Integrated Design and Process Technology, **IDPT-2005**
Printed in the United States of America, June, 2005

A Tool for Requirements-Based Programming

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:

In order to demonstrate the correctness of a system, developers today must resort to either exhaustive testing or some combination of testing and formal verification following the use of appropriate methods in the development process. While formal methods have afforded numerous successes, their application today presents serious issues, e.g., costs to gear up to apply them (time, expensive staff), and scalability and reproducibility (as long as standards in the field are not settled). The testing path cannot be walked to the ultimate goal, because exhaustive testing is infeasible for all but trivial systems. So system verification remains problematic, as do, similarly, system and requirements validation. The predominant view today is that provably correct system development depends on having a formal model of the system—leading to a desire for a mathematically sound method to automate the transformation of customer requirements into a formal model. Such a method, an augmentation of requirements-based programming, will be briefly described in this paper, and a prototype tool to support it will be presented. The method and tool enable both requirements validation and system verification for the class of systems whose behavior can be described as scenarios. An application of the tool to a prototype automated ground control system for NASA missions is presented.

I. Introduction

Automating the software development process has been one of the major goals of Software Engineering almost from the outset [10]. Such automation promises to reduce errors, increase productivity, and improve the quality of code.

Approaches such as Model-Based Development (MBD) or Model-Driven Development (MDD) do much to facilitate automatic code generation. The basic tenet of these approaches is that higher quality code can be produced by having developers focus on the generation of an appropriate model, from which code can be generated. This is in contrast with a more *traditional* development approach whereby models (designs) are continually updated in various iterations of portions of the system development lifecycle. Clearly, having an appropriate model *up front* simplifies the task of automating code generation.

Automatic code generation has been successful, and there are a number of commercially-available tools, many of which support particular notations, such as the Unified Modeling Language (UML) or Specification and Design

Language (SDL). A number of tool vendors successfully market tools to support code generation from these notations (e.g., Rational's ROSE and Telelogic's Tau).

II. Limitations to Current Approaches

We will not critique commercially available (nor public domain) tools for automatic code generation here. We are not questioning the quality or integrity of the code produced by available tools.

However, we do observe that many popular tools have been seen to produce large amounts of extraneous code that is unrelated to the model from which code is being generated, and that cannot be justified as merely necessary standard code [9]. In addition, all available tools are constrained by the quality and appropriateness of the model from which the code is to be generated [10].

Developing an appropriate model *ab initio* is no trivial task. As Brooks [3] puts it, specifying and designing the system is the main challenge, not coding it and testing that code: *I believe the hard part of building software to be the specification, design and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.*

The *Specification-Implementation Gap* refers to the significant change that must be made in transforming a specification (expressed at suitable levels of abstraction and in a manner suitable for various types of analysis) into an efficint implementation in a programming language [9], [10].

The greater problem, or *gap*, however, is the *Analysis-Specification Gap*, which refers to the *jump*, and consequent difficulty, of taking (often complex) requirements and specifying them in a manner that is clear, concise, complete and yet amenable to analysis. This *gap* is significantly more difficult to overcome [9], [10].

Requirements-Based Programming (RBP) [5], [6] represents a great step in that direction.

III. Formal Requirements-Based Programming

Requirements-Based Programming (RBP) essentially extends Model-Based Development so that requirements are systematically and mechanically transformed into executable code.

This may seem to be an obvious goal in modern system development, but RBP does in fact go a step further than current development methods. RBP seeks to ensure 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) [16].

Our belief is that Requirements-Based Programming should be formal [17]. That is, that in addition to supporting tractability from requirements through to code, there should be an underlying formalism. Without such formality, proof of correctness is impossible [2].

A. 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 complex NASA mission software, flight software, and ground control systems, amongst others.

The approach, described in greater detail in [10], [17], 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.

B. 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 [8] 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 [11], [12], but other languages may alternately be used).

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 [16].

A "short-cut" approach to R2D2C [9], [10] 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 [16].

IV. Laws of Concurrency

We use the term *laws of concurrency* to refer to a large number of rules, equivalences and interrelationships of CSP operators, as detailed in [8]. These *laws* allow us to demon-

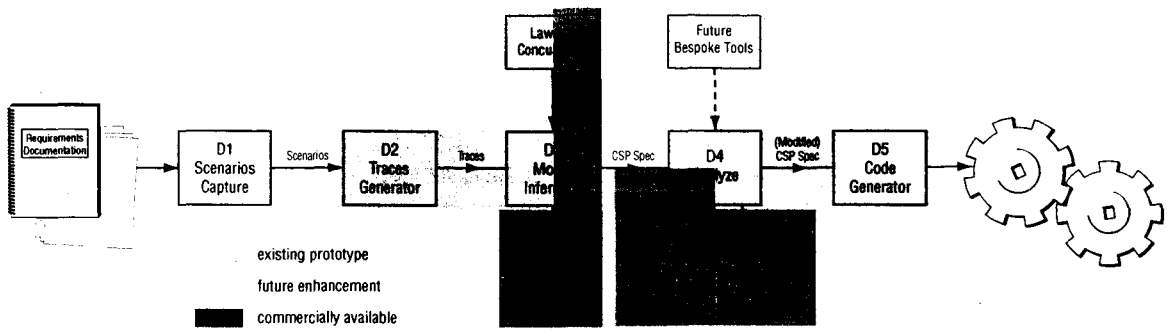

Fig. 1. The R2D2C approach and current status of the prototype.

strate equivalences in CSP models, most of which also hold for concurrent systems in general.

In CSP, two process expressions are deemed to be equivalent if they can be demonstrated to have the same semantic interpretation in the current semantic model of CSP (e.g., traces, failures-divergences) that we are applying. For complex systems in which we wish to consider deadlock and livelock, a more powerful semantic model, such as failures-divergences, may be required. For simple systems, it may simply be the original traces model of CSP [12]. (A trace of a process, in CSP terminology, is the sequence of atomic events that the process has engaged in up to a given point in time.)

For example, in the simple traces model, we can demonstrate that parallel composition is both commutative and associative by proving that

$$\mathbf{t}(P||Q) = \mathbf{t}(Q||P)$$

and

$$\mathbf{t}P||(Q||R) = \mathbf{t}(P||Q)||R.$$

These rules (commutativity and associativity) now enable us to rewrite CSP expressions in various ways. Additionally, *refinement* may be defined appropriately in the semantic model. If (in that semantic model) we can prove that

$$P_1||\ldots||P_n \sqsubseteq P$$

then we may move up and down through various levels of abstraction, and either increase (or decrease) the degree of concurrency in the CSP model by substituting $P_1||\ldots||P_n$ for P (or P for $P_1||\ldots||P_n$).

The R2D2C tool is supported by having a deep embedding of these laws of concurrency in the theorem prover (currently ACL2). Sufficient rules are added to the theorem prover to capture the laws, and their interrelation in various semantic models, to enable proof of a plethora of equivalences and refinements. The relevant semantic model of CSP may also be used to prove properties and invariants. It can be used to investigate whether certain conditions arise or not, whether certain occurrences can be recovered from, to demonstrate (in the more sophisticated models) the existence or absence of deadlock or livelock, etc.

Typically, the laws of concurrency are used to provide an interpretation for process expressions. That is, we prove two process expressions to be equivalent by demonstrating that they have the same interpretation in the given semantic model. For example, in the traces model, the rules might be used to generate the sets of traces of two process expressions, which are then compared to determine whether they are equal or not. In the R2D2C approach, we apply the laws in the opposite direction: a set of traces is examined to *infer* an equivalent process expression.

V. Prototype Tool

Our R2D2C approach may easily be adapted to use other notations. We are currently using CSP. In part, this is because we believe it is highly appropriate for the classes of systems with which we ourselves are concerned. Additionally, it is well known in the profession and widely taught in college courses, and so should be familiar to many. Lastly, several commercial and public domain tools are available to support its use and implementation in Java and other programming languages [7].

The formal model, expressed in CSP in this case, is the central part of the proposed approach, which conforms to a Model Driven Architecture (MDA) [13]. The prototype tool automatically generates the code from the CSP model (or design) (Figure 2) into which the tool has already transformed the requirements [16].

Two major issues must be addressed:

- translating the CSP model into code, and
- translating 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 [16]. There were several reasons for selecting the Java programming language [4] 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.

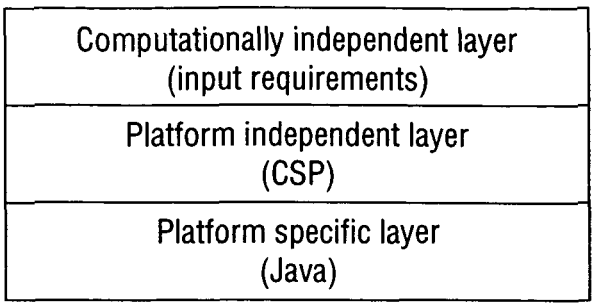

Fig. 2. MDA approach

- An off-the-shelf implementation (library) of CSP for Java [1] is available. While Communicating Sequential Processes for Java (JCSP) does not provide direct CSP-to-Java mapping, it conforms to the CSP model of communicating systems for Java multithreaded applications [14]. There is also support for distributed JCSP components using JCSP.net [23].
- Java Swing [22], 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 [16]. Figure 3 shows the high-level program flow.

The translators are implemented using the ANTLR (ANother Tool for Language Recognition) tool [15], 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 [20]. 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) [16].

VI. A Real Life Example

A. 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). The interested reader is directed to more detailed descriptions of LOGOS, given in [19] and [21].

B. 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 implementation in Java code.

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:

$$\begin{aligned} tSMA \supseteq\ & \{\langle\rangle, \langle inSMA, fault\rangle, \\ & \langle inSMA, fault, outSMA, fault\rangle\} \cup \\ & \{\langle\rangle, \langle inengSMA, data\rangle, \\ & \langle inengSMA, data, outSMA, data\rangle\} \end{aligned}$$

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

$$\begin{aligned} SMA = \ & inSMA?fault \rightarrow (outSMA!fault \rightarrow SKIP) \\ & \mid (inengSMA?data \rightarrow outengSMA!data \rightarrow SKIP) \end{aligned}$$

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

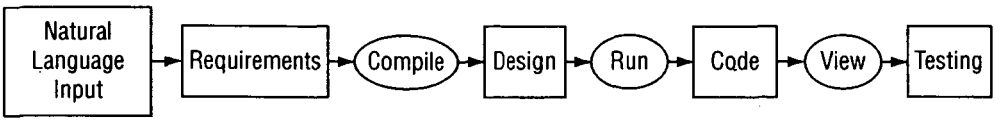

Fig. 3. High-level program flow

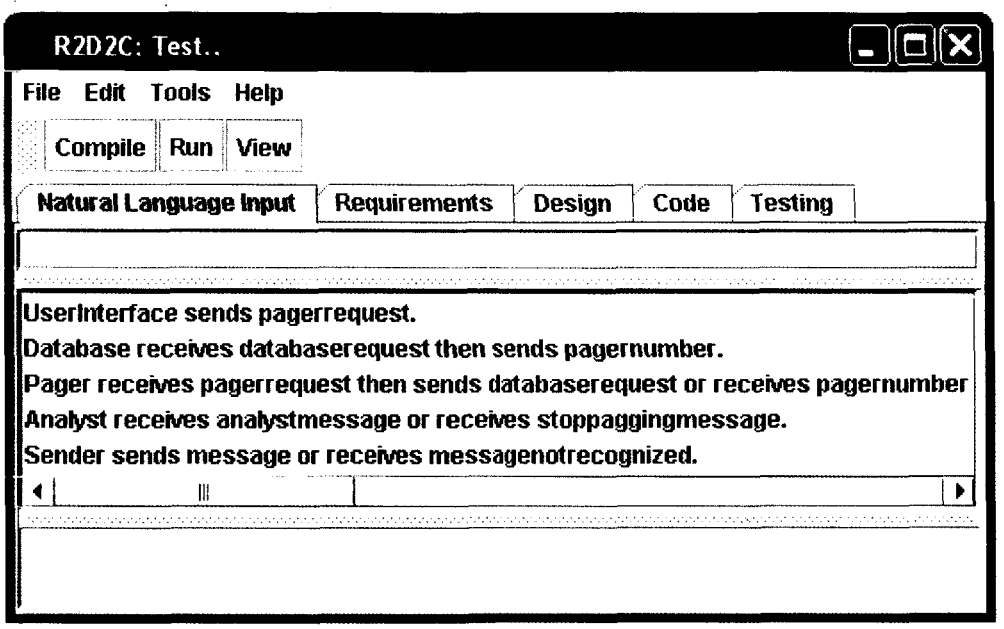

Fig. 4. Input requirements

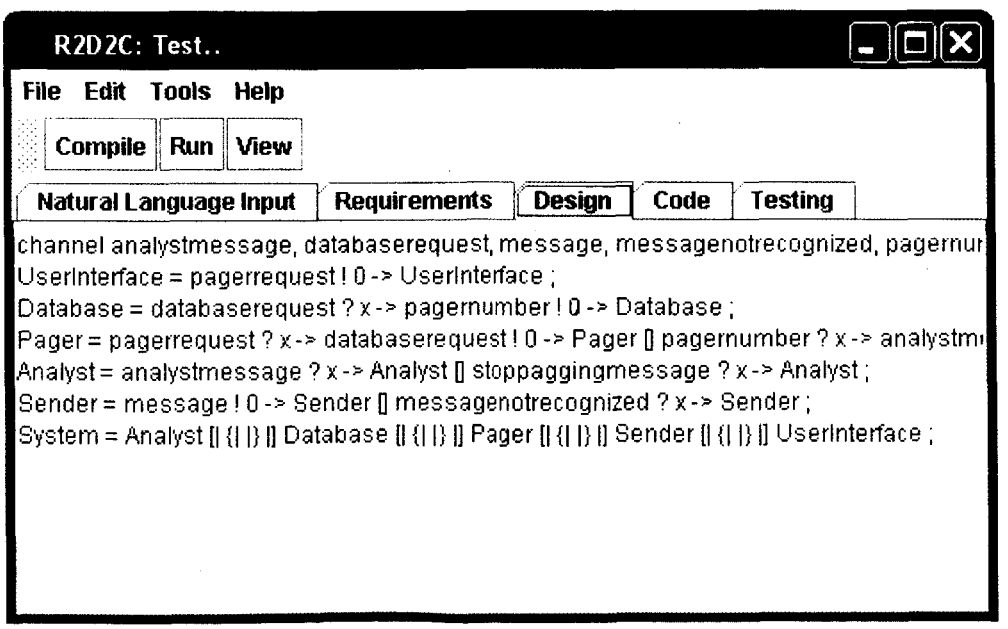

Fig. 5. CSP model

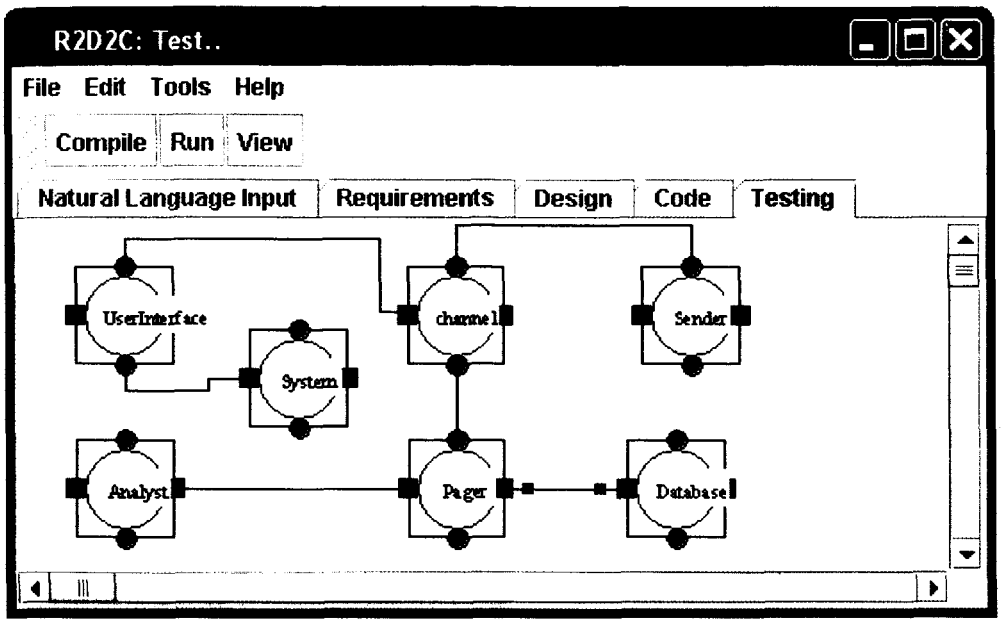

Fig. 6. Graphical representation of a system

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 7 shows a partial CSP description of the pager agent. This specification states that the process PAGER_BUS receives a message on its "*lin*" 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 contai ns 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;
```

$$PAGER\_BUS_{db\_waiting,\ paged} = pager.Iin?msg \rightarrow$$
$$case$$
$$GET\_USER\_INFO_{db\_waiting,\ paged,\ pagee,\ text}$$
$$if\ msg = (START\_PAGING,\ specialist,\ text)$$
$$BEGIN\_PAGING_{db\_waiting,\ paged,\ in\_reply\_to\_id(msg),\ pager\_num}$$
$$if\ msg = (RETURN\_DATA,\ pager\_num)$$
$$STOP\_CONTACT_{db\_waiting,\ paged,\ pagee}$$
$$if\ msg = (STOP\_PAGING,\ pagee)$$
$$pager.Iout!(head(msg),\ UNRECOGNIZED) \rightarrow PAGER\_BUS_{db\_waiting,\ paged}$$
$$otherwise$$

Fig. 7. Partial CSP description of the pager agent.

```
  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}}}
```

C. Results

A formal specification of LOGOS in CSP had previously been undertaken by hand [18]. This was most insightful, highlighting over 80 errors and anomalies in the requirements of a relatively small system (LOGOS is based, essentially, on ten interacting agents). While many of these were minor oversights that only 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.

Several errors of omission were found in LOGOS. A typical sampling of these follows.

- The implementation left undetermined what happens if the pager agent receives no response within a specified amount of time. In that case, should the pager agent automatically resubmit a page, or should this command come from the user interface agent?
- It was left undetermined whether the pager agent should change whom they are paging after some prescribed elapsed period of time with no response to the page, and what that time interval should be. Or, again, should that information come from the UIFA?
- It was left undetermined what happens when the pager agent receives a request to page someone that it has already paged, yet there is not a response to the page and a request to stop paging that person has not been received. In this situation, the pager agent can either re-page the party or ignore the request. In addition, if a party can be paged multiple times, does the software have to keep track of the number of times the party was paged or other relevant information?
- It was left undetermined whether the pager agent should cache specialist pager numbers and information for a specific amount of time, or should always request this information from the database (even if

there is an active, unanswered page for a specialist).
- There is nothing specified as to what should be done if the requested pagee does not exist.

Our prototype tool was able to uncover all of the errors and anomalies we found with our manual inspection. 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.

D. Range of Applicability

The R2D2C method and the associated tool are highly applicable to those classes of system whose requirements may be expressed as scenarios, whether in natural language or some graphical or other textual representation.

It is particularly appropriate for use with systems that involve high degrees of concurrency. This naturally includes most of NASA's missions, and in particular current and forthcoming autonomous systems, where missions will involve greater degrees of self management. This is essentially the class of systems for which we were first motivated to develop the approach.

VII. Further Applications & Future Work

We have described a prototype tool to support formal Requirements-Based Programming. In its current realization, that tool takes requirements expressed in (constrained) natural language, derives a formal model (currently expressed in CSP), and generates a simple Java implementation.

As one would expect, most of these notations may be substituted with alternative (but equivalent) notations. Requirements input may be given as UML use cases, for example, or in a tabular format. Other process algebras may be used for the internal representation of the formal model (design). And, as we pointed out in Section III-A, the *code* produced need not necessarily be code in a conventional programming language.

At the time of writing, we are currently applying the tool to procedures that may be used in the Hubble Robotic Servicing Mission (HRSM). HRSM is an unmanned servicing mission that will use two robot arms, using a combination of telecontrol and robotic automations, to replace cameras and gyroscopes and perform other upgrades on the Hubble Space Telescope (HST). Servicing will take extended periods of time, many lasting several days, under the constraints of limited battery power, and re-planning will be necessary after launch as it is only at this point that the final orbital phasing is known.

Our experiences using R2D2C to generate the ordering of telecontrolled operations, and the instructions for robotic devices, are described in [17].

We are also looking at using this technology for validation and verification of expert systems and for capturing expert system domain knowledge, which we view essentially as requirements. The approach may be applied to expert systems used in automating ground control of various spacecraft, such as Advanced Composition Explorer (ACE).

Future work will include
- the development of several analysis and validation tools (as shown in Figure 1);
- addition of support for *what-if* analysis and alternative implementations;
- support for a variety of graphical, textual, and tabular input notations, simultaneously;
- improving the quality of the inference (by a theorem prover) of process-based specifications from requirements;
- optimizing code generation;
- improving the user interface of the prototype; and
- applying the tool to further complex real-life examples.

VIII. Conclusions

A tool to support the Requirements-to-Design-to-Code (R2D2C) method and a simple example of its application, as presented in this paper, illustrates the potential for augmented requirements-based programming. The NASA prototype Lights Out Ground Operation System (LOGOS) is an example of the class of appropriate applications of the R2D2C method—where system requirements can be stated as scenarios. The behavior of a LOGOS agent can be described in terms of scenarios, which the prototype R2D2C tool can transform into EzyCSP, a subset of the formal specification language Communicating Sequential Processes (CSP). The tool transforms the CSP model into Java code representing the original requirements (scenarios). The transformations are provably equivalent, thus qualifying R2D2C as a mathematically sound method for transforming requirements into an implementation. Since the CSP model can be analyzed mathematically as to satisfaction of any given propositions, R2D2C produces validated requirements, and since the model can be transformed into a provably equivalent implementation, R2D2C produces a system with verifiable correctness.

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 In-

ternship Program and by the Information Systems Division (Code 580) at NASA Goddard Space Flight Center.

The technology reported in this paper is protected in the USA and other countries by patent applications assigned to the United States government.

REFERENCES

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

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

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

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

[6] 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.

[7] M. G. Hinchey. A formal design method for real-time Ada software. In L. Collingbourne, editor, *Ada: Towards Maturity*, Computer and Communications Series 6. IOS Press, Amsterdam, 1993.

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

[9] 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.

[10] M. G. Hinchey, J. L. Rash, and C. A. Rouff. A formal approach to requirements-based programming. In *Proc. IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS-2005*, Greenbelt, Maryland, USA, 4–5 April 2005 (to appear). IEEE Computer Society.

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

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

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

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

[15] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[16] J. L. Rash, M. G. Hinchey, C. A. Rouff, D. Gracanin, and J. D. Erickson. Experiences with a requirements-based programming approach to the development of a NASA autonomous ground control system. In *EASe, 2nd IEEE Workshop on Engineering of Autonomic Systems, Proc. ECBS 2005, 12th IEEE International Conference on Engineering of Computer-Based Systems*, Greenbelt, MD, 4–7 April 2005.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[23] 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.