An Approach to Generating and Verifying Complex Scripts and Procedures

James L. Rash, Michael G. Hinchey
NASA Goddard Space Flight Center
Information Systems Division
Greenbelt, MD 20771, USA
{james.l.rash, michael.g.hinchey}@nasa.gov

Denis Gračanin
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061, USA
gracanin@vt.edu

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

Abstract

Currently available tools and methods for system development that start with a formal model of a system and mechanically produce a provably equivalent implementation are valuable but not sufficient. The "gap" that such tools and methods leave unfilled is that the formal models cannot be proven to be equivalent to the system requirements as originated by the customer. For the classes of complex 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 appropriate graphical notations, into a provably equivalent formal model that can be used as the basis for code generation and other transformations. The same approach may be applied to address computer science aspects of bioinformatics problems. Many software tools for bioinformatics have been developed using scripting languages such as Perl and Python. Scripts are developed based on a set of requirements that can be expressed using English-like statements. Using our approach, these may be used to automatically generate and validate scripts rather than write them from scratch.

1 Introduction

Scripting languages, such as Perl and Python, enable scientists, often with limited backgrounds in computer programming, to quickly develop impressive applications, combining several pre-existing programs and databases [9]. Toolkits such as Bioperl [35] and Biopython [8] provide off-the-shelf software components that can quickly be combined and used in user-defined scripts.

This has, undoubtedly, greatly contributed to the surge in Bioinformatics research, and the recent advances of the Bioinformatics industry, by enabling scientists to be directly involved in the development of software that searches complex databases for particular patterns, or patterns with particular characteristics.

But software, even software that may seem quite short and simple, is far more complex than we might expect [6, 15, 37], and continues to lag behind hardware in terms of advances in reliability, performance, and cost of ownership [36]. The advantages brought about by advances in software tools (flexibility, speed of implementation, etc.) simultaneously brings significant disadvantages, namely a lack of quality assurance: patches and additions to public codebases may be widely available, but the accuracy, correctness, and quality of such open source code may be suspect.

2 Software Quality and Assurance Issues

2.1 Background

Software quality remains a major issue for the software engineering community. Best practice holds that we apply various proven techniques to ensure that software is safe, reliable, maintainable, and well-documented.

Notwithstanding, there have been a number of high-profile software failures. Examples include:

- The Mars Polar Lander, where failure to initialize a variable resulted in the craft crash landing on the Martian surface, instead of reverse thrusting and landing softly [4].

- Ariane 5, where it was assumed that the same launch software used in the prior version (Ariane 4) could be reused. The result was the loss of the rocket within seconds of launch [26].

- In the medical domain, in the mid-1980s, the Therac-25 computerized radiation therapy machine caused death and serious injuries by administering lethal doses of radiation to patients in what has been described as the worst series of radiation accidents in the history of medical accelerators [30].

2.2 The Therac-25 incidents

The Therac-25 incidents are an interesting and relevant example of, arguably, the most significant failure of software assurance in the medical/biological field.

Therac-25 was a dual-mode linear accelerator that could deliver either photons at 25 MeV or electrons at various energy levels. It was based on Therac-20, which in turn was based on the single-mode Therac-6. While Therac-20 included hardware interlocks for safety, in Therac-25 these were software-based. Despite several Therac-25 machines operating, reportedly correctly, for up to four years at various installations in the US, six incidents occurred where the device gave massive (and lethal) doses of radiation to patients.

Subsequent investigations discovered that "creative" setting of parameters by students at a radiology school regularly resulted in Therac-20 machines shutting down due to blown fuses and breakers. In fact, it transpired that Therac-20 incorporated the same software error as Therac-25, but what was merely a nuisance in Therac-20 (due to mechanical interlocks) was a fatal problem with Therac-25 [25]. The problem was "inherited" and exacerbated in Therac-25.

2.3 Bioinformatics and software quality

In 2002, the National Institute of Standards & Technology (NIST) estimated that economic losses due to poor software quality amounted to more than US$60 billion. The FDA [37] estimates that, between 1992 and 1998, 7.7% of medical device recalls were due to software failures.

Factors that non-specialist developers of software often fail to consider include [37]:

- Even very short programs can be complex and difficult to understand, due to branching.

- Software does not deteriorate with age. In fact, it may be improved over time by the discovery and correction of latent errors. However, new defects may be introduced during changes to software.

- Seemingly insignificant changes in software can result in significant and unexpected problems in other (seemingly unrelated) parts of the code.

- While some hardware can give forewarnings of failure, this is not the case with software. Many latent errors in software may not be visible until long after the software has been deployed.

- A characteristic of software is the speed and ease with which it can be changed. This may give the incorrect impression that software errors can easily be found and corrected.

- Testing must be augmented with other verification techniques, and a structured and well-documented development approach must be combined to ensure a comprehensive validation approach.

The FDA concludes [37]: "Because of its complexity, the development process for software should be even more tightly controlled than for hardware, in order to prevent problems that cannot be easily detected later in the development process", and that "time is needed to fully define and develop reusable software code and to fully understand the behavior of off-the-shelf components."

Clearly this is not the current trend in the Bioinformatics industry, where scientists are developing scripts that are not validated, and reusing open source components that may or may not have been tested and verified. A "significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering." [18].[1]

We cannot assume that scripts and seemingly simple software is correct, without the application of state-of-the-art techniques for software safety. In fact, the European Union's Machine Safety Directive 98/37/EC requires developers to demonstrate the use of such techniques, or suffer criminal liability [5].

The application of software validation techniques to Bioperl [38] is attempting to provide an ongoing, systematic testing of Bioperl, with patches and validated new code being added to the public codebase. The goal is to establish user confidence that software components will work as described.

[1] This quotation from Frank Houston of the FDA predates the surge in the Bioinformatics industry; but Houston's criticism of the medical devices industry may equally well be applied to current trends in the Bioinformatics industry.

3 Automatic Generation of Scripts and Procedures

Automatic code generation from requirements has been the ultimate objective of software engineering almost since the advent of high-level programming languages [28]. The intent is that automating the generation of code can reduce development lead-times and costs, significantly reduce common programming errors, or bugs, and facilitate the speedy and cost-effective generation of multiple software implementations for different platforms.

While automatic programming has been maligned in the past [6], the need for "requirements-based programming", or RBP, whereby requirements can be transformed into an implementation in a manner that supports the entire lifecycle of the development process, cannot be exaggerated [11]. The Bioinformatics industry can similarly exploit the benefits of such an approach in the automatic generation of complex scripts and procedures.

Scripts and/or procedures in the Bioinformatics domain are developed based on a set of requirements. These are likely to be expressed as statements in natural language (such as English). Rather than writing scripts from scratch, requirements-based programming techniques [13, 29] may be used to automatically generate and validate these scripts. In [28], we describe how the approach may be used to generate and validate complex procedures (for the robotics domain); a similar approach may be applied in the biological domain for the validation of complex laboratory procedures. In the remainder of this paper, we will concentrate on the generation and validation of scripts.

3.1 RBP vs. Automatic Programming

Automatic Programming requires the developer to represent the system to be implemented 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 and with a correspondingly minimized chance of introducing errors. Being able to automatically produce the formal model from customer requirements further reduces the chance of introduction of errors by developers and results in highly dependable complex systems. This is the goal of requirements-based programming.

We will not critique currently available system development tools and methods that are based on formal models here; but, to the best of our knowledge, they provide neither automated generation of the models from requirements nor automated proof of correctness of the models. Therefore, currently there is no automated means of producing a system—or script, or complex procedure—that is a provably correct implementation of the customer's requirements. Further, requirements engineering as a discipline has yet to produce an automated, mathematics-based process for requirements validation.

Several tools and products exist in the marketplace to automate code generation from a given model expressed in a particular notation. However, typically the code they generate includes portions that either are never executed or 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 presuppose the existence of an explicit (formal) model of reality that can be used as the basis for subsequent code generation. While such an approach is reasonable, the advantages and disadvantages of the various modeling approaches used in computing are well known and certain models can serve well to highlight certain issues while suppressing other less relevant details [27]. It is clear that the converse is also true. Certain models of reality, while successfully detailing many of the issues of interest to developers, can fail to capture some important issues, or perhaps even the most important issues.

3.2 Our Approach

Without a formal specification of the system under consideration, there is no possibility of determining any level of confidence in the correctness of an implementation of a complex system. The formal specification must fully, completely, and consistently capture the requirements set out. Clearly, 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, through which errors, omissions, and conflicts can be identified. 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 lifecycle.

The Requirements-to-Design-to-Code (R2D2C) method described in this paper 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. In this approach, engineers (or others) may express system requirements as scenarios in constrained (domain-specific) natural language, or in a range of other notations (including Unified Modeling Language (UML) use cases). These will be used to derive a formal model that is guaranteed to be equivalent to the requirements stated at the outset, and that will subsequently be used as a basis for code generation. The formal model can be expressed using a variety of formal methods. Currently we are using CSP, Hoare's language of Communicating Sequential Processes [16, 17], which is suitable for various types of analysis and investigation, and as the basis for fully formal implementations as well as automated test-case generation, etc.

The approach may be used to reverse engineer systems (that is, to retrieve models and formal specifications from existing code), and to "paraphrase" (in natural language, etc.) formal descriptions of existing systems. 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 scripts for Bioinformatics tools (see Section 5).

4 Requirements to Design to Code

4.1 R2D2C Method

The R2D2C approach involves a number of phases, which are reflected in the system architecture shown in Figure 1. The following describes each of these phases.

D1 Scenarios Capture: 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 Generator: 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 [20]—using the traces derived in D2. A deep[2] embedding of the laws of concurrency [12] 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.

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

D5 Code Generator: 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 [3], or converting to other notations suitable for code generation (e.g., converting CSP to B [7]) and then using the code generating capabilities of the B Toolkit.

It should be re-emphasized that the "code" that is generated may be code in a high-level programming language, scripts in a language such as Perl or Python, low-level instructions for (electro-) mechanical devices, natural-language business procedures and instructions, or the like.

Paraphrasing, whereby more understandable descriptions (above and beyond existing documentation) of existing systems or scripts are extracted, is likely to have useful application in future system maintenance where the original design documents have been lost or modified so much that the original design and requirements documents do not reflect the current system. In particular, it may prove useful in determining the actual functionality of scripts, which typically have very little documentation.

It is intended that the approach not only can be used to generate new scripts, but also (in "reverse engineering mode") can be used to provide verification and validation of existing scripts. Moreover, it can be used to combine existing scripts, which have been reverse-engineered to a formal model, combined with other formal models (and checked for consistency and compatibility), and then efficiently regenerated as a new script.

4.2 R2D2C Implementation

The current R2D2C implementation translates the CSP model into Java code [10]; the derived design is transformed into an equivalent software representation. The Java programming language was selected both for tool implementation and for the target platform for the following reasons.

- 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 it does not provide direct CSP-to-Java mapping, it conforms to the CSP model of communicating systems for Java multi-threaded applications [23]. There is also support for distributed JCSP components using JCSP.net [40].
- Java Swing [39], in combination with some Java IDEs, greatly simplifies user interface development.

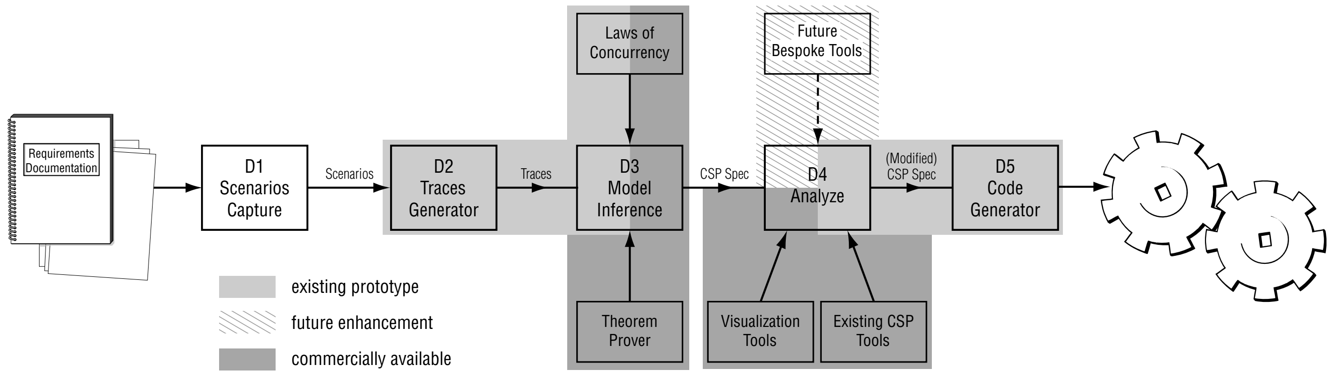


Figure 1. The R2D2C method.

- Availability of many Java-based translator development tools.

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 [34].

A planned front-end tool, a scenario editor, will support this process. An additional planned tool will enable an automated translation of constraints and restrictions into a propositional form that can be subjected to formal proof based on the CSP model. Appropriate algorithms will be developed to analyze properties of distributed systems that use a CSP-like communication infrastructure [19]. CSP models for a specific programming language or scripting language implementation further increase modeling capabilities (e.g., for Java [41]).

5 Bioinformatics Related Example: Bioperl

5.1 Background

Finding patterns in biological sequences has a goal of identifying parts that have a biological meaning [21, 22, 24]. There are several approaches to this problem. One such approach is based on the use of grammar and parsing. Ideally, one would use a grammar defining a gene or expressing protein folds. However, it is questionable as to whether such grammars are feasible. Some generic grammars describing the syntax of genes can be defined using certain grammar rules and patterns. Such grammars are usually very ambiguous and, while describing a gene structure, cannot be used to construct a sequence parser. Instead, there are many tools and programs that provide ready-made functionality for sequence manipulation, database access, invocation of molecular biology programs (e.g. Blast, clustalw, TCoffee, genscan, ESTscan, HMMER), and processing of the results.

Bioperl [35] provides a collection of perl modules used for the development of perl scripts for use in Bioinformatics applications. It also allows the development of scripts to analyze large quantities of sequence data. The user of Bioperl needs to have a basic understanding of the Perl programming language concepts. Very often these concepts are not well-known to an average user, who then needs to learn basic programming constructs quickly in order to write simple Perl scripts to perform tasks such as:

- Accessing sequence data from local and remote databases.
- Transforming database/file record formats.
- Manipulating sequences.
- Searching for similar sequences.
- Manipulating sequence alignments.
- Searching for genes and other structures on genomic DNA.
- Developing machine readable sequence annotations.
- Manipulating clusters of sequences.
- Representing non-sequence data.
- Graphically representing sequences;
- Using sequence alphabets.

```
Start sends enabled.

GeneOne receives enabled then sends gone.

ProteinOne receives gone then sends cone.

GeneTwo sends gtwo.

ProteinTwo receives gtwo then sends ctwo.

GeneOne receives ctwo then sends enabled.
```

Figure 2. R2D2C input scenario.

5.2 From scenarios to CSP

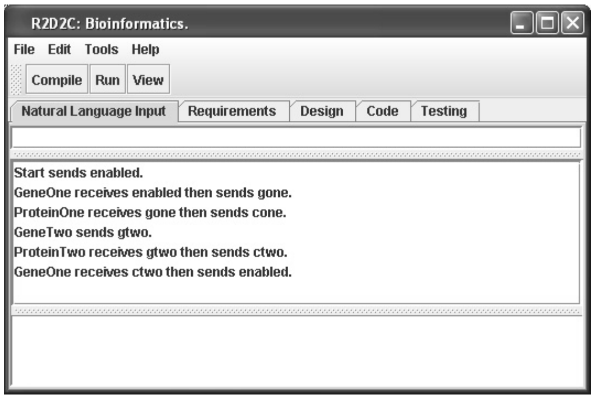
Let us consider an example from [9] (pp. 146–147). The problem is described in the form of a scenario. The scenario states [9]:

- Gene $GeneOne$ produces protein $ProteinOne$ in t_1 units of time; $ProteinOne$ dissipates in time u_1 and triggers condition $cone$.
- Gene $GeneTwo$ produces protein $ProteinTwo$ in t_2 units of time; $ProteinTwo$ dissipates in time u_2 and triggers condition $ctwo$.
- Once produced, $ProteinTwo$ positions itself in $GeneOne$ for u_2 units of time preventing $ProteinOne$ from being produced.

The scenario represents a process that is expressed and implemented using a Perl script. However, it is also possible to express this scenario using a formal model based on CSP [16, 17]. $GeneOne$, $ProteinOne$, $GeneTwo$, $ProteinTwo$ can be considered as separate processes with timing constraints implicitly included. (Timing constraints may be explicitly handled by using Timed CSP, a timed variant of CSP which extends the semantics of CSP with time [31, 32, 33].) The implicit pre-condition that $GeneOne$ must be enabled is handled by the $Start$ process. The events and conditions describing protein production are represented as messages $gone$, $cone$, $gtwo$, $ctwo$, and $enabled$. The resulting R2D2C input scenario is shown in Figure 2.

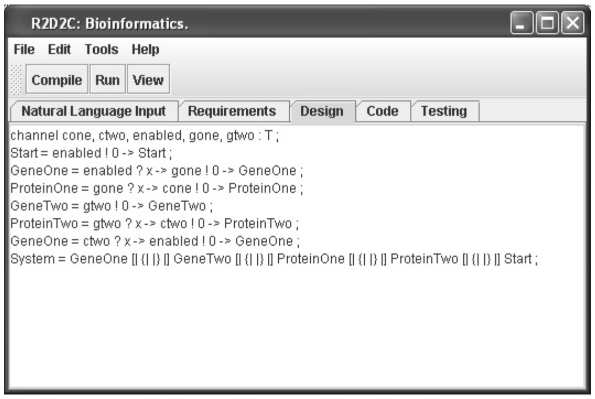
Constraints currently incorporated into the prototype R2D2C tool are at this stage minimal in number, but serve to demonstrate the potential of the R2D2C method for Bioinformatics scripts. Figure 3 shows the input (scenario), while Figures 4 and 5 show the corresponding CSP formal model.

In essence, the corresponding CSP model consists of five CSP processes that capture input requirements. The analysis of the CSP model can detect a deadlock (the script is



```
Start sends enabled.
GeneOne receives enabled then sends gone.
ProteinOne receives gone then sends cone.
GeneTwo sends gtwo.
ProteinTwo receives gtwo then sends ctwo.
GeneOne receives ctwo then sends enabled.
```

Figure 3. The R2D2C input entry.



```
channel cone, ctwo, enabled, gone, gtwo : T ;
Start = enabled ! 0 -> Start ;
GeneOne = enabled ? x -> gone ! 0 -> GeneOne ;
ProteinOne = gone ? x -> cone ! 0 -> ProteinOne ;
GeneTwo = gtwo ! 0 -> GeneTwo ;
ProteinTwo = gtwo ? x -> ctwo ! 0 -> ProteinTwo ;
GeneOne = ctwo ? x -> enabled ! 0 -> GeneOne ;
System = GeneOne [| {| |} |] GeneTwo [| {| |} |] ProteinOne [| {| |} |] ProteinTwo [| {| |} |] Start ;
```

Figure 4. The R2D2C CSP output.

blocked waiting for input) or a livelock (an endless looping) that should then be used to revise the requirements before developing the Bioperl code.

A frequent mistake in implementing these requirements is omission of constraints, either due to their "implicit" presence in the requirements, or due to errors in code development [9]. For example, consider the following revised version of the R2D2C input entry created by omitting `Start sends enabled.` (Figure 6).

In this case, nothing prevents $GeneOne$ from constantly generating $ProteinOne$ and ignoring $ProteinTwo$ inhibition.

The application of the prototype R2D2C tool to the example described above shows the potential benefit of an automated, mathematically-sound method for verifying Bioinformatics scripts. If the input requirements are not consistent or have problems (e.g., the corresponding CSP model has occurrences of deadlock or livelock), the corresponding

```
channel cone, ctwo, enabled, gone, gtwo : T ;
Start = enabled ! 0 -> Start ;
GeneOne = enabled ? x -> gone ! 0
  -> GeneOne ;
ProteinOne = gone ? x -> cone ! 0
  -> ProteinOne ;
GeneTwo = gtwo ! 0 -> GeneTwo ;
ProteinTwo = gtwo ? x -> ctwo ! 0
  -> ProteinTwo ;
GeneOne = ctwo ? x -> enabled ! 0
  -> GeneOne ;
System =
  GeneOne [| {| |} |]
  GeneTwo [| {| |} |]
  ProteinOne [| {| |} |]
  ProteinTwo [| {| |} |]
  Start ;
```

Figure 5. R2D2C CSP code.

```
GeneOne sends gone.

ProteinOne receives gone then sends cone.

GeneTwo sends gtwo.

ProteinTwo receives gtwo then sends ctwo.

GeneOne receives ctwo then sends enabled.
```

Figure 6. Revised input scenario.

Bioperl scripts are likely to have the same problems (unless the error is detected by the person coding the scripts).

5.3 From CSP to Perl

Converting from CSP to Perl can be implemented using threads. In the Perl interpreter thread model (*ithreads*), introduced in Perl 5.6.0, each thread runs in its own Perl interpreter. Any data sharing between threads is explicit using queues, special thread-safe objects.

A CSP channel is implemented as a queue (Figure 7) while CSP process channel communication is implemented by enqueueing and dequeueing the corresponding channel queues (Figure 8).

While the generated code is rather lengthy, it does guarantee the correct implementation of the requirements.

```
use threads;
use Thread::Queue;

my $enabledQueue = Thread::Queue->new;
my $coneQueue = Thread::Queue->new;
my $ctwoQueue = Thread::Queue->new;
my $goneQueue = Thread::Queue->new;
my $gtwoQueue = Thread::Queue->new;
......
```

Figure 7. Perl CSP channel code.

```
......
$GeneOne = threads->new(sub {
        while ($DataElement =
          $enabledQueue->dequeue) {
            ...
            $goneQueue->enqueue("gone");
        }
    });
......
$GeneOne->join;
```

Figure 8. Perl CSP process code.

6 Conclusions and Future Work

R2D2C is a unique approach to the automatic derivation of complex 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.

We have previously experimented with applying the approach in the automated development and post-implementation verification and validation of wireless sensor networks [14] and complex procedures for robotics [28]. The results have been extremely encouraging. Currently we are investigating applying the approach to the verification of expert systems, and complex scripts for spacecraft operation.

It is our contention that R2D2C, and other approaches that similarly provide mathematical soundness throughout the development lifecycle will be of benefit in the generation and verification of scripts. It will:

- dramatically increase quality assurance of scripts,

- ensure that scripts are true to the requirements,
- ensure that automatically-coded scripts are bug-free, and
- decrease costs and schedule impacts in the Bioinformatics domain.

Acknowledgements

This work was encouraged and funded in part by NASA Goddard Space Flight Center (GSFC) Information Systems Division and the GSFC Technology Transfer Office. We are grateful to Joe Hennessy, Ted Mecum, Nona Cheeks, Chris Kirkman, Diana Cox, Yvette Conwell-Brown, and Keith Dixon for their support and encouragement. John Erickson (University of Texas at Austin) worked with us and provided invaluable expertise on the prototype R2D2C environment. The approach described in this paper is protected under United States and International Patent Applications assigned to the United States government.

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

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

[5] J. P. Bowen and M. G. Hinchey. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, Aug. 1994.

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

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

[8] B. Chapman and J. Chang. Biopython: Python tools for computational biology. *SIGBIO Newsl.*, 20(2):15–19, 2000.

[9] J. Cohen. Bioinformatics—an introduction for computer scientists. *ACM Comput. Surv.*, 36(2):122–158, 2004.

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

[11] 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.

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

[13] 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)*. IEEE Computer Society Press, Los Alamitos, Calif., 3–8 April 2005.

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

[15] M. G. Hinchey, J. L. Rash, W. F. Truszkowski, C. A. Rouff, and R. Sterritt. You can't get there from here! Problems and potential solutions in developing new classes of complex systems. In *Proc. Eighth International Conference on Integrated Design and Process Technology (IDPT)*, Beijing, China, 13–17 June 2005. The Society for Design and Process Science.

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

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

[18] F. Houston. What do the simple folk do?: Software safety in the cottage industry. In *Proc. IEEE Computers in Medicine Conference*, 1985.

[19] S. T. Huang. A distributed deadlock detection algorithm for CSP-like communication. *ACM Transactions on Programming Languages and Systems*, 12(1):102–122, 1990.

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

[21] D. E. Krane and M. L. Raymer. *Fundamental Concepts of Bioinformatics*. Benjamin Cummings, San Francisco, 2003.

[22] S. A. Krawetz and D. D. Womble. *Introduction to Bioinformatics: Theoretical and Practical Approach*. Humana Press, Totowa, New Jersey, 2003.

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

[24] A. M. Lesk. *Introduction to Bioinformatics*. Oxford University Press, Oxford, UK, 2002.

[25] N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

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

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

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

[29] J. L. Rash, M. G. Hinchey, C. A. Rouff, D. Gračanin, and J. D. Erickson. Experiences with a requirements-based programming approach to the development of a NASA autonomous ground control system. In *Proc. IEEE Workshop on Engineering of Autonomic Systems (EASe 2005) held at the IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005)*. IEEE Computer Society Press, Los Alamitos, Calif., 3–8 April 2005.

[30] J. A. Rawlinson. Report on the Therac-25. In *OCTRF/OCI Physicists Meeting*, Kingston, Ont., Canada, 7 May 1987.

[31] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Hemel Hempstead, UK, 1997.

[32] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, London, 1999.

[33] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In *Proc. REX, Real-Time: Theory in Practice Workshop*, volume 600 of *LNCS*, pages 640–675. Springer-Verlag, 3-7 June 1991.

[34] 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.

[35] J. Stajich and E. Birney. The Bioperl project: motivation and usage. *SIGBIO Newsl.*, 20(2):13–14, 2000.

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

[37] U.S. Department of Health and Human Services, Food and Drug Administration. General principles of software validation; final guidance for industry and FDA staff, 11 Jan. 2002.

[38] P. van Heusdan. Applying software validation techniques to Bioperl. In *2004 Bioinformatics Open Source Conference*, Glasgow, UK, 29–30 July 2004. Abstract.

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

[40] 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.

[41] P. H. Welch and J. M. R. Martin. A CSP model for Java multithreading. In *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 114–122, University of Limerick, Ireland, 10–11 June 2000.