Experiences with a Requirements-Based Programming Approach to the Development of a NASA Autonomous Ground Control System

James L. Rash, Michael G. Hinchey
NASA Goddard Space Flight Center
Information Systems Division
Greenbelt, MD, USA
{james.l.rash, michael.g.hinchey}@nasa.gov

Christopher A. Rouff
SAIC
Advanced Concepts Business Unit
McLean, VA 22102
rouffc@saic.com

Denis Gračanin
Department of Computer Science
Virginia Tech
Blacksburg, VA
gracanin@vt.edu

John Erickson
Department of Computer Sciences
University of Texas at Austin
Austin, TX
jderick@cs.utexas.edu

Abstract

Requirements-to-Design-to-Code (R2D2C) is an approach to the engineering of computer-based systems that embodies the idea of requirements-based programming in system development. It goes further, however, in that the approach offers not only an underlying formalism, but full formal development from requirements capture through to the automatic generation of provably-correct code. As such, the approach has direct application to the development of systems requiring autonomic properties. We describe a prototype tool to support the method, and illustrate its applicability to the development of LOGOS, a NASA autonomous ground control system, which exhibits autonomic behavior. Finally, we briefly discuss other areas where the approach and prototype tool are being considered for application.

1. Introduction

We have advocated that computer-based systems should be autonomic [20], and, more specifically, that autonomous systems are necessarily autonomic [21]. Clearly, too, autonomic systems are inherently autonomous, as they are required to necessarily adapt and evolve to meet their goals of being self healing, self configuring, self optimizing, and self protecting.

Such systems can prove to be exceedingly complex, and consequently their development extremely difficult. Often, the complete behavior of the system cannot be foreseen at the outset, partly because of the evolving nature of the system, and partly because it is difficult to capture all of the necessary domain knowledge before development begins. Because of this and the system's emergent behavior (that is, behavior that is exhibited by a system as it evolves, but which was not anticipated) the system cannot be fully tested using traditional methods [18].

Clearly formal methods can go a long way towards solving these problems, and can reduce reliance on testing. However, they are still perceived to be difficult to use [4], and their uptake in industry has not been as commonplace as one would have expected.

2. Requirements-Based Programming

2.1. Background

Requirements-Based Programming (RBP) has been advocated [7, 8] as a viable means of developing complex, evolving systems. It embodies the idea that requirements can be systematically and mechanically transformed into executable code.

This may seem to be an obvious goal in the engineering of computer-based 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, called a design (or, more correctly, a design specification), from which an implementation will be derived. This model must itself be derived from the system requirements, but there is a large "gap" in going from requirements to design [11]. RBP seeks to eliminate this "gap" by ensuring

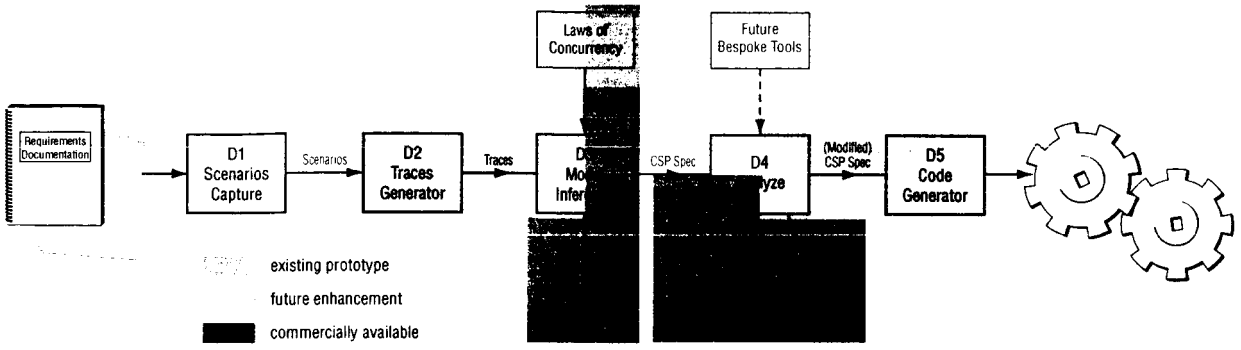

Figure 1. The R2D2C approach and current status of the prototype.

that the ultimate implementation can be fully traced back to the actual requirements (although, as proposed by its advocates, it does not necessarily entail full mathematical provability of the equivalence of a set of requirements and its implementation).

2.2. R2D2C

R2D2C (Requirements-to-Design-to-Code) is a NASA patent-pending 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, described in greater detail in [11], embodies the main idea of requirements-based programming. It goes further, however, in that the approach 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; these include, for example, instructions for controlling physical devices, and rules embodying the knowledge contained in expert systems. In these contexts, NASA is currently applying the approach to the verification of the instructions and procedures to be generated by the Hubble Space Telescope Robotic Servicing Missions (HRSM) and in the validation of the rule base used in the ground control of the ACE spacecraft.

In the remainder of this paper we describe a prototype tool to support the R2D2C method and report on our experiences in applying it to validate the prototype Lights-Out Ground Operations System (LOGOS), an autonomous system exhibiting autonomic properties [21, 22].

3. Requirements to Design to Code

R2D2C takes, as input, system requirements written by engineers (and others) as scenarios in natural language, or UML use cases, or some other appropriate graphical or textual representation. From the scenarios, an automated theorem prover in which the laws of concurrency [9] have been embedded infers a corresponding process-based specification expressed in an appropriate formal language (currently we are using CSP, Hoare's language of Communicating Sequential Processes [13, 14]).

A process-based specification is far more amenable to analysis, and also forms a more appropriate basis for code generation. As much as possible, R2D2C makes use of widely-available tools and notations that are well-trusted and that have been demonstrated to be useful in the development of high-quality systems.

A "short-cut" approach to R2D2C [10, 11] avoids the use of an automated theorem prover, which is computationally expensive. This alternative approach involves the inference of a corresponding process-based specification (in a language we have named EzyCSP) without a theorem prover, but requires a (one time) proof of the translation in order to preserve the mathematical underpinnings of the R2D2C approach. Figure 1 illustrates those parts of the approach for which we have built a prototype tool (described in the remainder of this paper), and shows where commercially-available and public domain tools may be used to support the approach.

3.1. Prototype Tool

The CSP formal model is the central part of the proposed approach, which conforms to a Model Driven Architecture (MDA) [15]. The prototype tool automatically generates the code from the CSP model (or design) (Figure 2) into which the tool has already transformed the requirements.

In order to develop a tool based on CSP, two major issues must be addressed — how to translate the CSP model

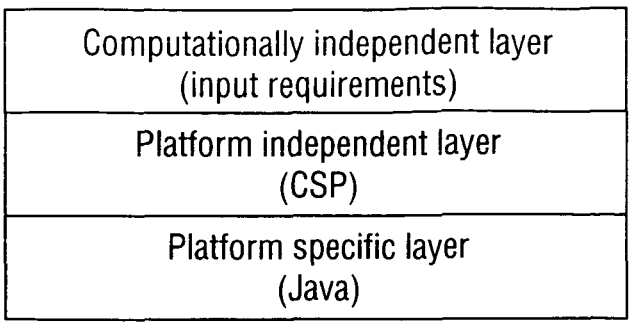

Figure 2. MDA approach

into code and how to translate the requirements into the CSP model. The tool transforms the derived design (CSP model) into an equivalent software representation (code) using Java as the target programming language. There were several reasons for selecting the Java programming language [6] both for tool implementation and for the target platform:

- Java is a general-purpose, concurrent, class-based, object-oriented programming language, with very few implementation and hardware dependencies.
- An off-the-shelf implementation (library) of CSP for Java [2] is available. While JCSP does not provide direct CSP-to-Java mapping, it conforms to the CSP model of communicating systems for Java multi-threaded applications [16]. There is also support for distributed JCSP components using JCSP.net [24].
- Java Swing [23], in combination with some available Java IDEs, greatly simplifies user interface development.
- Many Java-based translator development tools are available.

The prototype tool implementation in Java uses off-the-shelf components. A Swing-based user interface provides a transparent layer for entering the requirements and viewing the resulting model. Figure 3 shows the high-level program flow.

The translators are implemented using the ANTLR [1] tool, which provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. A discussion of ANTLR and some related tools can be found in [19]. An English-like input language, specified as an ANTLR grammar, is used to specify user requirements (Figure 4). ANTLR uses the grammar to automatically generate the translator. The translator is then used to generate the CSP model that corresponds to the user requirements (Figure 5). Figure 6 shows the graph-based representation of the system (under development).

4. Experiences in Applying the Prototype Tool

4.1. LOGOS

The Lights-Out Ground Operations System (LOGOS) is a proof-of-concept NASA system for automatic control of ground stations when satellites pass overhead and under their control. LOGOS is a community of autonomous software agents, exhibiting autonomic behavior and cooperating to perform the functions that in the past have been performed by human operators using traditional software tools such as orbit generators and command sequence planners. It is designed to operate in "lights out" mode (i.e., without human intervention except in situations where problems and anomalies can no longer be dealt with by the system itself). See [18] and [21] for more detailed discussion of LOGOS and its autonomic properties.

4.2. LOGOS in R2D2C

We will not consider the entire LOGOS system here. Although a relatively small system, it is too extensive to illustrate in its entirety in this paper. Instead, we will take a couple of example agents from the system, and illustrate their mapping from natural language descriptions through to simple Java implementations.

Let us first illustrate, via a trivial example, how scenarios map to CSP. Suppose we have the following as part of one of the scenarios for the system:

> if the Spacecraft Monitoring Agent receives a "fault" advisory from the spacecraft the agent sends the fault to the Fault Resolution Agent
> OR
> if the Spacecraft Monitoring Agent receives engineering data from the spacecraft the agent sends the data to the Trending Agent

That part of the scenario could be mapped to structured text as:

> inSMA?fault from Spacecraft
> then outSMA!fault to FIRE
> else
> inengSMA?data from Spacecraft
> then outengSMA!data to TREND

The laws of concurrency would allow us to derive the traces as:

$$tSMA \supseteq \{\langle\rangle, \langle inSMA, fault\rangle, \langle inSMA, fault, outSMA, fault\rangle\} \bigcup \{\langle\rangle, \langle inengSMA, data\rangle, \langle inengSMA, data, outSMA, data\rangle\}$$

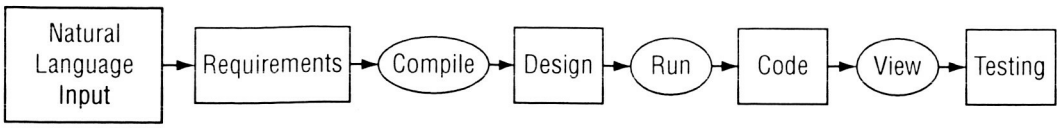

Figure 3. High-level program flow

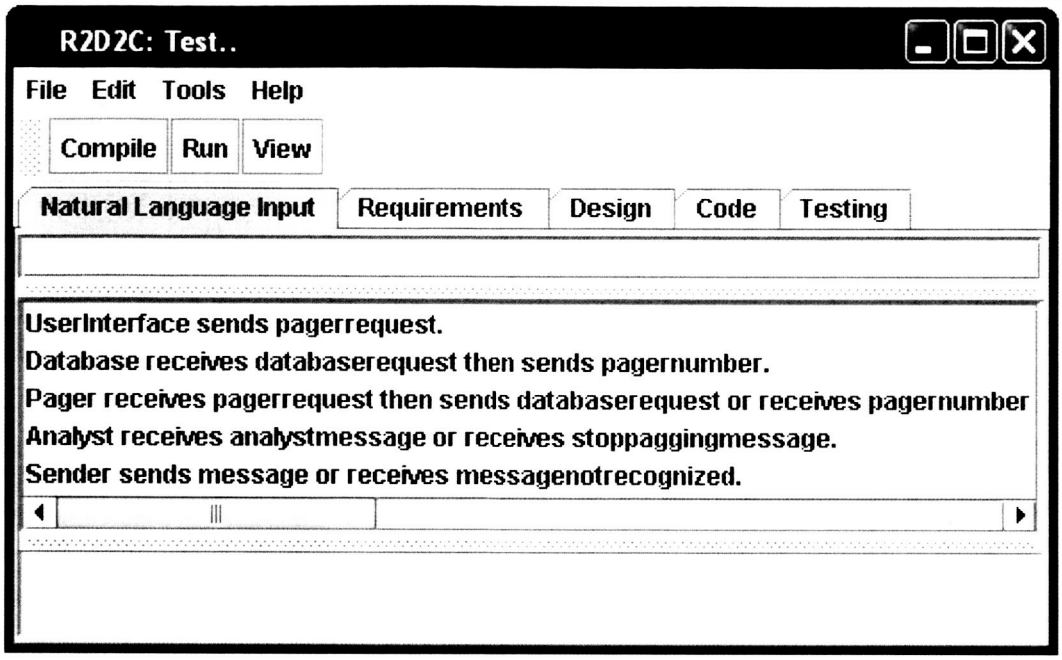

Figure 4. Input requirements

From the traces, we can infer an equivalent CSP process specification as:

$$SMA = inSMA?fault \rightarrow (outSMA!fault \rightarrow SKIP) \mid (inengSMA?data \rightarrow outengSMA!data \rightarrow SKIP)$$

Let us now consider a slightly larger example, the LOGOS Pager Agent, and illustrate its implementation in Java. The pager agent sends pages to engineers and controllers when there is a spacecraft anomaly and there is no analyst logged on to the system. The pager agent receives requests from the user interface agent that no analyst is logged on, gets paging information from the database agent (which keeps relevant information about each user of the system — in this case the analyst's pager number), and, when instructed by the user interface agent that the analyst has logged on, stops paging. These scenarios can be restated in more structured natural language as follows:

if the Pager agent receives a request from the User Interface agent, the Pager agent sends a request to the database agent for an analyst's pager information and puts the message in a list of requests to the database agent

OR

if the Pager agent receives a pager number from the database agent, then the pager agent removes the message from the paging queue and sends a message to the analyst's pager and adds the analyst to the list of paged people

OR

if the Pager agent receives a message from the user interface agent to stop paging a particular analyst, the pager sends a stop-paging command to the analyst's pager and removes the analyst from the paged list

OR

if the Pager agent receives another kind of message, reply to the sender that the message was not recognized

The above scenarios would then be translated into CSP.

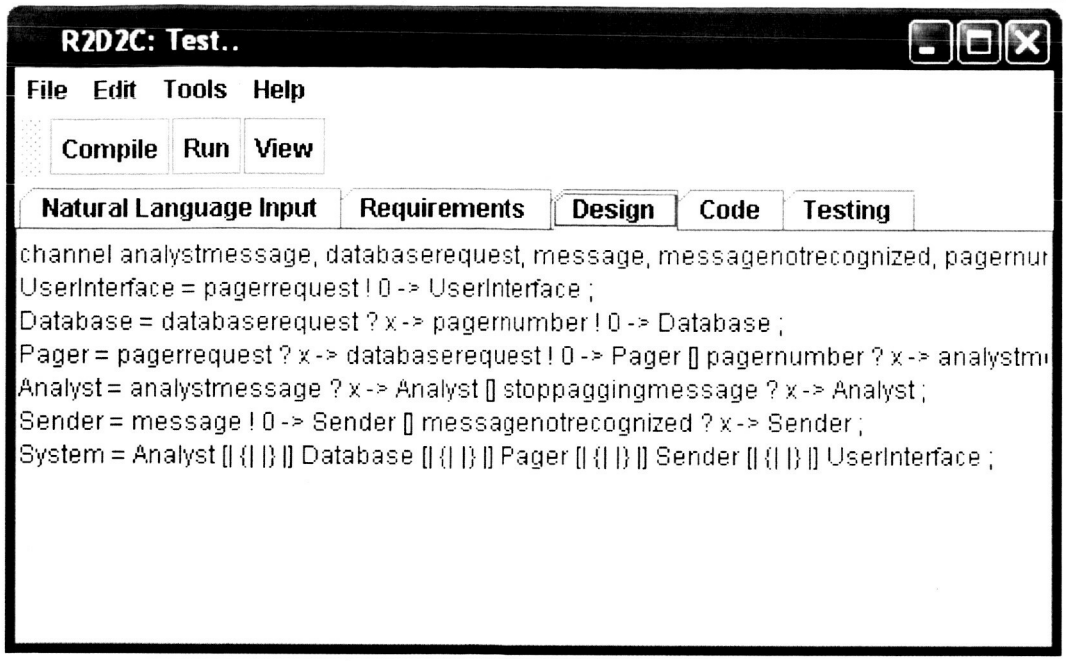

Figure 5. CSP model

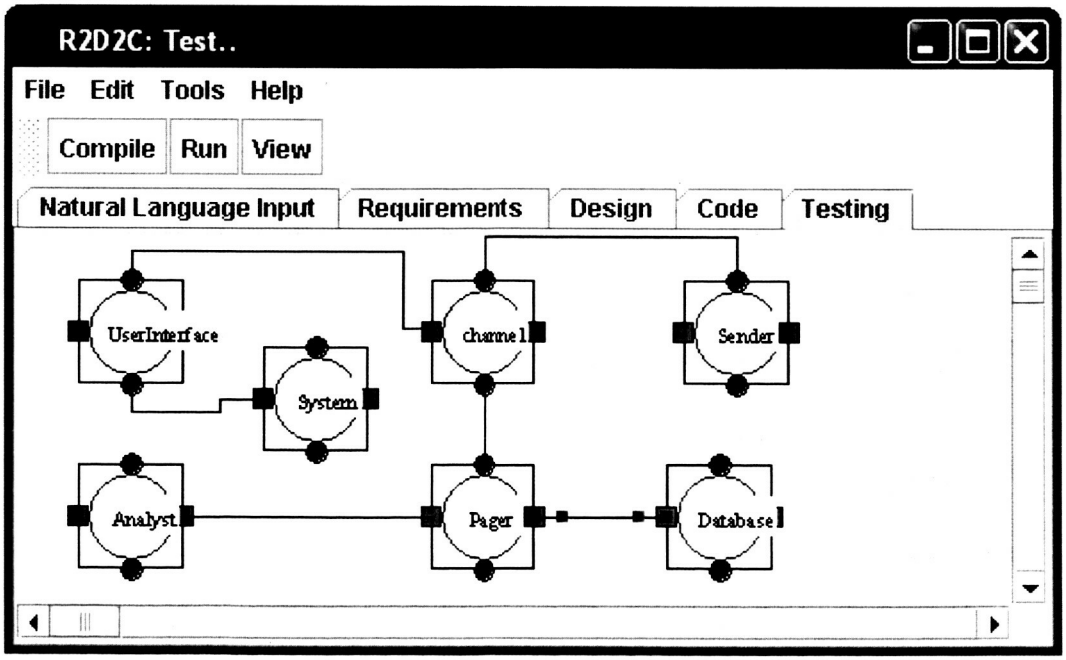

Figure 6. Graphical representation of a system

$$PAGER\_BUS_{db\_waiting,paged} = pager.Iin?msg \rightarrow$$

$$case$$

$$GET\_USER\_INFO_{db\_waiting,paged,pagee,text} \quad if msg = (START\_PAGING, specialist, text)$$

$$BEGIN\_PAGING_{db\_waiting,paged,in\_reply\_to\_id(msg),pager\_num} \quad if\ msg = (RETURN\_DATA.pager\_num)$$

$$STOP\_CONTACT_{db\_waiting,paged,pagee} \quad if\ msg = (STOP\_PAGING, pagee)$$

$$pager.Iout!(head(msg), UNRECOGNIZED) \rightarrow PAGER\_BUS_{db\_waiting,paged} \quad otherwise$$

Figure 7. Partial CSP description of the pager agent.

Figure 7 shows a partial CSP description of the pager agent. This specification states that the process PAGER_BUS receives a message on its "*Iin*" channel and stores it in a variable called "*msg*". Depending on the contents of the message, one of four different processes is executed. If the message has a START_PAGING performative, then the GET_USER_INFO process is called with parameters of the type of specialist to page (*pagee*) and the text to send the pagee. If the message has a RETURN_DATA performative with a pagee's pager number, then the database has returned a pager number and the BEGIN_PAGING process is executed with a parameter containing the original message id (used as a key to the *db_waiting* set) and the passed pager number. The third type of message that the pager agent might receive is one with a STOP_PAGING performative. This message contains a request to stop paging a particular specialist (stored in the *pagee* parameter). When this message is received, the STOP_PAGING process is executed with the parameter of the specialist type. If the pager agent receives any other message than the above three messages, an error message is returned to the sender of the message (which is the first item of the list) stating that the message is "UNRECOGNIZED". After this, the PAGER_BUS process is again executed.

The R2D2C prototype tool will produce Java code from the CSP model as follows:

```
class Pager extends Thread {
  Transaction analystmessage;
  Transaction databaserequest;
  Transaction messagenotrecognized;
  Transaction pagernumber;
  Transaction pagerrequest;
  Transaction stoppaggingmessage;
  boolean running;

  public Pager(Transaction analystmessage,
    Transaction databaserequest,
    Transaction messagenotrecognized,
    Transaction pagernumber,
    Transaction pagerrequest,
    Transaction stoppaggingmessage) {

    this.analystmessage = analystmessage;
    this.databaserequest = databaserequest;
    this.messagenotrecognized =
      messagenotrecognized;
    this.pagernumber = pagernumber;
    this.pagerrequest = pagerrequest;
    this.stoppaggingmessage =
      stoppaggingmessage;}

  public void run() {
    int index = 0;
    running = true;

    while (running) {
      switch (index) {
      case 0:
        while (pagerrequest.committed() ==
          false);
        Test.out.println("pagerrequest");
        Test.out.flush();
        while (databaserequest.committed() ==
          false);
        Test.out.println("databaserequest");
        Test.out.flush();
        break;
      case 1:
        while (pagernumber.committed() ==
          false);
        Test.out.println("pagernumber");
        Test.out.flush();
        while (analystmessage.committed() ==
          false);
        Test.out.println("analystmessage");
        Test.out.flush();
        break;
      case 2:
        while (stoppaggingmessage.committed() ==
          false);
        Test.out.println("stoppaggingmessage");
        Test.out.flush();
        break;
      case 3:
        while (messagenotrecognized.committed() ==
          false);
        Test.out.println("messagenotrecognized");
        Test.out.flush();
        break;}
      index++;
      index}}}
```

4.3. Results

A formal specification of LOGOS in CSP had previously been undertaken by hand [17]. This was most insightful, highlighting over 80 errors and anomalies in the require-

ments of a relatively small system (LOGOS is based, essentially, on ten interacting agents). While many of these were minor oversights that would have caused inconveniences, others were more significant.

A great advantage of using an example for which we already have a formal specification is that we can compare the system derived by our prototype tool with the manually derived formal specification.

Our prototype tool was able to uncover all of the errors and anomalies we found with our manual specification. We were surprised when we first ran it to find that it halted within seconds, having found yet another error that had been introduced into the requirements (due to a typographical error) when changes were made following the original manual formal specification. The prototype tool can cope with the LOGOS requirements, generating a design and a Java implementation in a matter of minutes, whereas manual specification had taken several days and code generation by hand took several weeks.

5. Future Applications

The prototype tool described in this paper is designed to support a NASA patent-pending method for Requirements-Based Programming (RBP). The uniqueness of the method is not in supporting RBP, but in supporting it with a development process that is mathematically tractable over the entire development process. This fully formal development offers levels of assurance and confidence significantly higher than traditionally available.

The method is not limited to producing executable code, however [11]. In addition to applying the approach to agent-based systems (such as LOGOS) as described in this paper, and to Wireless Sensor Networks (WSNs) [12], we are currently examining applications of the approach to the verification of expert systems and robotic applications.

5.1. Expert Systems

A suitable translator from the C Language Integrated Production System (CLIPS) [5], rather than natural language, enables us to use this technology to examine expert system rule bases for consistency, etc., with potential application to existing automated ground control center operations (e.g., for the POLAR and ACE missions). More significantly, we can generate CLIPS rules from CSP just as we would generate code in Java. It is expected that this will be a major asset in capturing domain knowledge for expert systems, and maintaining correctness throughout the entire process of expert system development.

In addition, it will allow us to reverse engineer, validate, and ultimately re-engineer existing rule bases.

While an expert system might not immediately strike us as being an autonomic system, complex expert systems do indeed meet all of the necessary criteria. NASA uses expert systems, for example, to perform automated ground control and monitoring for several classes of spacecraft. Rule bases must constantly be updated based on data received from the spacecraft, to ensure that the spacecraft is protected from unsafe situations, and operating at peak performance.

5.2. Robotic Operations

We have begun exploratory work to determine how the approach and tool may be used in validating robotic procedures to be used in the Hubble Robotic Servicing Mission (HRSM).

Robotic devices will be used in the replacement of cameras, etc. on the Hubble Space Telescope (HST). We expect that the tool will prove to be useful in performing *what-if* analysis of ordering of instructions in complex repair processes, in particular where resources (time being one of the most precious resources) are limited.

Additionally, the approach may be useful in actually generating instructions for the robotic arm that will be used to perform much of the maintenance, in much the same way as it currently generates Java code.

6. Conclusions

The difficulty of developing many autonomous and autonomic applications is explained by their inherent complexity. Often, required autonomous behavior results in emergent, unexplained behavior that could not, reasonably, have been foreseen. The need to exhibit autonomic behavior often compounds the situation, giving rise to necessary self-managing behavior that could not reasonably be expected to be the subject of even the most exhaustive testing plans.

Only with fully formal underpinnings for the development process can we be assured of correctness [3]. Formal development processes will become more and more important in future autonomic computing systems, and the continued success of the Autonomic Computing initiative is predicated on the ability to develop complex systems that both exhibit autonomic self-managing behaviors *and* operate correctly (with respect to their requirements).

The experience related in this paper leads us to be confident that such tools will offer greater levels of assurance in other domains, and enhance both the quality and performance of future autonomous and autonomic systems.

Acknowledgements

Part of this work was supported by the NASA Goddard Space Flight Center Technology Transfer Office. Denis

Gračanin was supported by an ASEE/NASA Summer Faculty Fellowship hosted at the NASA Software Engineering Laboratory (Code 581), NASA Goddard Space Flight Center. John Erickson was supported by the NASA Student Internship Program and by the Information Systems Division (Code 580) at NASA Goddard Space Flight Center.

References

[1] ANTLR: ANother Tool for Language Recognition. http://www.antlr.org/.

[2] Communicating sequential processes for Java (JCSP). http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[3] 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.

[4] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[5] J. C. Giarratano. *CLIPS User's Guide*. NASA Johnson Space Center, Houston, Texas, 1992.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification*. Addison Wesley, Boston, second edition, 2000.

[7] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[8] 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.

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

[10] 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, 2004.

[11] M. G. Hinchey, J. L. Rash, and C. A. Rouff. A formal approach to requirements-based programming. In *Proc. 12th IEEE International Conference on Engineering of Computer Based Systems*, Greenbelt, Maryland, 4–7 April 2005.

[12] M. G. Hinchey, J. L. Rash, and C. A. Rouff. Towards an automated development methodology for dependable systems with application to sensor networks. In *Workshop on Information Assurance in Wireless Sensor Networks (WSNIA2005), Proc. 24th IEEE International Performance Computing and Communications Conference (IPCCC 2005)*, Phoenix, AZ, 7–9 April 2005. IEEE Computer Society Press.

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

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, and Hemel Hempstead, UK, 1985.

[15] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[16] D. Lea. *Concurrent Programming in Java^{TM}: Design Principles and Patterns*. The Java^{TM} Series. Addison-Wesley Professional, Reading, Massachusetts, second edition, 2000.

[17] C. A. Rouff, J. L. Rash, and M. G. Hinchey. Experience using formal methods for specifying a multi-agent system. In *Proc. Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000)*, Tokyo, Japan, 2000. IEEE Computer Society Press.

[18] C. A. Rouff, W. F. Truszkowski, M. G. Hinchey, and J. L. Rash. Verification of emergent behaviors in swarm based systems. In *Proc. 11th IEEE International Conference on Engineering Computer-Based Systems (ECBS), Workshop on Engineering Autonomic Systems (EASe)*, pages 443–448, Brno, Czech Republic, May 2004. IEEE Computer Society Press.

[19] Y. Smaragdakis, S. S. Huang, and D. Zook. Program generators and the tools to make them. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 92–100. ACM Press, 2004.

[20] 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)*, Greenbelt, MD, 4–5 April 2005.

[21] 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*, 2006 (to appear).

[22] 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, 24–27 May 2004. IEEE Computer Society Press.

[23] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *JFC Swing Tutorial, The: A Guide to Constructing GUIs*. Addison Wesley, Boston, second edition, 2004.

[24] P. H. Welch, J. R. Aldous, and J. Foster. CSP networking for Java (JCSP.net). In *Proceedings of the Global and Collaborative Computing Workshop (ICCS 2002)*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, 2002.