A Compliance Notation for Verifying Concurrent Systems

D. M. Atiya
Research area: Verification of concurrent high-integrity systems.

1. Introduction

Over the last four decades, growing research efforts have been dedicated to the verification of software programs. The common thread between these efforts is bringing some sort of formalism to the development process, e.g. capturing the error early in the development process by writing a formal specification, verifying that a program complies with its specification through mathematical proofs, or constructing a (correct) program by mathematically manipulating the specification. Clearly, adding formality to the system development process has the great benefit of ensuring higher degrees of safety and reliability. However, formal proofs and formal verification of development steps impose their own cost on the software development process; at least the cost for learning the associated mathematical notation and techniques, if nothing else. This created a large controversy in the computer science community about the importance of formal development of software, perhaps the most notable example of which is the question of whether conducting proofs about program correctness is better than testing the program for errors. More recently [2], researchers began to realise that the important question is not about whether the system should be formally developed or not. Rather, it is about "what degree of formality" is needed for the development process. This recent view advocates an overall development process in which both formal and informal techniques can be used hand-in-hand with each other. Typically, the degree of formality used would depend on the critical aspects of the system concerned, e.g. security and safety.

The compliance notation [8] provides such a unified framework where both formal and informal techniques can be employed in software development. The notation has been designed to give software designers and programmers a means to show, formally and/or informally, that a certain program complies with its specification. To achieve this goal, the notation employs:

1. A high-level specification language, Z [9], which allows for a formal specification of the system without worrying about details of its implementation.
2. The SPARK [1] programming language, which is a safe subset of Ada designed for use in high-integrity applications. The use of SPARK not only increases the chance of writing trustworthy programs, but also facilitates the program verification process.
3. Knuth's Literate Programming technique [3], which allows computer programs to be presented in an order more suited to human comprehension rather than the order demanded by the language compiler.
4. Morgan's Refinement Calculus [5], which allows the formality present in the program specification to be continued all the way to code.

The appealing feature of the compliance notation is that, with just one tool [4] supporting the whole notation, all the different levels of the development process can be achieved, starting from system specification in Z and ending up with a verified SPARK code. Moreover, not only are the different levels of system development accomplished within the same framework, but also a

great deal of freedom is offered for choosing the appropriate degree of formality associated with the development process. Typically, informal arguments (in the style of Knuth's literate programming) are used to reason about the correctness of the non-critical parts of the system at hand. On the other hand, formal arguments (in the style of Morgan's refinement calculus) are used only to justify the implementation of the system's critical parts. This way, the compliance notation can be used to provide a development process that falls anywhere between *strictly formal* and *totally informal.* This *polymorphic* characteristic of the notation makes it appealing to a wide range of industrial applications, from those applications that do not require formal treatment at all to high-integrity systems where formal development is a typical requirement.

2. Compliance Notation and Verification of Concurrent System

Recently, there has been growing evidence about the merit of the notation as a successful tool for system verification. The notation has been successfully applied in industrial projects, see [6]. Also, an evaluation of the notation and its tool by Praxis[1] stated they believe that the present tool satisfies the requirements of the UK's defence standard (DefStan 00-55) for safety-critical systems [6]. However, the notation is currently limited to sequential systems with virtually no support for verifying concurrent systems – our research aims to fill this gap (see section 4).

The specification language adopted by the compliance notation, Z, has shown a great deal of success in describing the functional or data aspects of systems, but it fails to capture the communication aspects of concurrent systems. Also, the programming language, SPARK, currently contains no support for tasking. Finally, concurrent systems themselves have their own intrinsic aspects that add to the challenge of the verification process. For example, the presence of multiple threads of control in concurrent systems can lead to subtle and often unanticipated interactions between the running processes; in particular, issues such as interference, race conditions, deadlock, and livelock are peculiar to concurrent programming. Also, many concurrent systems, e.g. distributed databases, are reactive, meaning that they are designed to engage in an ongoing series of interactions with their environments. Thus, unlike traditional sequential programs, reactive systems should not terminate. Consequently, traditional notions of correctness and program semantics that rely on relating inputs to expected outputs upon termination (like currently adopted in the compliance notation) are no longer valid for verifying reactive systems.

3. Verification of Concurrent Systems: a look back

Techniques based on refinement, state exploration, and model checking are effective means for program verification, and much research work has already been pursued in these areas. However, refinement techniques usually involve large number of small steps, with the possibility of proof obligations to be discharged at each step. This might turn out to be a tedious and error-prone job to do, particularly if the formal treatment is not really required for constructing all the various parts of the program at hand. Also, traditional refinement, based on pre- and postconditions as its semantic framework, may not be applicable to the construction of concurrent programs. State exploration techniques, on the other hand, are limited to finite-state programs. Thus, many real life examples of concurrent systems that are not finite-state cannot be verified using state exploration methods. Model checking can be applied to finite state programs and other infinite-

[1] Praxis Critical Systems: http://www.praxis-cs.co.uk

state programs that have special forms. However, because the number of states in the model may grow exponentially with the number of concurrently executing components, even finite-state programs may suffer combinatorial explosion problems, which limits the use of model checking algorithms or makes them inapplicable at all.

4. Our Goal

In this work, we claim that the compliance notation can be extended so as to incorporate appropriate support for concurrent systems. Although the notation is currently limited to the Z specification language and SPARK programming language, our conjecture relies on the fact that the philosophy behind the compliance notation is general and there is no reason why it could not be developed for other specification/programming languages. Thus, this research will attempt to extend the compliance notation and incorporate it with:

1. A specification language that captures the communication aspects of concurrent systems,
2. A concurrent programming language, and
3. A semantic framework that relates the programming language constructs to their specification.

If successful, the resulting system will avoid the tedium of the traditional refinement techniques, since both formal and informal arguments are used in the verification process. Also, unlike state exploration and model checking, the resulting notation will have the capability of addressing both finite-state and infinite-state systems, without the possibility of state-explosion problems. However, the three choices mentioned above are restricted by certain limitations, and careful decisions have to be made regarding the languages adopted or the semantics proposed. For example, not only should the specification language be suitable for specifying concurrent systems but also it should be amenable for refinement. The programming language, on the other hand, should exhibit typical design properties such as simplicity of formal description and verifiability, especially if this language is to be used in safety-critical applications. Finally, lessons learnt from research work on concurrency theory suggest that the current semantic framework of the compliance notation, based on weakest preconditions, is not suitable for concurrent systems development. Rather, the new semantic framework should be based on the invariant behaviour of the concurrent system under development.

5. The Research and its Evaluation

We start by determining the specification/programming language to be incorporated into the new notation. Our choice for the specification language is influenced by the work of J. Woodcock and A. Cavalcanti on the "***Circus***" language [10] for specifying concurrent systems. The language presents a combination of Z and CSP [7], and hence it is capable of representing both the state and concurrent aspects of concurrent systems. The key distinguishing feature of Circus, however, is that it is supported with a theory of refinement. Our choice for the programming language, on the other hand, is influenced by industry. Ada has proved a very successful language in industry, especially in the area of high-integrity systems. However, concurrency in Ada is hard to reason about and is not addressed yet by existing verification systems. Currently there is ongoing research at the University of York for introducing a reduced tasking model of Ada (***RavenSPARK***). RavenSPARK is intended to be a safe concurrent language, like SPARK for sequential programs, through which we can gain more confidence about the written code.

Having decided on both the specification and the programming language, the next step would be to provide the semantic framework linking the two languages together. This semantics would form the basis for generating the verification conditions (VCs) justifying the refinement steps from the program specification to the corresponding implementation. Finally, as we also aim for an automatic verification process, various algorithms have to be constructed so as to generate the VCs incurred at any refinement step within the notation. The following items represent the different stages we are going to pursue, so as to eventually reach our targets:

1. **Understanding the Compliance Tool**: This stage of work is already underway and aims for understanding how Z is embedded into the compliance tool and how the VCs, relating Z and SPARK, are generated. This work will give us a guide for how to embed Circus into the tool, and how to construct the algorithms for generating the VCs relating Circus paragraphs to RavenSPARK code.
2. **Relating CSP to Ada Tasking**: The tasking model supported by Ada is different from that provided by Circus, due to the differences between CSP processes and Ada tasks. To tackle this issue, we plan for a theoretical study of both languages as well as experiments with some small case studies. The experiments with case studies shall be used as a test of whatever theoretical result we produce, as well as a guide for further investigation to the problem.
3. **Extending the Compliance Notation**: Following the previous two stages, we come to the point of incorporating the new languages constructs into compliance notation. After getting the compliance tool to accept the new notation, we shall write the algorithms for generating the VCs incurred at the various possible refinement steps. Typically, these algorithms would ensure that the semantics of the program code is equivalent to (or at least implies) the semantics of the specification. More test cases, of larger complexities, would be required at this stage to affirm the work done.
4. **Evaluation**: This activity expands over all stages of work. In the early stages, small and simple test cases shall be used to investigate the relationships between Ada tasks and Circus processes. Then, larger test cases shall be used to test the algorithms generating the VCs. Eventually, in order to show the viability of the new compliance notation, we shall use realistic case studies of Flight Control Systems (FCS), which are known to be safety-critical applications. The FCS case studies will be provided by QinetiQ[2] and our work will be subject to a close review by their experts at all various stages. Finally, for the research community, credible evidence of our results will be provided through a series of publications in key conferences and/or journals.

6. Contributions and Related Work

Extending the compliance notation in the way presented above will provide a tractable and cost-effective way for verifying concurrent systems. Bearing in mind the wide acceptance of Ada in safety-critical systems, our work is expected to contribute to the industry of this important area of applications. In fact, our work is already of special interest to QinetiQ and intended to be used for verifying their applications on FCS.

[2] Previously ‘Defence Evaluation and Research Agency’ (DERA); URL: http://www.qinetiq.com

Related to our work is [10], where the authors aim at the calculational refinement of Circus specification to programs written in languages similar to ***occam*** or ***Handel-C***. Unlike the work in [10], we are not interested in introducing a refinement calculus to derive programs from their specifications. Rather, following the general approach of the compliance notation, our emphasis would be on setting out the rules for calculating the VCs from the program and the specification. This approach provides a direct way of calculating the VCs, which removes some of the overhead involved with the refinement process. Also, using a mix of formal and informal techniques, our framework is more applicable where the strict refinement approach is not viable (e.g. in justifying nonfunctional requirements) or too costly (e.g. in justifying informal parts of the system).

7. Conclusions

The strength of the compliance notation in verifying computer systems lies in its ability to employ both formal and informal techniques at the same time. However, the use of compliance notation is currently restricted to sequential programs. Extending the compliance notation to verifying concurrent systems is both feasible and beneficial: feasible since the notation itself is not limited to a particular specification/programming language, and beneficial since the literature still lacks a tractable way of verifying concurrent systems. In this work, we aim to incorporate the specification language Circus and the programming language RavenSPARK (both languages support concurrency) into the compliance notation. Real life FCS applications will be used as case studies for examining the merits of the new notation. Results from these case studies should provide credible evidence of how successful the new system is in verifying concurrent systems.

References

[1] J. Barnes. *High Integrity Ada: The SPARK Approach*, Addison Wesley, 1997.

[2] J. Bowen, A. Fett, and M. Hinchey (eds). *ZUM'98: the Z Formal Specification Notation*, Springer-Verlag, 1998.

[3] D. E. Knuth, *Literate programming*, The Computer Journal, Vol. 27, No. 2, pp. 97-111, 1984.

[4] Lemma 1 Ltd, *ProofPower Compliance Tool: User Guide*, 2000.

[5] C. C. Morgan. *Programming from Specifications*, 2nd ed., Prentice Hall International, 1994.

[6] C. M. O'Halloran and A. Smith. *Don't Verify, Abstract!*, in Proceedings of ASE98, pp. 53-62, 1998.

[7] S. Schneider, *Concurrent and Real-time Systems: The CSP Approach*, John Wiley, 2000.

[8] C. T. Sennett. *Demonstrating the compliance of Ada programs with Z specifications*, in Proceedings of the 5th Refinement Workshop, C. B. Jones, R. C. Shaw and B. T. Denvir (eds), BCS-FACS, Springer Verlag, 1992.

[9] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*, Prentice Hall, 1996.

[10] J. C. P. Woodcock and A. L. C. Cavalcanti. *A Concurrent Language for Refinement*, in 5th Irish Workshop on Formal Methods, 2001.

[11] J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus: a concurrent refinement language*, draft, June 2001.