Abstract

A general-purpose method to mechanically transform system requirements into a provably equivalent model has yet to appear. Such a method represents a necessary step toward high-dependability system engineering for numerous possible application domains, including distributed software systems, sensor networks, robot operation, complex scripts for spacecraft integration and testing, and autonomous systems. Currently available tools and methods that start with a formal model of a system and mechanically produce a provably equivalent implementation are valuable but not sufficient. The "gap" that current tools and methods leave unfilled is that their formal models cannot be proven to be equivalent to the system requirements as originated by the customer. For the classes of systems whose behavior can be described as a finite (but significant) set of scenarios, we offer a method for mechanically transforming requirements (expressed in restricted natural language, or in other appropriate graphical notations) into a provably equivalent formal model that can be used as the basis for code generation and other transformations.

1 Introduction

Complex systems in general cannot attain high dependability without addressing the crucial remaining open issues of software dependability. The need for ultra-high dependability systems increases continually, along with a correspondingly increasing need to ensure correctness in system development. By "correctness," we mean that the implemented system is equivalent to the requirements, and that this equivalence can be proved mathematically.

Development of a system that will have a high level of reliability requires the developer to represent the system as a formal model that can be proven to be correct. Through the use of currently available tools, the model can then be automatically transformed into code with minimal or no human intervention to reduce the chance of inadvertent introduction of errors by developers. Automatically producing the formal model from customer requirements—which, as a general capability, is unavailable today—would further reduce the chance of insertion of errors by developers.

Available system development tools and methods that are based on formal models provide neither automated generation of the models from requirements nor automated proof of correctness of the models. Therefore, today there is no automated, generally applicable means to produce a system—or even a procedure—that is a provably correct implementation of the customer's requirements. Furthermore, requirements engineering as a discipline has yet to produce an automated, mathematics-based process for requirements validation.

2 Problem Statement

Automatic code generation from requirements has been the ultimate objective of software engineering almost since the advent of high-level programming languages, and calls for a "requirements-based programming" capability have become deafening[1]. Several tools and products exist in the marketplace today for automatic code generation from a given model; however, they typically 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 [Mat00], in system development toolsets such as the B-Toolkit [LH96] or the VDM++ toolkit [IFA00], 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, as shown in Figure 1(a). 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 [Par95]. 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. Existing reverse-engineering approaches suffer from a similar plight. In typical approaches, such as the one illustrated in Figure 1(b), a model is extracted from an existing system and is then represented in various ways, for example as a digraph [McL92]. The re-engineering process then involves using the resulting representation as the basis for code generation, as above.

2.1 Specifications, Models, and Designs

The model on which automatic code generation is based is referred to as a design, or more correctly, a design specification. There is typically a mismatch between the design and the implementation (sometimes termed the "specification-implementation gap") in that the process of going from a suitable design to an implementation involves many practical decisions that must be made by the au-

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

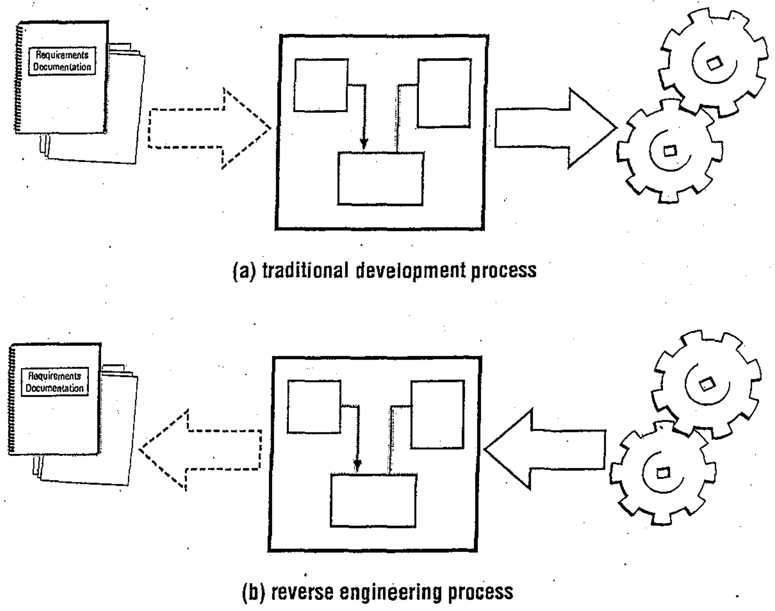
Requirements Documentation
(a) traditional development process
Requirements Documentation
(b) reverse engineering process

Figure 1: (a) The traditional software development process from requirements to code. (b) The reverse engineering process from code to a system description.

tomated tool used for code generation without any clear-cut justifications, other than the predetermined implementation decisions of the tool designers. There is a more problematic "gap," termed the "analysis-specification gap," that emphasizes the problem of capturing requirements and adequately representing them in a specification that is clear, concise, and complete. This specification must be formal, or proof of correctness is impossible [Bau80].

Unfortunately, many are reluctant to embrace formal specification techniques, believing them to be difficult to use and apply [Hal90] [BH95], despite many industrial success stories [HB95] [HB99]. Our experience at NASA Goddard Space Flight Center (GSFC) has been that while engineers are happy to write descriptions as natural language scenarios, or even using semi-formal notations such as UML use cases, they are loath to undertake formal specification. However, absent a formal specification of the system under consideration, there is no possibility of determining any level of confidence in the correctness of an implementation. More importantly, we must ensure that this formal specification fully, completely, and consistently captures the prescribed requirements. We cannot expect requirements to be perfect, complete, and consistent from the outset, which is why it is even more important to have a formal specification, which can highlight errors, omissions, and conflicts. The formal specification must also reflect changes and updates from system maintenance, as well as changes and compromises in requirements, so that it remains an accurate representation of the system throughout the system's lifecycle.

2.2 A Novel Approach

Our approach involves providing a mathematically tractable round-trip engineering approach to system development. The approach described herein is provisionally named R2D2C ("Requirements to Design to Code").

In this approach, 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 2) that is guaranteed to be equivalent to the requirements stated at the outset, and which subsequently will 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 [Hoa78] [Hoa85], 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.

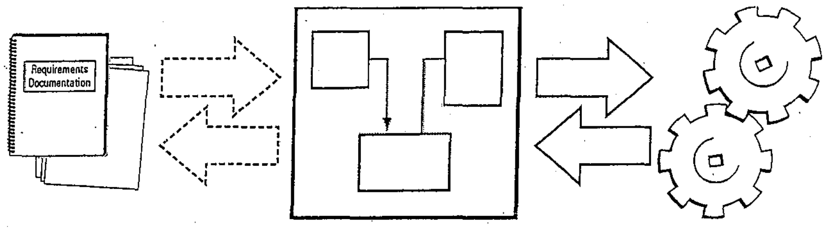

Figure 2: The R2D2C approach, generating a formal model from requirements and producing code from the formal model, with automatic reverse engineering.

3 Technical Approach

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 2. 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 are currently experimenting with using it to generate instructions for robotic devices to be used on the proposed Hubble Robotic Servicing Mission (HRSM). We are also experimenting with using it as a basis for an expert system verification tool, and as a means of capturing expert knowledge for expert systems. (Such potential applications will be described in Section 5.)

Section 3.1 describes the approach at a relatively high level. Section 3.2 describes an intermediate version of the approach, for which we have built a prototype tool, and with which we have successfully undertaken some examples.

3.1 R2D2C

The R2D2C approach involves a number of phases, which are reflected in the system architecture described in Figure 3. 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 D1.

D3 *Model Inference*: A formal model, or formal specification, expressed in CSP, is inferred by an automatic theorem prover—in this case, ACL2 [KP00]—using the traces derived in phase 2. A deep[2] embedding of the laws of concurrency [HJ95] 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 [FSE99], or converting to other notations suitable for code generation (e.g., converting CSP to B [But99]) and then using the code generating capabilities of the B Toolkit.

It should be re-emphasized that the "code" generated may be code in a high-level programming language, low-level instructions for (electro-) mechanical devices, natural-language business procedures and instructions, or the like. As Figure 4 illustrates, the above process may also be run in reverse:

R1 *Model Extraction*: Using various reverse engineering techniques [vZ93], a formal model expressed in CSP may be extracted.

R2 *Traces Generation*: The theorem prover may be used to automatically generate traces based on the laws of concurrency and the embedded knowledge of CSP.

R3 *Analysis*: Traces may be analyzed, used to check for various conditions, undesirable situations arising, etc.

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

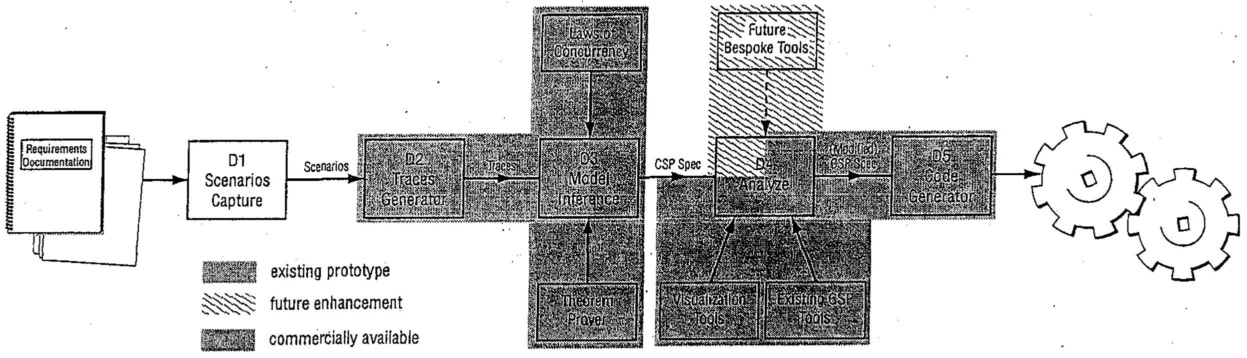

Figure 3: The entire process with D1 through D5 illustrating the development approach, and R1 through R4 the reverse engineering.

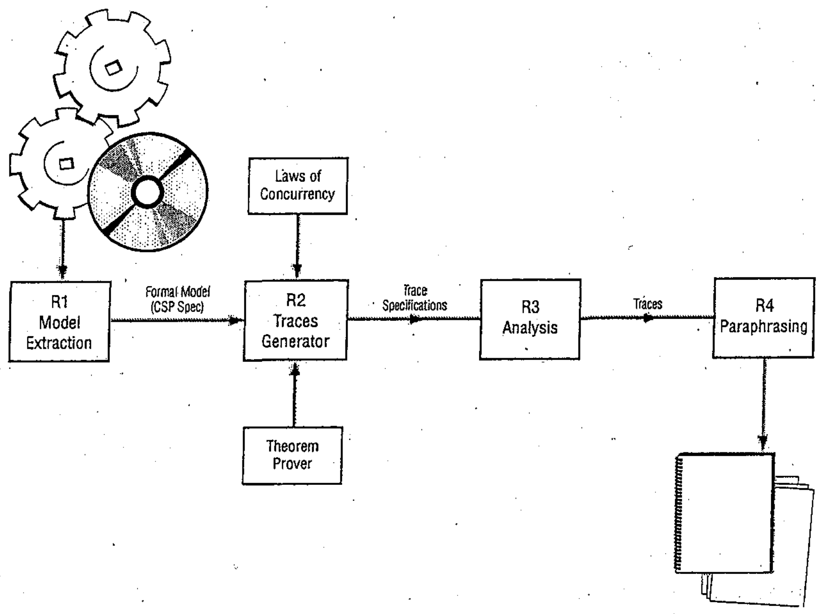

Figure 4: Reverse engineering of system using R2D2C.

R4 *Paraphrasing*: A description of the system (or system components) may be retrieved in the desired format (natural language scenarios, UML use cases, etc.).

Paraphrasing, whereby more understandable descriptions (above and beyond existing documentation) of existing systems or system components are extracted, is likely to have useful application in future system maintenance for systems whose original design documents are lost or systems that have been modified so much that the original design and requirements documents do not reflect the current system.

3.2 Short-cut R2D2C

The approach described in Section 3.1 is the way that R2D2C is intended to be applied, from requirements specification through code generation. The approach, however, requires significant computing power in the form of an automated theorem prover performing significant inferences based on traces input and its "knowledge" of the laws of concurrency. While this is well warranted for certain applications, it is likely to be beyond the resources of many developers and organizations. As a practical concession, we also define a reduced version of R2D2C called the "shortcut version" (Figure 5), whereby the use of a theorem prover is avoided, yet without sacrificing high confidence in the validity of the approach. The following describes each of the phases for the shortcut R2D2C:

S1 *Scenarios Capture*: As before, intended system behavior is described by scenarios input in natural language, or an appropriate graphical or semi-formal notation.

S2 *Translation to Intermediate Notation*: Scenarios are translated to an intermediate notation, termed EzyCSP, which is a simple natural language-like subset of CSP that can be used to describe a large number of situations and scenarios (recall that scenarios are domain specific).

S3 *Analysis*: While far more simple than CSP, EzyCSP allows some simple analyses to be performed.

S4 *Implementation in Java*: EzyCSP is sufficiently simple that it may easily be translated to Java and executed.

This simplified or shortcut approach clearly has significant disadvantages when compared to our full approach. First, the correctness of the development process is contingent on the correctness of both the translation of scenarios to the intermediate (EzyCSP) notation and the translation of EzyCSP to Java. However, the correctness of the translators for these is assured via a proof of correctness undertaken with the ACL2 theorem prover. Second, we do not obtain, for free, a reverse process suitable to support reverse and (ultimately) re-engineering; however, a Java-to-EzyCSP translator would certainly be possible for highly constrained subsets of Java.

The significant advantage of this simplified approach, however, is that although a proof of correctness involving a theorem prover is still required, this is required exactly once and would be performed by the support system developers (presumably expert in the art). This is significantly less expensive computationally than using a theorem prover in the development of each individual application.

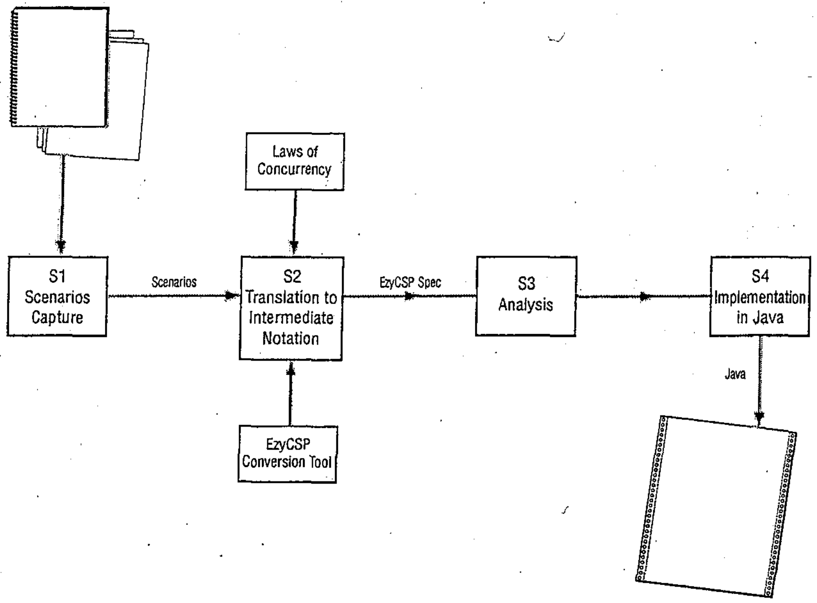

Figure 5: Short cut R2D2C.

4 A Simple Example

The Lights-Out Ground Operating System (LOGOS) is a proof-of-concept NASA system for automatic control of ground stations when satellites pass overhead and under their control. The system exhibits both autonomous and autonomic properties [TRRH04] [THRRar], and operates by having a community of distributed autonomous software modules work cooperatively to perform the functions previously undertaken by human operators using traditional software tools, such as orbit generators and command sequence planners. A post-implementation formal specification of the system was undertaken in CSP. The interested reader is directed to [RRH00] and [HRR01] for a detailed discussion of our experiences in applying formal methods to LOGOS. Using CSP, a number of anomalies, conflicts, and omissions in the system were discovered that had not been detected in testing and/or actual execution. This experience is typical of highly distributed systems, such as sensor networks or other multi-agent based systems where dependability is both very important and very difficult to evaluate. The same approach can be used for space-based wireless sensor network (WSN) systems where a control station is in charge of several WSNs located on spacecraft in deep space. An example is the Autonomous NanoTechnology Swarm (ANTS) [CMN+00] mission, which is at the concept development phase. This mission will send 1,000 pico-class (approximately 1 kg) spacecraft to explore the asteroid belt. The ANTS spacecraft will act as a sensor network making observations of asteroids and analyzing their composition.

4.1 Specification of LOGOS

We will not consider the entire LOGOS/ANTS related system here. Although a relatively small system, it is too extensive to illustrate in its entirety in this report. 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 from a wireless sensor network, how scenarios map to CSP. Suppose we have the following as part of one of the scenarios for the system. The elements interacting in the example scenario are the WSN Monitoring Agent (WSNMA), the Fault Resolution Agent (FIRE), and the Trending Agent (TREND).

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

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

```
inWSNMA?fault from WSN
then outWSNMA!fault to FIRE
else
inengWSNMA?data from WSN
then outengWSNMA!data to TREND
```

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

$$tWSNMA \supseteq \{\langle\rangle, \langle inWSNMA{,}fault\rangle, \langle inWSNMA{,}fault{,}outWSNMA{,}fault\rangle\}$$
$$\bigcup \{\langle\rangle, \langle inengWSNMA{,}data\rangle, \langle inengWSNMA{,}data{,}outWSNMA{,}data\rangle\}$$

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

$$WSNMA = inWSNMA?fault \rightarrow (outWSNMA!fault \rightarrow SKIP)$$
$$\mid (inengWSNMA?data \rightarrow outengWSNMA!data \rightarrow SKIP)$$

Let us now consider a slightly larger example, the Pager Agent, and illustrate its implementation in Java. The pager agent sends pages to engineers and controllers when there is a WSN 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 that 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. The following is a partial CSP description of the pager agent:

```
$PAGER\_BUS_{db\_waiting,paged} = pager.Iin?msg \rightarrow$
  $case$
      $GET\_USER\_INFO_{db\_waiting,paged,pagee,text}$
          $if\ msg = (START\_PAGING, specialist, text)$

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

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

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

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-in 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.

5 Application Areas

The motivation for this work was the need for automatic code generation for ultra-high dependability systems, but the method described in this report is applicable in a number of other areas. The following areas may find significant value from the use of this system.

Sensor Networks

NASA is currently conducting research and development on sensor networks for planetary and solar system exploration, as well as to support its Mission to Planet Earth. In addition to the ANTS mission, a similar mission is being considered to explore the rings of Saturn. Sensor networks are also being considered for planetary (e.g., Martian) exploration, to yield valuable scientific information on weather and geological aspects. For the Mission to Planet Earth, sensor networks are already being researched and developed towards capabilities for early warnings about natural disasters and climate change. With the "system of systems" nature of sensor networks, the inter-relatedness of these systems all networked together will create a level of complexity that will require a new level of dependability and a corresponding new approach to system and software development.

Projected NASA sensor networks are highly distributed autonomous systems that must operate with a high degree of reliability, especially the solar system and planetary exploration networks, which cannot (due to large distances) be controlled in real time from Earth and must operate under extremes of dynamic environmental conditions. Because of the complexity of these systems, as well as their distributed and parallel nature, they will have an extremely large state space and will be impossible to test using traditional testing techniques. The more "code" or instructions that can be generated automatically from a verifiably correct model, the less likely that people will introduce errors during the development process. In addition, the higher the level of abstraction that developers can work from, as is afforded through the use of scenarios to describe system behavior, the less likely that a mismatch will occur between requirements and implementation, and the more likely that the system will be validated. Working from a higher level of abstraction will also allow errors in the system to be more easily caught, because developers can better see the "big picture" of the system. In addition to allowing complex systems developers to work at a higher level of abstraction, R2D2C also converts the scenarios into a formal model that can be analyzed for concurrency-related errors as well as consistency and completeness, in addition to domain-specific errors.

Expert Systems

We have been studying the potential use of this approach in the development, maintenance, and verification of expert systems. In particular, we have been giving consideration to using the R2D2C method in verifying the expert system used in the NASA ground control center for the POLAR spacecraft, which performs multi-wavelength imaging of the Earth's aurora. The POLAR ground control expert system has rules written in the production system CLIPS [GR04] for automated "lights out" (untended) operation of the spacecraft. A suitable translator from CLIPS, rather than natural language, to CSP (or EzyCSP) enables us to use this technology to examine existing expert system rule bases for consistency, etc. What has proven to be of great interest, however, is the ability to generate CLIPS rules from CSP (or EzyCSP), just as we would generate code in Java or C++. POLAR ground control center personnel expect this would be a great benefit because it would give them a means of capturing expert knowledge, from natural language description through to CLIPS rules, while maintaining correctness, which heretofore has not been available.

Robotic Operations

As pointed out earlier, the "code" generated by this approach need not be specifically code in a programming language. To this end, we have been experimenting with generating code to control robotic devices. Perhaps more interesting is the use of this approach to investigate the validity and correctness of procedures for complex robotic assembly or repair tasks. We have begun exploratory work in this direction, to validate procedures from the Hubble Robotic Servicing Mission (HRSM)—for example, the procedures for replacement of cameras, etc., on the Hubble Space Telescope (HST).

6 Related Work

Harel [Har01] [HM03] has advocated scenario-based programming through UML use cases and play-in scenarios. This work differs in that it uses scenarios in the form of structured text that is easily understandable by engineers and non-engineers. In addition, the results of converting the structured text to traces and then from traces to a formal model allows us to use a wide range of formal methods tools (e.g., model checkers), which can be used to verify and validate the system.

NASA Ames has been working on the automatic translation of UML use cases to executable code, and they report success in using the approach on large applications [WSK03]. Our approach is different, however, in that we are not limited to UML use cases, nor to natural language. R2D2C will work equally well with any input mechanism whereby requirements can be represented as scenarios, and traces extracted. Our approach works equally well with graphical, mathematical, and textual requirements representations. More importantly, the key to our approach, and what makes it invaluable for high-dependability applications, is the full formal basis, and complete mathematical tractability from requirements through to code. To our knowledge, no other currently available automated development methodology can make this claim.

7 Conclusions and Future Work

R2D2C is a unique approach to the automatic derivation of ultra-high dependability systems. It is unique in that it supports fully (mathematically) tractable development from requirements elicitation through to automatic code generation (and back again). While other approaches have supported various subsets of the development lifecycle, there has been heretofore a "jump" in deriving (from the requirements) the formal model that is a prerequisite for sound automatic code generation. Yet, R2D2C is a simple approach, combining techniques and notations that are well understood, well tried and tested, and trusted. The novelty of the approach, and the part of the approach that achieves continuity in the development process, is the use of a theorem prover to reverse the laws of concurrency, and to achieve levels of inference that would be impossible for a human being to perform on all but trivial systems.

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
  - that requirements are complete and consistent
  - that implementations are true to the requirements
  - that automatically coded systems are bug-free
  - 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

Future work will include improving the quality of the embedding of CSP in ACL2, and optimizing the embedding for efficiency. We plan a plethora of support tools to allow us to easily change the level of abstraction in a formal model, to visualize various system models and changes in those models, and to aid in tracking changes through the development process (or the reverse engineering process). We plan to enhance our existing prototype to support the full version of R2D2C, to make it into a fully functional robust prototype, and to apply it to more significant examples than the one presented in this paper. Planned enhancements to the "short-cut" version of R2D2C include incorporating additional front-end translation from various system description languages, including rule-based system languages (e.g., CLIPS).

Acknowledgments

This work was funded in part by the NASA Goddard Space Flight Center Technology Transfer Office. We are grateful to Ted Mecum, Nona Cheeks, Chris Kirkman, Diana Cox, Yvette Conwell-Brown, and Keith Dixon for their support and encouragement. Denis Gracanin (Virginia Tech) and John Erickson (University of Texas at Austin) worked with us on the intermediate approach described in Section 3.2, and undertook the implementation of the prototype described in that section. The approach described in this paper is protected under United States and international Patent Applications assigned to the United States government.

References

[Bau80] 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.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[But99] Michael 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.

[CMN$^+$00] Steven A. Curtis, J. Mica, J. Nuth, G. Marr, Michael L. Rilee, and Maharaj K. Bhat. ANTS (Autonomous Nano-Technology Swarm): An artificial intelligence approach to asteroid belt resource exploration. In *Proc. Int'l Astronautical Federation, 51st Congress*, October 2000.

[FSE99] *Failures-Divergences Refinement: User Manual and Tutorial.* Formal Systems (Europe), Ltd., 1999.

[GR04] Joseph C. Giarratano and Gary D. Riley. *Expert Systems: Principles and Programming*. PWS Publishing Company, Boston, 4th edition, 15 October 2004.

[Hal90] J. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Har01] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[HB95] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, 1995.

[HB99] Michael G. Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. FACIT Series. Springer-Verlag, London, UK, 1999.

[HJ95] Michael G. Hinchey and Steven A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill International, London, UK, 1995.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, 1985.

[HRR01] M. Hinchey, J. Rash, and C. Rouff. Verification and validation of autonomous systems. In *Proc. SEW-26, 26th Annual NASA/IEEE Software Engineering Workshop*, pages 136–144, Greenbelt, MD, November 2001. NASA Goddard Space Flight Center, IEEE Computer Society Press, Los Alamitos, Calif.

[IFA00] IFAD. The VDM++ toolbox user manual. Technical report, IFAD, 2000.

[KP00] Matt Kaufmann and Panagiotis Manolios and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods Series. Kluwer Academic Publishers, Boston, 2000.

[LH96] K. Lano and H. Haughton. *Specification in B: an Introduction Using the B-Toolkit*. Imperial College Press, London, UK, 1996.

[Mat00] The MathWorks, Inc., Natick, Massachusettes. *Getting Started with MATLAB*, 2000.

[McL92] F. McLoughlin. *ADaGGIO—An Automated Directed Graph Diagramming Tool*. Master's Thesis, Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland, 1992.

[Par95] David 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.

[RRH00] Christopher A. Rouff, James L. Rash, and Michael 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, Los Alamitos, Calif.

[THRRar] 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).

[TRRH04] Walter F. Truszkowski, James L. Rash, Christopher A. Rouff, and Michael 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.

[vZ93] H. J. van Zuylen. *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley and Sons, London, UK, 1993.

[WSK03] J. Whittle, J. Saboo, and R. Kwan. From scenarios to code: An air traffic control case study. In *Proc. ICSE-25, 25th IEEE/ACM International Conference on Software Engineering*, pages 490–495, Portland, Oregon, 2003. IEEE Computer Society Press, Los Alamitos, Calif.