided. Whereas the formal notation is used to specify requirements or model a system design in a mathematical logic, the tool support helps to demonstrate that

the implemented system meets its specification. Even if a full proof is hard to achieve in practice due to engineering and cost limitations, it makes software and hardware systems more reliable. By using formal methods, we can reason about a system and perform a mathematical verification of that system's specification; i.e., we can detect and isolate errors in the early stages of the software development.

By using formal methods *appropriately* within the software development process of aerospace systems, developers gain the benefit of reducing overall development cost. In fact, costs tend to be increased early in the system lifecycle, but reduced later on at the coding, testing, and maintenance stages, where correction of errors is far more expensive. Due to their precise notation, formal methods help to capture requirements abstractly in a precise and unambiguous form and then, through a series of semantic steps, introduce design and implementation level detail.

Formal methods have been recognized as an important technique to help ensure quality of aerospace systems, where system failures can easily cause safety hazards. For example, for the development of the control software for the C130J Hercules II, Lockheed Martin applied a correctness-by-construction approach based on formal (SPARK) and semi-formal (Consortium Requirements Engineering) methods [5]. The results showed that this combination was sufficient to eliminate a large number of errors and brought Lockheed Martin significant dividends in terms of high quality and less costly software. Note that an important part of the success of this approach is due to the use of the appropriate formal language. The use of a light version of SPARK, where the complex and difficult-to-understand parts of the Ada language had been removed, allowed for a higher level of abstraction, reducing the overall system complexity.

So-called synchronous languages [6] are formal languages dedicated to the programming of reactive systems. Synchronous languages (e.g., Lustre) were successfully applied in the development of automatic control software for critical applications like the software for nuclear plants, Airbus airplanes and the fight control software for Rafale fighters [7].

R2D2C (Requirements-to-Design-to-Code) [8] is a NASA approach to the engineering of complex computer systems where the need for correctness of the system, with respect to its requirements, is particularly high. This category includes NASA mission software, most of which exhibits both *autonomous* and *autonomic properties*. The approach embodies the main idea of *requirements-based programming* [9] and offers not only an underlying formalism, but also full formal development from requirements capture through to automatic generation of provably correct code. Moreover, the approach can be adapted to generate instructions in formats other than conventional programming languages—for example, instructions for controlling a physical device, or rules embodying the knowledge contained in an expert system. In these contexts, NASA has applied the approach to the verification of the instructions and procedures to be generated by the Hubble Space Telescope Robotic Servicing Missions and in the validation of the rule base used in the ground control of the ACE spacecraft [10].

3.4 Abstraction

The software engineering community recognizes abstraction as one of the best means of emphasizing important system aspects, thus helping to drive out unnecessary complexity and to come up with better solutions. According to James Rumbaugh, abstraction presents a selective examination of certain system aspects with the goal of emphasizing those aspects considered important for some purpose and suppressing the unimportant ones [11]. Designers of aerospace systems, shall consider abstraction provided by formal methods. Usually, aerospace software projects start with understanding the basic concepts of operations and requirements gathering, which results into a set of informal requirements (see Figure 1).

Once these requirements are documented, they can be formalized, e.g., with Lustre or R2D2C (see Section 3.3). The next step will be to describe the design in more detail. This is to specify how the desired software system is going to operate. Just as Java and C++ are high-level programming languages, in the sense that they are typed and structured, the formal languages dedicated to aerospace are structured and domain-specific and thus, they provide high-level structures to emphasize on the important properties of the system in question.

3.5 Decomposition and Modularity

In general, a complex aerospace system is a combination of distributed and heterogeneous components. Often, the task of modeling an aerospace system is about decomposition and modularity. Decomposition and modularity are well known concepts, which are the fundamentals of software engineering methods. Decomposition is an abstraction technique where we start with a high-level depiction of the system and create low-level abstractions of the latter, where features and functions fit together. Note that both high-level and low-level abstractions should be defined explicitly by the designer and that the low-level abstractions eventually result into components. This kind of modularity is based on explicitly-assigned functions to components, thus reducing the design effort and complexity. The design of complex systems always requires multiple decompositions.

3.6 Separation of Concerns

This Section describes a methodological approach to designing aerospace systems along the lines of the *separation-of-concerns* idea—one of the remarkable means of complexity reduction. This methodology strives to optimize the development process at its various stages and it has proven its efficiency in the hardware design and the software engineering of complex aerospace systems. As we have seen in Section 3.3, complex aerospace systems are composed of interacting components, where the separation-of-concerns methodology provides separate design "concerns" that (i) focus on complementary aspects of the component-based

system design, and (ii) have a systematic way of composing individual components into a larger system. Thus, a fundamental insight is that the design concerns can be divided into four groups: *component behavior*, *inter-component interaction*, *component integration*, and *system-level interaction*. This makes it possible to hierarchically design a system by defining different models for the behavior, interaction, and integration of the components.

- *component behavior*: This is the core of system functionality, and is about the computation processes with the single components that provides the real added value to the overall system.
- *inter-component interaction*: This concern might be divided into three subgroups:
    - *communication*: brings data to the computational components that require it, with the right quality of service, i.e., time, bandwidth, latency, accuracy, priority, etc.;
    - *connection*: this is the responsibility of system designers to specify which components should communicate with each other;
    - *coordination*: determines how the activities of all components in the system should work together.
- *component integration*: Addresses the concept of efficient matching of the various design elements of an aerospace system into the most efficient way possible. Component integration is typically a series of multidisciplinary design optimization activities that involve component behavior and inter-component interaction concerns. To provide this capability, the design environment (and often the aerospace system itself) incorporates a mechanism that automates component retrieval, adaptation, and integration. Component integration may also require *configuration* (or re-configuration) which is about giving concrete values to the provided component-specific parameters, e.g., tuning control or estimation gains, determining communication channels and their inter-component interaction policies, providing hardware and software resources and taking care of their appropriate allocation, etc.
- *system-level interaction*: For an aerospace system, the interaction between the system and its environment (including human users) becomes an integral part of the computing process.

The clear distinction between the concerns allows for a much better perception and understanding of the system's features and consequently the design of the same. Separating behavior from interaction is essential in reconciling the disparity between concerns, but it may lead aerospace system designers to make a wrong conclusion: intended component behaviors can be designed in isolation from their intended interaction models.

3.7 Requirements-Based Programming

Requirements-Based Programming (RBP) has been advocated [9] as a viable means of developing complex, evolving systems. It embodies the idea that requirements can be systematically and mechanically transformed into executable code. Generating code directly from requirements would enable software development to better accommodate the ever increasing demands on systems. In addition to increased software development productivity through eliminating manual efforts in the coding phase of the software lifecycle, RBP can also increase the quality of generated systems by automatically performing verification on the software—if the transformation is based on the formal foundations of computing. This may seem to be an obvious goal in the engineering of aerospace software systems, but RBP does in fact go a step further than current development methods. System development typically assumes the existence of a model of reality (design or, more precisely, a design specification), from which an implementation will be derived.

4 Designing Unmanned Space Systems

Space poses numerous hazards and harsh conditions, which makes it a very hostile place for humans. Without risking human lives, robotic technology such as robotic missions, automatic probes and unmanned observatories allow for space exploration. Unmanned space exploration poses numerous technological challenges. This is basically due to the fact that unmanned missions are intended to explore places where no man has gone before and thus, such missions must deal, often autonomously and with no human control, with unknown factors, risks, events and uncertainties.

4.1 Intelligent Agents

Both autonomy and artificial intelligence lay the basis for unmanned space systems. So-called “intelligent agents” [12] provide for the ability of space systems to act without human intervention. An agent can be viewed as perceiving its environment through sensors and acting upon that environment through effectors (see Figure 2). Therefore, in addition to the requirements traditional for an aerospace system such as reliability and safety, when designing an agent-based space system, we must also tackle issues related to agent-environment communication.

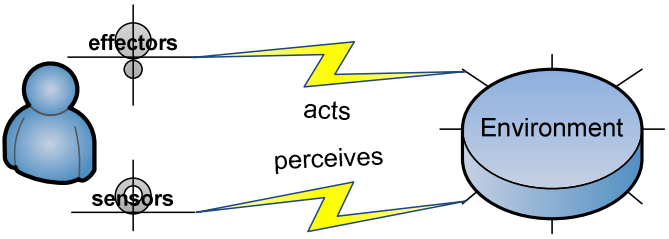

Fig. 2 Agent-Environment Relationship

Therefore, to design efficiently, we must consider the operational environment, because it plays a crucial role in an agent's behavior. There are a few important classes of environment that must be considered in order to properly design the agent-environment communication. The agent environment can be:

- *fully observable* (vs. *partially observable*) – the agent's sensors sense the complete state of the environment at each point in time;
- *deterministic* (vs. *stochastic*) – the next state of the environment is completely determined by the current state and the action executed by the agent;
- *episodic* (vs. *sequential*) – the agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself;
- *static* (vs. *dynamic*) – the environment is unchanged while an agent is deliberating (the environment is semi-dynamic if it does not change with the passage of time but the agent's performance does);
- *discrete* (vs. *continuous*) – an agent relies on a limited number of clearly defined distinct environment properties and actions;
- *single-agent* (vs. *multi-agent*) – there is only one agent operating in the environment.

As we have mentioned above, space systems are often regarded as *multi-agent systems*, where many intelligent agents interact with each other. These agents are considered to be *autonomous entities* that interact either cooperatively or non-cooperatively (on a selfish base). A popular multi-agent system approach is the so-called *intelligent swarms*. Conceptually, a swarm-based system consists of many simple entities (agents) that are independent but grouped as a whole appear to be highly organized. Without a centralized supervision, but due to simple local interactions and interactions with the environment, the swarm systems expose complex behavior emerging from the simple microscopic behavior of their members.

4.2 Autonomic Systems

The aerospace industry is currently approaching *autonomic computing* (AC) recognizing in its paradigm a valuable approach to the development of single intelligent agents and whole spacecraft systems capable of self-management. In general, AC is considered a potential solution to the problem of increasing system complexity and costs of maintenance. AC proposes a multi-agent architectural approach to large-scale computing systems, where the agents are special autonomic elements (AEs) [13, 14]. The "Vision of Autonomic Computing" [14] defines AEs as components that manage their own behavior in accordance with policies, and interact with other AEs to provide or consume computational services.

4.2.1 Self-management

An autonomic system (AS) is designed around the idea of self-management, which traditionally results into designing four basic policies (objectives) - *self-configuring*, *self-healing*, *self-optimizing*, and *self-protecting* (often termed as self-CHOP policies). In addition, in order to achieve these self-managing objectives, an autonomic system (AS) must constitute the following features:

- *self-awareness* – aware of its internal state;
- *self-situation* – environment awareness, situation and context awareness;
- *self-monitoring* – able to monitor its internal components;
- *self-adjusting* – able to adapt to the changes that may occur.

Both objectives (policies) and features (attributes) form generic properties applicable to any AS. Essentially, AS objectives could be considered as system requirements, while AS attributes could be considered as guidelines identifying basic implementation mechanisms.

4.2.2 Autonomic Element

An AS might be decomposed (see Section 3.5) into AEs (*autonomic elements*). In general, an AE extends programming elements (i.e., objects, components, services) to define a self-contained software unit (design module) with specified interfaces and explicit context dependencies. Essentially, an AE encapsulates rules, constraints and mechanisms for self-management, and can dynamically interact with other AEs. As stated in the IBM Blueprint [13], the core of the AEs is a special *control loop*. The latter is a set of functionally related units: *monitor*, *analyzer*, *planner*, and *executor*, all of them sharing knowledge (see Figure 3). A basic control loop is composed of a *managed element* (also called a *managed resource*) and a *controller* (called *autonomic manager*). The autonomic manager makes decisions and controls the managed resource based on measurements and events.

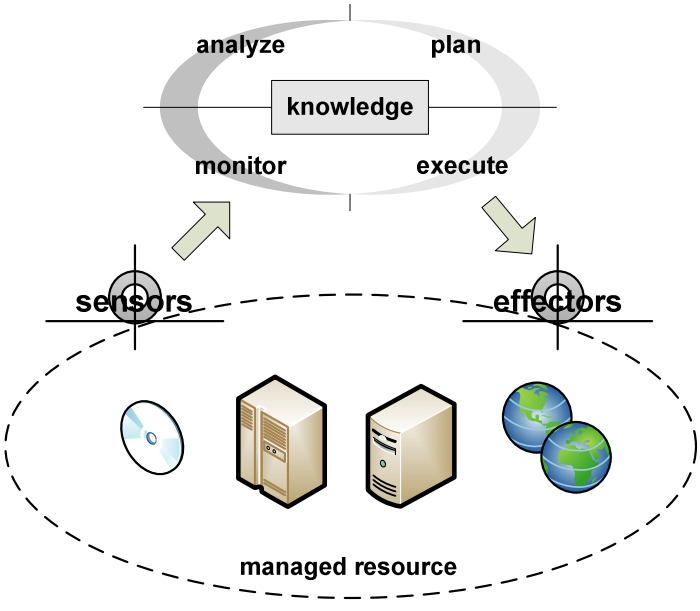

Fig. 3 AE Control Loop and Managed Resource

4.2.3 Awareness

Awareness is a concept playing a crucial role in ASs. Conceptually, awareness is a product of knowledge processing and monitoring. The AC paradigm addresses two kinds of awareness in ASs [13]:

- *self-awareness* – a system (or a system component) has detailed knowledge about its own entities, current states, capacity and capabilities, physical connections and ownership relations with other (similar) systems in its environment;
- *context-awareness* – a system (or a system component) knows how to negotiate, communicate and interact with environmental systems (or other components of a system) and how to anticipate environmental system states, situations and changes.

4.2.4 Autonomic Systems Design Principles

Although recognized as a valuable approach to unmanned spacecraft systems, the aerospace industry (NASA, ESA) does not currently employ any development approaches that facilitate the development of autonomic features. Instead, the development process of autonomic components and systems is identical to the one of traditional software systems (see Figure 1), thus causing inherent difficulties in:

- expressing autonomy requirements;
- designing and implementing autonomic features;
- efficiently testing autonomic behavior.

For example, the experience of developing autonomic components for ESA's ExoMars [15] has shown that the traditional development approaches do not cope well with the non-deterministic behavior of the autonomic elements – a proper testing requires a huge number of test cases. The following is a short overview of aspects and features that are needed to be addressed by an AS design.

Self-* Requirements. Like any other contemporary computer systems, ASs also need to fulfill specific functional and non-functional requirements (e.g., safety requirements). However, unlike other systems, the development of an AS is driven by the self-management objectives and attributes (see Section 4.2.1) that must be implemented by that very system. Such properties introduce special requirements, which we term *self-* requirements*. Note that self-management requires 1) *self-diagnosis* to analyze a problem situation and to determine a diagnosis, and 2) *self-adaptation* to repair the discovered faults. The ability of a system to perform adequate self-diagnosis depends largely on the quality and quantity of its knowledge of its current state, i.e., on the system *awareness* (see Section 4.2.3).

Knowledge. In general, an AS is intended to possess awareness capabilities based on well-structured *knowledge* and algorithms operating over the same. Therefore, *knowledge representation* is one of the important design activities of developing ASs. Knowledge helps ASs achieve awareness and autonomic behavior, where the more knowledgeable systems are, the closer we get to real intelligent systems.

Adaptability. The core concept behind adaptability is the general ability to change a system's observable behavior, structure, or realization. This requirement is amplified by self-adaptation (or *automatic adaptation*). Self-adaptation enables a system to decide on-the-fly about an adaptation on its own, in contrast to an ordinary adaptation, which is explicitly decided and triggered by the system's environment (e.g., a user or administrator). Adaptation may result to changes in some *functionality*, *algorithms* or *system parameters* as well as the *system's structure* or any other aspect of the system. If an adaptation leads to a change of the complete system model, including the model that actually decides on the adaptation, this system is called a *totally reconfigurable system*. Note that self-adaptation requires a model of the system's environment. (often referred to as *context*) and therefore, self-adaptation may be also called *context adaptation*.

Monitoring. Since monitoring is often regarded as a prerequisite for awareness, it constitutes a subset of awareness. For ASs, monitoring (often referred to as self-monitoring) is the process of obtaining knowledge through a collection of sensors instrumented within the AS in question. Note that monitoring is not responsible for diagnostic reasoning or adaptation tasks. One of the main challenges of

monitoring is to determine which information is most crucial for analysis of a system's behavior, and when. The notion of monitoring is closely related to the notion of context. Context embraces the *system state*, its *environment*, and any *information relevant to the adaptation*. Consequently, it is also a matter of context, which information indicates an erroneous system state and hence characterizes a situation in which a certain adaptation is necessary. In this case, adaptation can be compared to error handling, as it transfers the system from an erroneous (unwanted) system state to a well-defined (wanted) system state.

Dynamicity. Dynamicity embraces the system ability to change at runtime. In contrast to adaptability this only constitutes the technical facility of change. While adaptability refers to the conceptual change of certain system aspects, which does not necessarily imply the change of components or services, dynamicity is about the technical ability to remove, add or exchange services and components. There is a close but not dependable relation between both dynamicity and adaptability. Dynamicity may also include a system ability to exchange certain (defective or obsolete) components without changing the observable behavior. Conceptually, dynamicity deals with concerns like preserving states during functionality change, starting, stopping and restarting system functions, etc.

Autonomy. As the term Autonomic Computing already suggests, autonomy is one of the essential characteristics of ASs. AC aims at freeing human administrators from complex tasks, which typically requires a lot of decision making without human intervention (and thus without direct human interaction). Autonomy, however, is not only intelligent behavior but also an organizational manner. Context adaptation is not possible without a certain degree of autonomy. Here, the design and implementation of the AE *control loop* (see Section 4.2.2) is of vital importance for autonomy. A rule engine obeying a predefined set of conditional statements (e.g., if-then-else) put in an endless loop is the simplest form of control loop's implementation. In many cases, such a simple- rule-based mechanism however may not be sufficient. In such cases, the control loop should facilitate force feedback learning and learning by observation to refine the decisions concerning the priority of services and their granted QoS, respectively.

Robustness. Robustness is a requirement that is claimed for almost every system. ASs should benefit from robustness since this may facilitate the design of system parts that deal with self-healing and self-protecting. In addition, the system architecture could ease the appliance of measures in cases of errors and attacks. Robustness states the first and most obvious step on the road to dependable systems. Beside a special focus on error avoidance, several requirements aiming at correcting errors should also be forced. Robustness could be often achieved by decoupling and asynchronous communication, e.g., between interacting AEs (autonomic elements). Error avoidance, error prevention, and fault tolerance are approved techniques in software engineering, which shall help us in preventing from error propagation when designing ASs.

Mobility. Mobility enfolds all parts of the system: from mobility of code on the lowest granularity level via mobility of services or components up to mobility of devices or even mobility of the overall system. Mobility enables dynamic discovery and usage of new resources, recovery of crucial functionalities, etc. Often, mobile devices are used for detection and analysis of problems. For example, AEs may rely on mobility of code to transfer some functionality relevant for security updates or other self-management issues.

Traceability. Traceability enables the unambiguous mapping of the logical onto the physical system architecture, thus facilitating both system implementation and deployment. The deployment of system updates is usually automatic and thus, it requires traceability. Traceability is additionally helpful when analyzing the reasons for wrong decisions made by the system.

4.2.5 Formalism for Autonomic Systems

ASs are special computer systems that emphasize *self-management* through *context* and *self-awareness* [13, 14]. Therefore, an AC formalism should not only provide a means of description of system behavior but also should tackle the issues vital for autonomic systems self-management and awareness. Moreover, an AC formalism should provide a well-defined semantics that makes the AC specifications a base from which developers may design, implement, and verify ASs (including autonomic aerospace components or systems).

ASSL (Autonomic System Specification Language) [16] is a *declarative* specification language for ASs with well-defined semantics. It implements modern programming language concepts and constructs like inheritance, modularity, type system, and high abstract expressiveness. Being a formal language designed explicitly for specifying ASs, ASSL copes well with many of the AS aspects (see Section 4.2). Moreover, specifications written in ASSL present a view of the system under consideration, where specification and design are intertwined. Conceptually, ASSL is defined through *formalization tiers* [16]. Over these tiers, ASSL provides a *multi-tier specification model* that is designed to be scalable and exposes a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. ASSL defines ASs with special *self-managing policies*, *interaction protocols*, and *autonomic elements*. As a formal language, ASSL defines a neutral, implementation-independent representation for ASs. Similar to many formal notations, ASSL enriches the underlying logic with modern programming concepts and constructs thereby increasing the expressiveness of the formal language while retaining the precise semantics of the underlying logic.

The authors of this paper have successfully used ASSL to design and implement autonomic features for part of NASA's ANTS (Autonomous Nano-Technology Swarm) concept mission [17].

5 Conclusions

We have presented key *fundamentals* in designing complex aerospace software systems. By relying on our experience, we have discussed best design practices that can be used as guidelines by software engineers to build their own design strategy directing them towards the "right design concept" that can be applied to design a reliable aerospace system meeting the important safety requirements. Moreover, we have talked about design principles and the application of formal methods in the aerospace industry. Finally, we have shown the tremendous advantage of the so-called ASs (autonomic systems). ASs offer a solution for unmanned spacecraft systems, because the former are capable of self-adaptation, thus increasing the reliability of the unmanned systems where human intervention is not feasible or impractical. Although recognized as a valuable approach to unmanned spacecraft systems, the aerospace industry (NASA, ESA) does not currently employ any development approaches that facilitate the development of autonomic features. This makes both the implementation and testing of such features hardly feasible. We have given a short overview of aspects and features that are needed to be addressed by an AS design in order to make such a design efficient. To design and implement efficient ASs (including autonomic aerospace systems) we need AC-dedicated frameworks and tools. ASSL (Autonomic System Specification Language) is such a formal method, which we have successfully used at Lero—the Irish Software Engineering Research Centre, to develop autonomic features for a variety of systems, including NASA's ANTS (Autonomous Nano-Technology Swarm) prospective mission.

Acknowledgment. This work was supported in part by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre.

References

1. Gilbert, D., Aparicio, M., Atkinson, B., Brady, S., Ciccarino, J., Grosof, B., O'Connor, P., Osisek, D., Pritko, S., Spagna, R., Wilson, L.: IBM Intelligent Agent Strategy. White Paper, IBM Corporation (1995)
2. Philippe, C.: Verification, Validation, and Certification Challenges for Control Systems. In: Samad, T., Annaswamy, A.M. (eds.) The Impact of Control Technology. IEEE Control Systems Society (2011)
3. Herrmann, D.S.: Software Safety and Reliability. IEEE Computer Society Press, Los Alamitos (1999)
4. NASA-STD-8719.13A: Software Safety. NASA Technical Standard (1997)
5. Amey, P.: Correctness By Construction: Better Can Also Be Cheaper. CrossTalk Magazine. The Journal of Defense Software Engineering (2002)
6. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, Boston (1993)
7. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., De Simone, R.: The Synchronous Languages Twelve Years Later. Proceedings of the IEEE 91(1), 64–83 (2003)

8. Hinchey, M.G., Rash, J.L., Rouff, C.A.: Requirements to Design to Code: Towards a Fully Formal Approach to Automatic Code Generation. Technical Report TM-2005-212774, NASA Goddard Space Flight Center, Greenbelt, MD, USA (2004)
9. Harel, D.: From Play-In Scenarios To Code: An Achievable Dream. IEEE Computer 34(1), 53–60 (2001)
10. ACE Spacecraft, Astrophysics Science Division at NASA's GSFC (2005), http://helios.gsfc.nasa.gov/ace_spacecraft.html
11. Blaha, M., Rumbaugh, J.: Object-Oriented Modeling and Design with UML, 2nd edn. Pearson, Prentice Hall, New Jersey (2005)
12. Gilbert, D., Aparicio, M., Atkinson, B., Brady, S., Ciccarino, J., Grosof, B., O'Connor, P., Osisek, D., Pritko, S., Spagna, R., Wilson, L.: IBM Intelligent Agent Strategy. White Paper, IBM Corporation (1995)
13. IBM Corporation: An architectural blueprint for autonomic computing, 4th edn. White paper, IBM Corporation (2006)
14. Kephart, J.O., Chess, D.M.: The vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)
15. ESA: Robotic Exploration of Mars, http://www.esa.int/esaMI/Aurora/SEM1NVZKQAD_0.html
16. Vassev, E.: ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing, Germany (2009)
17. Truszkowski, M., Hinchey, M., Rash, J., Rouff, C.: NASA's swarm missions: The challenge of building autonomous software. IT Professional 6(5), 47–52 (2004)