ISBN 978-83-60810-14-9
ISSN 1896-7094

You Can't Get There From Here! Problems and Potential Solutions in Developing New Classes of Complex Computer Systems

Michael G. Hinchey
Lero—the Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
mike.hinchey@lero.ie

James L. Rash, Walter F. Truszkowski
NASA Goddard Space Flight Center
Systems Engineering Division
Greenbelt, MD 21035 USA
{james.l.rash, walter.f.truszkowski}@nasa.gov

Roy Sterritt
School of Computing and Mathematics
University of Ulster
Northern Ireland
r.sterritt@ulster.ac.uk

Christopher A. Rouff
Lockheed Martin Advanced Technology Laboratories
Arlington, VA 22203 USA
christopher.rouff@lmco.com

***Abstract*—The explosion of capabilities and new products within the sphere of Information Technology (IT) has fostered widespread, overly optimistic opinions regarding the industry, based on common but unjustified assumptions of quality and correctness of software. These assumptions are encouraged by software producers and vendors, who at this late date have not succeeded in finding a way to overcome the lack of an automated, mathematically sound way to develop correct systems from requirements. NASA faces this dilemma as it envisages advanced mission concepts that involve large swarms of small spacecraft that will engage cooperatively to achieve science goals. Such missions entail levels of complexity that beg for new methods for system development far beyond today's methods, which are inadequate for ensuring correct behavior of large numbers of interacting intelligent mission elements. New system development techniques recently devised through NASA-led research will offer some innovative approaches to achieving correctness in complex system development, including autonomous swarm missions that exhibit emergent behavior, as well as general software products created by the computing industry.**

I. Introduction

SOFTWARE has become pervasive. We encounter it in our everyday lives: the average electric razor contains the equivalent of more than 100,000 lines of code, several high-end cars contain more software than the onboard systems of the Space Shuttle. We are reliant on software for our transportation and entertainment, to wash our clothes and cook our meals, and to keep us in touch with the outside world via the Internet and our mobile phones.

The Information Technology industry, driven by software development, has made remarkable advances. In just over half a century, it has developed into a trillion-dollar-per-year industry, continually breaking its own records [17], [27].

Some breathtaking statistics have been reported for the hardware and software industries [16], [46]:

- The Price-to-Performance ratio halves every 18 months, with a 100-fold increase in performance every decade.
- Performance progress in the next 18 months will equal *all* progress made to date.
- New storage available equals the sum of all previously available storage *ever*.
- New processing capability equals the sum of all previous processing power.

Simultaneously, a number of flawed assumptions have arisen regarding the way we build both software and hardware systems [38], [46], which include:

- Human beings can achieve perfection; they can avoid making mistakes during installation, maintenance and upgrades.
- Software will eventually be bug-free; the focus of companies has been to hire better programmers, and the focus of universities is to better train software engineers in development lifecycle models.
- Mean-time between failure (MTBF) is already very large (approximately 100 years) and will continue to increase.
- Maintenance costs are a function of the purchase price

This paper was previously published in Proc. Eighth International Conference on Integrated Design and Process Technology (IDPT), 2005. Reprinted with permission.

This work is funded in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (www.lero.ie); by the NASA Office of Safety and Mission Assurance, under its Software Assurance Research Program project *Formal Approaches to Swarm Technologies* (FAST), administered by the NASA IV&V Facility; by the Office of Technology Transfer, NASA Goddard Space Flight Center; by the NASA Software Engineering Laboratory, NASA Goddard Space Flight Center; and by the University of Ulster Computer Science Research Institute and the Centre for Software Process Technologies (CSPT), funded by Invest NI through the Centres of Excellence Programme under the European Union Peace II initiative.

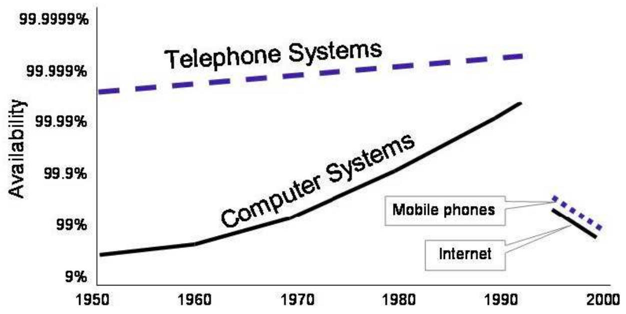

Fig. 1. Contrasting availability of Telephone Systems, Computer Systems, Internet, and Mobile Phones.

of hardware; and, as such, decreasing hardware costs (price/performance) results in decreases in maintenance costs.

II. Software Problems

With the situation stated this way, many flawed assumptions regarding the IT industry come into view. The situation is even worse if we focus primarily on software. The Computing industry has failed to avoid software-related catastrophes. Notable examples include:

- Therac-25, where cancer patients were given lethal doses of radiation during radiation therapy [33].
- Ariane 5, where it was assumed that the same launch software used in the prior version (Ariane 4) could be reused. The result was the loss of the rocket within seconds of launch [34].
- The Mars Polar Lander, where failure to initialize a variable resulted in the craft crash landing on the Martian surface, instead of reverse thrusting and landing softly [29].

Progress in software regularly lags behind hardware. In the last decade, for example, two highly software-intensive applications, namely Internet communications and mobile phone technology, have suffered reduced availability and increased *down time*, while their hardware counterparts, computer hardware and telephony systems, have continued to improve. Figure 1 illustrates this trend [17].

A. An Historic Problem

The realization that software development has lagged greatly behind hardware is hardly a new one [6], nor is the realization that our software development processes have some severe deficiencies.

Brooks, in a widely quoted and much-referenced article [7], warns of complacency in software development. He stresses that, unlike hardware development, we cannot expect to achieve great advances in productivity in software development unless we concentrate on more appropriate development methods. He highlights how software systems can suddenly turn from being well-behaved to behaving erratically and uncontrollably, with unanticipated delays and increased costs. Brooks sees software systems as “werewolves” and rightly points out that there is no single technique, no Silver Bullet, capable of slaying such monsters [6].

On the contrary, more and more complex systems are run on highly distributed, heterogeneous networks, subject to strict performance, fault tolerance, and security constraints, all of which may conflict. Many engineering disciplines must contribute to the development of complex systems in an attempt to satisfy all of these requirements. No single technique is adequate to address all issues of complex system development; rather, different techniques must be applied at different stages of development (and throughout the development process) to ensure unambiguous requirements statements, precise specifications that are amenable to analysis and evaluation, implementations that satisfy the requirements and various (often conflicting) goals, re-use, re-engineering and reverse engineering of legacy code, appropriate integration with existing systems, ease-of-use, predictability, dependability, maintainability, fault tolerance, etc. [6].

Brooks [7] differentiates between the *essence* (that is, problems that are necessarily inherent in the nature of software) and *accidents* (that is, problems that are secondary and caused by current development environments and techniques). He points out the great need for appropriate means of coming to grips with the conceptual difficulties of software development—that is, for appropriate emphasis on specification and design, rather than on coding and testing.

In his article [7], he highlights some successes that have been achieved in gaining improvements in productivity, but points out that these address problems in the current development process, rather than the problems inherent in software itself. In this category, he includes: the advent of high-level programming languages, time-sharing, and unified programming environments. Object-oriented programming, techniques from artificial intelligence, expert systems, automatic programming, program verification, and the advent of workstations, he sees as non-bullets, as they will not help in slaying the werewolf.

He sees software reuse, rapid prototyping, incremental development, and the employment of top-class designers as potential starting points for the Silver Bullet, but warns that none in itself is sufficient.

Brooks’ article has been very influential, and remains one of the classics of software engineering. His viewpoint has been criticized, however, as being overly pessimistic and for failing to acknowledge some promising developments [6].

Harel, in an equally influential paper, written as a rebuttal to Brooks [19], points to developments in Computer-Aided Software Engineering (CASE) and visual formalisms [18] as potential *bullets*. Harel’s view is far more optimistic. He writes five years after Brooks, and has seen the developments in that period. The last forty years of system development have been equally difficult, according to Harel, and, using a conceptual vanilla framework, the development community has devised means of overcoming many difficulties. As we address more complex systems, Harel argues that we must devise similar frameworks that are applicable to the classes of system we are developing.

Harel, along with many others, including the authors of this paper, believes that appropriate techniques for modeling must have a rigorous mathematical semantics, and appropriate means for representing constructs. This differs greatly from Brooks, who sees representational issues as mainly *accidental*.

III. New Challenges for Software Engineering

Clearly there have been significant advances in software engineering tools, techniques, and methods, since the time of Brooks' and Harel's papers. In many cases, however, the advantages of these developments have been mitigated by corresponding increases in demand for greater, more complex functionality, stricter constraints on performance and reaction times, and attempts to increase productivity and reduce costs, while simultaneously pushing systems requirements to their limits. NASA, for example, continues to build more and more complex systems, with impressive functionality, and increasingly autonomous behavior. In the main, this is essential. NASA missions are pursuing scientific discovery in ways that require autonomous systems. While manned exploration missions are clearly in NASA's future (such as the Exploration Initiative's plans to return to the moon and put Man on Mars), several current and future NASA missions, for reasons that we will explain below, necessitate autonomous behavior by unmanned spacecraft.

We will describe some of the challenges for software engineering emerging from new classes of complex systems being developed by NASA and others. We will discuss these in Section III-A with reference to a NASA concept mission that is exemplary of many of these new systems. Then, in Section IV we will present some techniques that we are addressing, which may lead towards a Silver Bullet.

A. *Challenges of Future NASA Missions*

Future NASA missions will exploit new paradigms for space exploration, heavily focused on the (still) emerging technologies of autonomous and autonomic systems. Traditional missions, reliant on one large spacecraft, are being superceded or complemented by missions that involve several smaller spacecraft operating in collaboration, analogous to swarms in nature. This offers several advantages: the ability to send spacecraft to explore regions of space where traditional craft simply would be impractical, increased spatial distribution of observations, greater redundancy, and, consequently, greater protection of assets, and reduced costs and risk, to name but a few. Planned missions entail the use of several unmanned autonomous vehicles (UAVs) flying approximately one meter above the surface of Mars, covering as much of the surface of Mars in seconds as the now famous Mars rovers did in their entire time on the planet; the use of armies of tetrahedral walkers to explore the Mars and Lunar surface; constellations of satellites flying in formation; and the use of miniaturized pico-class spacecraft to explore the asteroid belt.

These new approaches to exploration missions simultaneously pose many challenges. The missions will be unmanned and necessarily highly autonomous. They will also exhibit all of the classic properties of autonomic systems, being self-protecting, self-healing, self-configuring, and self-optimizing. Many of these missions will be sent to parts of the solar system where manned missions are simply not possible, and to where the round-trip delay for communications to spacecraft exceeds 40 minutes, meaning that the decisions on responses to problems and undesirable situations must be made *in situ* rather than from ground control on Earth.

Verification and Validation (V&V) for complex systems still poses a largely unmet challenge in the field of Computing, yet the challenge is magnified with increasing degrees of system autonomy. It is an even greater open question as to the extent to which V&V is feasible when the system possesses the ability to adapt and learn, particularly in environments that are dynamic and not specially constrained. Reliance on testing as the primary approach to V&V becomes untenable as systems move towards higher levels of complexity, autonomy, and adaptability in such environments. Swarm missions will fall into this category, and an early concern in the design and development of swarms will be the problem of predicting, or at least bounding, and controlling emergent behavior.

The result is that formal specification techniques and formal verification will play vital roles in the future development of NASA space exploration missions. The role of formal methods will be in the specification and analysis of forthcoming missions, enabling software assurance and proof of correctness of the behavior of these systems, whether or not this behavior is emergent (as a result of composing a number of interacting entities, producing behavior that was not foreseen). Formally derived models may also be used as the basis for automating the generation of much of the code for the mission. To address the challenge in verifying the above missions, a NASA project, Formal Approaches to Swarm Technology (FAST), is investigating the requirements of appropriate formal methods for use in such missions, and is beginning to apply these techniques to specifying and verifying parts of a future NASA swarm-based mission.

B. *ANTS: A NASA Concept Mission*

The Autonomous Nano-Technology Swarm (ANTS) mission will involve the launch of a swarm of autonomous pico-class (approximately 1kg) spacecraft that will explore the asteroid belt for asteroids with certain characteristics. Figure 2 gives an overview of the ANTS mission [47]. In this mission, a transport ship, launched from Earth, will travel to a point in space where gravitational forces on small objects (such as pico-class spacecraft) are all but negligible. Objects that remain near such a point (termed a Lagrangian point) are in a stable orbit about the Sun and will have a fixed geometrical relationship to the Sun-Earth system. From the transport ship positioned at such a point, 1000 spacecraft that have been assembled en route from Earth will be launched into the asteroid belt.

Because of their small size, each ANTS spacecraft will carry just one specialized instrument for collecting a specific type of data from asteroids in the belt. As a result, spacecraft

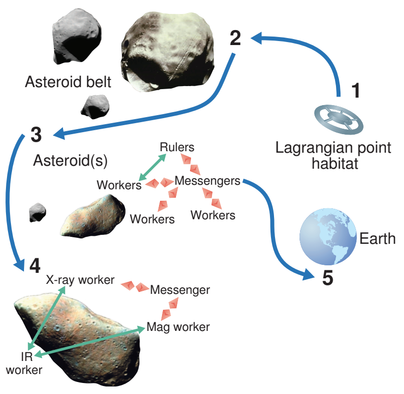

Fig. 2. NASA's Autonomous Nano Technology Swarm (ANTS) mission scenario.

must cooperate and coordinate using a hierarchical social behavior analogous to colonies or swarms of insects, with some spacecraft directing others. To implement this mission, a heuristic approach is being considered that provides for a social structure to the swarm based on the above hierarchy. Artificial intelligence technologies such as genetic algorithms, neural nets, fuzzy logic and on-board planners are being investigated to assist the mission to maintain a high level of autonomy. Crucial to the mission will be the ability to modify its operations autonomously to reflect the changing nature of the mission and the distance and low-bandwidth communications back to Earth.

Approximately 80 percent of the spacecraft will be workers that will carry the specialized instruments (e.g., a magnetometer, x-ray, gamma-ray, visible/IR, neutral mass spectrometer) and will obtain specific types of data. Some will be coordinators (called rulers) that have rules that decide the types of asteroids and data the mission is interested in, and that will coordinate the efforts of the workers. The third type of spacecraft are messengers that will coordinate communication between the rulers and workers, and communications with the Earth ground station, including requests for replacement spacecraft with specialized instruments as these are required. The swarm will form sub-swarms under the direction of a ruler, which contains models of the types of science that it wants to perform. The ruler will coordinate workers each of which uses its individual instrument to collect data on specific asteroids and feed this information back to the ruler who will determine which asteroids are worth examining further. If the data matches the profile of a type of asteroid that is of interest, an imaging spacecraft will be sent to the asteroid to ascertain the exact location and to create a rough model to be used by other spacecraft for maneuvering around the asteroid. Other teams of spacecraft will then coordinate to finish the mapping of the asteroid to form a complete model.

C. Problematic Issues

1) Size and Complexity: While the use of a swarm of miniature spacecraft is essential for the success of ANTS, it simultaneously poses several problems in terms of adding significantly to the complexity of the mission.

The mission will launch 1000 pico-class spacecraft, many of which possibly will be destroyed by collisions with asteroids, since the craft, having no means of maneuvering other than solar sails, will be very limited in their collision-avoidance capabilities. The several hundred surviving spacecraft must be organized into effective groups that will collect science data and make decisions as to which asteroids warrant further investigation. These surviving spacecraft effectively form a wireless sensor network [23] tens of millions of miles from Earth. The overhead for communications is clearly significant.

To keep the spacecraft small, each craft only carries a single instrument. That is why several craft must coordinate to investigate particular asteroids and collect different types of science data. Again, while miniaturization is important, the use of such a scheme has a major drawback: we have no *a priori* knowledge as to which instruments will be lost during normal operations (where we expect to regularly lose craft due to collisions).

The need to identify lost capabilities and instruments, and then replace them, presents an extremely complex problem. In the case of lost messengers and rulers, other craft may be *promoted* to replace them. It is merely the software that differentiates messengers and rulers from other workers, so mobile code serves to overcome this problem. When an instrument is lost, however, we have a rather different problem. A worker with a damaged instrument can be reserved for use as a ruler, and another spacecraft with an identical instrument can replace it.

An alternative would be add more features (instruments) into each spacecraft, but this would increase both their size (a problem in such a constrained environment) and their power requirements. The addition of features, of course, also increases complexity, as identified by Lawson [32].

2) Emergent Behavior: In swarm-based systems, interacting agents (often homogeneous or near homogeneous agents) are developed to take advantage of their emergent behavior. Each of the agents is given certain parameters that it tries to maximize. Bonabeau et al. [4], who studied self-organization in social insects, state that “complex collective behaviors may emerge from interactions among individuals that exhibit simple behaviors” and describe emergent behavior as “a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions among its lower-level components.”

Intelligent swarms [3] use swarms of simple intelligent agents. Swarms have no central controller: they are self-organizing based on the emergent behaviors of the simple interactions. There is no external force directing their behavior and no one agent has a global view of the intended macroscopic behavior. Though current NASA swarm missions differ from true swarms as described above, they do have many of the same attributes and may exhibit emergent behavior. In addition, there are a number of US government projects that are looking at true swarms to accomplish complex missions.

3) Autonomy: Autonomous operation is essential for the success of the ANTS mission concept.

Round trip communications delays of up to 40 minutes, and limited bandwidth on communications with Earth, mean that effective control from the ground station is impossible. Ground controllers would not be able to react sufficiently quickly during encounters with asteroids to avoid collisions with asteroids and even other ANTS spacecraft. Moreover, the delay in sending instructions to the spacecraft would be so great that situations would likely have changed dramatically by the time the instructions were received.

But autonomy implies absence of centralized control. Individual ANTS spacecraft will operate autonomously as part of a subgroup under the direction of that subgroup's *ruler*. That ruler will itself autonomously make decisions regarding asteroids of interest, and formulate plans for continuing the mission of collecting science data. The success of the mission is predicated on the validity of the plans generated by the rulers, and requires that the rulers generate sensible plans that will collect valid science data, and then make valid informed decisions.

That autonomy is possible is not in doubt. What is in doubt is that autonomous systems can be relied upon to operate correctly, in particular in the absence of a full and complete specification of what is required of the system.

4) Testing and Verification: As can be seen from the brief exposition above, ANTS is a highly complex system that poses many significant challenges. Not least amongst these are the complex interactions between heterogeneous components, the need for continuous re-planning, re-configuration, and re-optimization, the need for autonomous operation without intervention from Earth, and the need for assurance of the correct operation of the mission.

As mission software becomes increasingly more complex, it also becomes more difficult to test and find errors. Race conditions in these systems can rarely be found by inputting sample data and checking whether the results are correct. These types of errors are time-based and only occur when processes send or receive data at particular times, or in a particular sequence, or after learning occurs. To find these errors, the software processes involved have to be executed in all possible combinations of states (state space) that the processes could collectively be in. Because the state space is exponential (and sometimes factorial) to the number of states, it becomes untestable with a relatively small number of processes. Traditionally, to get around the state explosion problem, testers have artificially reduced the number of states of the system and approximated the underlying software using models.

One of the most challenging aspects of using swarms is how to verify that the emergent behavior of such systems will be proper and that no undesirable behaviors will occur. In addition to emergent behavior in swarms, there are also a large number of concurrent interactions between the agents that make up the swarms. These interactions can also contain errors, such as race conditions, that are very difficult to ascertain until they occur. Once they do occur, it can also be very difficult to recreate the errors since they are usually data and time dependent.

As part of the FAST project, NASA is investigating the use of formal methods and formal techniques for verification and validation of these classes of mission, and is beginning to apply these techniques to specifying and verifying parts of the ANTS concept mission. The role of formal methods will be in the specification and analysis of forthcoming missions, while offering the ability to perform software assurance and proof of correctness of the behavior of the swarm, whether this behavior is emergent or not.

IV. Some Potentially Useful Techniques

A. *Autonomicity*

Autonomy may be considered as bestowing the properties of self-governance and self-direction, i.e., control over one's goals [15], [26], [43]. Autonomicity is having the ability to self-manage through properties such as self-configuring, self-healing, self-optimizing, and self-protecting. These are achieved through other self-properties such as self-awareness (including environment awareness), self-monitoring, and self-adjusting [45].

Increasingly, self-management is seen as the only viable way forward to cope with the ever increasing complexity of systems. From one perspective, self-management may be considered a specialism of self-governance, i.e., autonomy where the goals/tasks are specific to management roles [46]. Yet from the wider context, an autonomic element (AE), consisting of an autonomic manager and managed component, may still have its own specific goals, but also the additional responsibility of management tasks particular to the wider system environment.

It is envisaged that in an autonomic environment the AEs communicate to ensure a managed environment that is reliable and fault tolerant and meets high level specified policies (where a policy consists of a set of behavioral constraints or preferences that influences the decisions made by an autonomic manager [10]) with an overarching vision of system-wide policy-based self-management. This may result in AEs monitoring or *watching out for* other AEs. In terms of autonomy and the concern of undesirable emergent behavior, an environment that dynamically and continuously monitors can assist in detecting race conditions and reconfiguring to avoid damage (self-protecting, self-healing, self-configuring, etc.). As such, autonomicity becoming mainstream in the industry can only assist in improving techniques, tools, and processes for autonomy [44].

B. *Hybrid Formal Methods*

The majority of formal notations currently available were developed in the 1970s and 1980s and reflect the types of distributed systems being developed at that time. Current distributed systems are evolving and may not be able to be specified in the same way that past systems have been developed. Because of this, it appears that many people are combining formal methods into integrated approaches to address some of the new features of distributed systems (e.g., mobile agents, swarms, and emergent behavior).

Integrated approaches have been very popular in specifying concurrent and agent-based systems. Integrated approaches often combine a process algebra or logic-based approach with a model-based approach. The process algebra or logic-based approach allows for easy specification of concurrent systems, while the model-based approach provides strength in specifying the algorithmic part of a system.

Some recent hybrid approaches include:

- CSP-OZ, a combination of CSP and Object-Z [11]
- Object-Z and Statecharts [8]
- Timed Communicating Object Z [13]
- Temporal B [5]
- Temporal Petri Nets (Temporal Logic and Petri Nets) [1]
- ZCCS, a combination of Z and CCS [14]

These and new hybrid formal methods are being investigated to address swarm and other complex NASA missions [41].

C. Automatic Programming

For many years, automatic programming has referred, primarily, to the use of very high-level languages to describe solutions to problems, which could then be translated down and expressed as code in more traditional (lower level) programming languages. Parnas [36] implies that the term is glamorous, rather than having any real meaning, precisely because it is the solution that is being specified rather than the problem that must be solved. Brooks [7] supports this view, and equally criticizes the field of visual programming, arguing that it will never produce anything of value.

Writing just five years after Brooks, Harel [19] disagrees, faulting Brooks for failing to recognize advances in *visual formalisms*. Now, writing almost two decades after Brooks, we argue that automatic code generation is not only a viable option, it is essential to the development of the classes of complex system we are discussing here, and as exemplified by ANTS.

Autonomous and autonomic systems, exhibiting complex emergent behavior, cannot, in general, be fully specified at the outset. The roles and behaviors of the system will vary greatly over time. While we may try to write specifications that constrain the system, it is clear that not all behavior can be specified in advance. Consequently, the classes of system we are discussing will often require that code is generated, or modified, during execution. As a result, the classes of system we are describing here will *require* automatic code generation.

Several tools already exist that successfully generate code from a given model. Unfortunately, many of these tools have been shown to generate code, portions of which are never executed, or portions of which cannot be justified from either the requirements or the model. Moreover, existing tools do not and cannot overcome the fundamental inadequacy of all currently available automated development approaches, which is that they include no means to establish a provable equivalence between the requirements stated at the outset and either the model or the code they generate.

Traditional approaches to automatic code generation, including those embodied in commercial products such as Matlab [35], in system development toolsets such as the B-Toolkit [31] or the VDM++ toolkit [28], or in academic research projects, presuppose the existence of an explicit (formal) model of reality that can be used as the basis for subsequent code generation. While such an approach is reasonable, the advantages and disadvantages of the various modeling approaches used in computing are well known and certain models can serve well to highlight certain issues while suppressing other less relevant details [37]. It is clear that the converse is also true. Certain models of reality, while successfully detailing many of the issues of interest to developers, can fail to capture some important issues, or perhaps even the most important issues.

That is why, we believe, future approaches to automatic code generation must be based on Formal Requirements-Based Programming [39].

D. Formal Requirements Based Programming

Requirements-Based Programming refers to the development of complex software (and other) systems, where each stage of the development is fully traceable back to the requirements given at the outset. In essence, Requirements-Based Programming takes Model-Based Development and adds a *front end* [40].

The difference is that Model-Based Development holds that emphasis should be placed on building a model of the system with such high quality that automatic code generation is viable. While this has worked well, and made automatic code generation feasible, there is still the large *analysis-specification* gap that remains unaddressed. Requirements-Based Programming addresses that issue and ensures that there is a direct mapping from requirements to design, and that this design (model) may then be used as the basis for automatic code generation.

There have been calls for the community to address Requirements-Based Programming, as it offers perhaps the most promising approach to achieving *correct* systems [20]. Although the use of Requirements-Based Programming does not specifically presuppose the existence of an underlying formalism, the realization that proof of correctness is not possible without formalism [2] certainly implies that Requirements-Based Programming should be formal.

In fact, Formal Requirements-Based Programming, coupled with a graphical representation for system requirements (e.g., UML use cases) possesses the features and advantages of a visual formalism described by Harel [18].

1) R2D2C: R2D2C, or Requirements-to-Design-to-Code [22], [39], is a NASA patent-pending approach to Requirements-Based Programming.

In R2D2C, engineers (or others) may write specifications as scenarios in constrained (domain-specific) natural language, or in a range of other notations (including UML use cases). These will be used to derive a formal model (Figure 3) that is

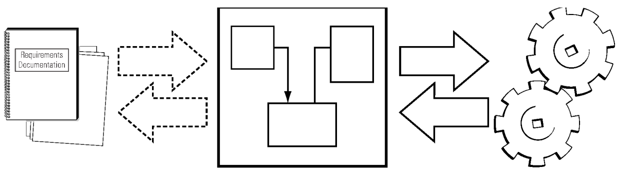

Fig. 3. The R2D2C approach, generating a formal model from requirements and producing code from the formal model, with automatic reverse engineering.

guaranteed to be equivalent to the requirements stated at the outset, and which will subsequently be used as a basis for code generation. The formal model can be expressed using a variety of formal methods. Currently we are using CSP, Hoare's language of Communicating Sequential Processes [24], [25], which is suitable for various types of analysis and investigation, and as the basis for fully formal implementations as well as for use in automated test case generation, etc.

R2D2C is unique in that it allows for full formal development from the outset, and maintains mathematical soundness through all phases of the development process, from requirements through to automatic code generation. The approach may also be used for reverse engineering, that is, in retrieving models and formal specifications from existing code, as shown in Figure 3. The approach can also be used to "paraphrase" (in natural language, etc.) formal descriptions of existing systems. In addition, the approach is not limited to generating high-level code. It may also be used to generate business processes and procedures, and we have been experimenting with using it to generate instructions for robotic devices that were to be used on the Hubble Robotic Servicing Mission (HRSM), which, at the time of writing, has not received a final go-ahead. We are also experimenting with using it as a basis for an expert system verification tool, and as a means of capturing domain knowledge for expert systems.

2) R2D2C Technical Approach: The R2D2C approach involves a number of phases, which are reflected in the system architecture described in Figure 4. The following describes each of these phases.

D1 Scenarios Capture: Engineers, end users, and others write scenarios describing intended system operation. The input scenarios may be represented in a constrained natural language using a syntax-directed editor, or may be represented in other textual or graphical forms.

D2 Traces Generation: Traces and sequences of atomic events are derived from the scenarios defined in phase D1.

D3 Model Inference: A formal model, or formal specification, expressed in CSP is inferred by an automatic theorem prover — in this case, ACL2 [30] — using the traces derived in phase D2. A deep[1] embedding of the laws of concurrency [21] in the theorem prover gives it sufficient knowledge of concurrency and of CSP to perform the inference. The embedding will be the topic of a future paper.

D4 Analysis: Based on the formal model, various analyses can be performed, using currently available commercial or public domain tools, and specialized tools that are planned for development. Because of the nature of CSP, the model may be analyzed at different levels of abstraction using a variety of possible implementation environments. This will be the subject of a future paper.

D5 Code Generation: The techniques of automatic code generation from a suitable model are reasonably well understood. The present modeling approach is suitable for the application of existing code generation techniques, whether using a tool specifically developed for the purpose, or existing tools such as FDR [12], or converting to other notations suitable for code generation (e.g., converting CSP to B [9]) and then using the code generating capabilities of the B Toolkit.

[1]"Deep" in the sense that the embedding is semantic rather than merely syntactic.

3) Advantages of the R2D2C Approach: We have not yet had an opportunity to apply R2D2C to ANTS, although that is certainly our plan.

In addition to applying it to the HRSM procedures [39], we have applied R2D2C to LOGOS, a NASA prototype Lights-Out Ground Operating System, that exhibits both autonomous and autonomic properties [48], [49]. We illustrate the use of a prototype tool to apply R2D2C to LOGOS in [40], and describe our success with the approach.

Here, we summarize some benefits of using R2D2C, and hence of using Formal Requirements-Based Programming in system development. It is our contention that R2D2C, and other approaches that similarly provide mathematical soundness throughout the development lifecycle, will:

- Dramatically increase assurance of system success by ensuring
  - completeness and consistency of requirements
  - that implementations are true to the requirements
  - that automatically coded systems are bug-free; and that
  - that implementation behavior is as expected
- Decrease costs and schedule impacts of ultra-high dependability systems through automated development
- Decrease re-engineering costs and delays

E. Tool Support

John Rushby [42] argues that tools are not the *most* important thing about formal methods, they are the *only* important thing about formal methods. Although we can sympathize, we do not support such an extreme viewpoint. Formal methods would not be practical without suitable representation notations, proof systems (whether automated and supported by tools, or not), a user community, and evidence of successful application.

We do agree, however, that tool support is vital, and not just for formal methods. Structured design methods *took off* when they were *standardized*, in the guise of UML. But

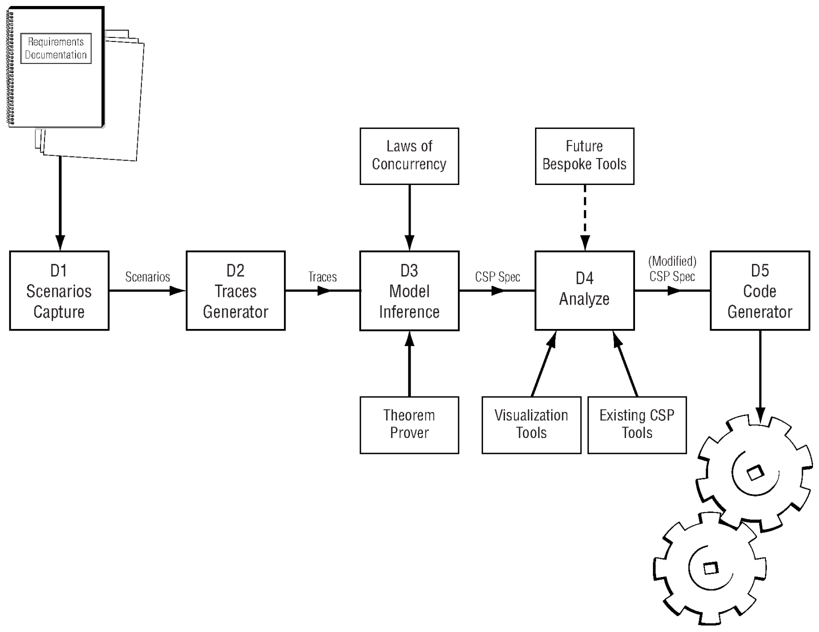

Fig. 4. The entire process with D1 thru D5 illustrating the development approach.

it is only with the advent of tool support for UML that they became popular. The situation is analogous to high-level programming languages: while the community was well convinced of their benefits, it was only with the availability of commercial compilers that they became widely used.

Tools are emerging for the development of complex agent-based systems such as Java-based Aglets and tools for autonomic systems. For automatic code generation and Formal Requirements-Based Programming to be practical, the development community will need commercial-quality tools.

V. Conclusion

The computing industry thrives on the assumption in the marketplace that software is reliable and correct, but many examples from experience over the decades cast doubt on the validity of this assumption. There is no automated, general purpose method for building correct systems that fully meet all customer requirements. This represents a major gap that has yet to be fully addressed by the software engineering community. Requirements-based programming has been described along with new automated techniques recently devised at NASA for ensuring correctness of the system model with respect to the requirements, as a possible way to close this gap.

In future mission concepts that involve advanced architectures and capabilities — such as swarm missions whose individual elements not only can learn from experience but also must pursue science goals cooperatively — NASA faces system development challenges that cannot be met with techniques currently available in the computing industry. The challenges boil down to building reliability and correctness into mission systems, where complexity, autonomous operation, machine adaptation, dangerous environments, and remoteness combine to push such missions far into uncharted territory in systems engineering. With approaches such as autonomic computing and automated requirements-based programming, NASA will have greater possibilities for achieving success with these advanced mission concepts.

References

[1] I. Bakam, F. Kordon, C. L. Page, and F. Bousquet. Formalization of a spatialized multiagent model using Coloured Petri Nets for the study of an hunting management system. In *Proc. First International Workshop on Formal Approaches to Agent-Based Systems (FAABS I)*, number 1871 in LNAI, Greenbelt, Maryland, April 2000. Springer.

[2] F. L. Bauer. A trend for the next ten years of software engineering. In H. Freeman and P. M. Lewis, editors, *Software Engineering*, pages 1–23. Academic Press, 1980.

[3] G. Beni and J. Want. Swarm intelligence. In *Proc. Seventh Annual Meeting of the Robotics Society of Japan*, pages 425–428, Tokyo, Japan, 1989. RSJ Press.

[4] E. Bonabeau, G. Théraulaz, J.-L. Deneubourg, S. Aron, and S. Camazine. Self-organization in social insects. *Trends in Ecology and Evolution*, 12:188–193, 1997.

[5] L. Bonnet, G. Florin, L. Duchien, and L. Seinturier. A method for specifying and proving distributed cooperative algorithms. In *Proc. DIMAS-95*, November 1995.

[6] J. P. Bowen and M. G. Hinchey. *High-Integrity System Specification and Design*. FACIT Series. Springer-Verlag, London, UK, 1999.

[7] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[8] R. Büssow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with Statecharts and Z: A case study. In Astesiano, editor, *Proc. International Conference on Fundamental Approaches to Software Engineering*, number 1382 in LNCS, pages 71–87, Berlin, 1998. Springer-Verlag.

[9] M. J. Butler. *csp2B : A Practical Approach To Combining CSP and B*. Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, February 1999.

[10] C. Fellenstein. *On Demand Computing*. IBM Press Series on Information Management. Prentice-Hall, Upper Saddle River, New Jersey, USA, 2005.

[11] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Universität Oldenburg, Germany, 2000.

[12] Formal Systems (Europe), Ltd. *Failures-Divergences Refinement: User Manual and Tutorial*, 1999.

[13] A. K. Gala and A. D. Baker. Multi-agent communication in JAFMAS. In *Proc. Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents ('99)*, Seattle, Washington, 1999.

[14] A. J. Galloway and W. J. Stoddart. An operational semantics for ZCCS. In M. Hinchey and S. Liu, editors, *Proc. IEEE International Conference on Formal Engineering Methods (ICFEM-97)*, pages 272–282, Hiroshima, Japan, November 1997. IEEE Computer Society Press, Los Alamitos, Calif.

[15] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[16] J. N. Gray. What next? A few remaining problems in information technology. Turing Award Lecture (ACM FCRC), May 1999.

[17] J. N. Gray. Dependability in the Internet era. In *Proc. High Dependability Computing Consortium Workshop*, Santa Cruz, California, 7 May 2001.

[18] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[19] D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1):8–20, January 1992.

[20] D. Harel. Comments made during presentation at "Formal Approaches to Complex Software Systems" panel session. *ISoLA-04 First International Conference on Leveraging Applications of Formal Methods*, Paphos, Cyprus. 31 October 2004.

[21] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill International, London, UK, 1995.

[22] M. G. Hinchey, J. L. Rash, and C. A. Rouff. 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.

[23] M. G. Hinchey, J. L. Rash, and C. A. Rouff. Towards an automated development methodology for dependable systems with application to sensor networks. In *Proc. IEEE Workshop on Information Assurance in Wireless Sensor Networks (WSNIA 2005), Proc. International Performance Computing and Communications Conference (IPCCC-05) (Reprinted in Proc. Real Time in Sweden 2005 (RTiS2005), the 8th Biennial SNART Conference on Real-time Systems, 2005)*, Phoenix, Arizona, 7–9 April 2005. IEEE Computer Society Press, Los Alamitos, Calif.

[24] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, 1985.

[26] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Presented at agenda 2001, scotsdale, arizona, 2001, IBM T. J. Watson Laboratory, October 15, 2001.

[27] P. M. Horn. Meeting the needs, realizing the opportunities. In C. W. Wessner, editor, *Capitalizing on New Needs and New Opportunities: Government - Industry Partnerships in Biotechnology and Information Technologies (2001) Board on Science, Technology, and Economic Policy (STEP)*, pages 149–152. The National Academies Press, 2001.

[28] IFAD. The VDM++ toolbox user manual. Technical report, IFAD, 2000.

[29] JPL Special Review Board. Report on the Loss of the Mars Polar Lander and Deep Space 2 missions. Pasadena, California, USA, March 2000.

[30] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods Series. Kluwer Academic Publishers, Boston, 2000.

[31] K. Lano and H. Haughton. *Specification in B: an Introduction Using the B-Toolkit*. Imperial College Press, London, UK, 1996.

[32] H. W. Lawson. Rebirth of the computer industry. *Communications of the ACM*, 45(6):25–29, 2002.

[33] N. G. Leveson. Medical devices: The Therac-25 story. In *Safeware: System Safety and Computers*, pages 515–553. Addison Wesley Publishing Company Inc., 1995.

[34] J. L. Lyons. ARIANE 5: Flight 501 failure, report by the inquiry board, 19 July 1996.

[35] The MathWorks, Inc., Natick, Massachusettes. *Getting Started with MATLAB*, 2000.

[36] D. L. Parnas. Software aspects for strategic defense systems. *American Scientist*, November 1985.

[37] D. L. Parnas. Using mathematical models in the inspection of critical software. In *Applications of Formal Methods*, International Series in Computer Science, pages 17–31. Prentice Hall, Englewood Cliffs, NJ, 1995.

[38] D. Patterson and A. Brown. Recovery-Oriented Computing (Keynote talk). In *Proc. High Performance Transaction Systems Workshop (HPTS)*, October 2001.

[39] J. L. Rash, M. G. Hinchey, C. A. Rouff, and D. Gračanin. Formal requirements-based programming for complex systems. In *Proc. International Conference on Engineering of Complex Computer Systems*, Shanghai, China, 16–20 June 2005. IEEE Computer Society Press, Los Alamitos, Calif.

[40] J. L. Rash, M. G. Hinchey, C. A. Rouff, D. Gračanin, and J. D. Erickson. A tool for requirements-based programming. In *Proc. International Conference on Integrated Design and Process Technology (IDPT 2005)*, Beijing, China, 13–17 June 2005. The Society for Design and Process Science.

[41] C. A. Rouff, W. F. Truszkowski, J. L. Rash, and M. G. Hinchey. A survey of formal methods for intelligent swarms. Technical Report TM-2005-212779, NASA Goddard Space Flight Center, Greenbelt, Maryland, 2005.

[42] J. Rushby. Remarks, panel session on The Future of Formal Methods in Industry. In J. P. Bowen and M. G. Hinchey, editors, *Proc. 9th International Conference of Z Users, LNCS 967*, pages 239–241, Limerick, Ireland, September 1995. Springer-Verlag.

[43] R. Sterritt. Towards autonomic computing: Effective event management. In *Proc. 27th Annual IEEE/NASA Software Engineering Workshop (SEW)*, pages 40–47, Greenbelt, Maryland, USA, 3–5 December 2002. IEEE Computer Society Press, Los Alamitos, Calif.

[44] R. Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering: a NASA Journal*, 1(1), April 2005.

[45] R. Sterritt and D. W. Bustard. Autonomic computing—a means of achieving dependability? In *Proc. IEEE International Conference on the Engineering of Computer Based Systems (ECBS-03)*, pages 247–251, Huntsville, Alabama, USA, April 2003. IEEE Computer Society Press, Los Alamitos, Calif.

[46] R. Sterritt and M. G. Hinchey. Why computer based systems *Should* be autonomic. In *Proc. 12th IEEE International Conference on Engineering of Computer Based Systems (ECBS 2005)*, pages 406–414, Greenbelt, MD, April 2005.

[47] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. NASA's swarm missions: The challenge of building autonomous software. *IEEE IT Professional*, 6(5):47–52, September/October 2004.

[48] W. F. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff. Autonomous and autonomic systems: A paradigm for future space exploration missions. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 36(3):279–291, May 2006.

[49] W. F. Truszkowski, J. L. Rash, C. A. Rouff, and M. G. Hinchey. Some autonomic properties of two legacy multi-agent systems — LOGOS and ACT. In *Proc. 11th IEEE International Conference on Engineering Computer-Based Systems (ECBS), Workshop on Engineering Autonomic Systems (EASe)*, pages 490–498, Brno, Czech Republic, May 2004. IEEE Computer Society Press, Los Alamitos, Calif.