Integrating Theories and Techniques for Program Modelling, Design and Verification
– Positioning the Research at UNU-IIST in Collaborative Research on the Verified Software Challenge

Bernard K. Aichernig, He Jifeng, Zhiming Liu* and Mike Reed

International Institute for Software Technology
United Nations University, Macao SAR, China
Email: {bka,hjf,lzm,mike}@iist.unu.edu

Abstract. This submission presents our understanding of the Grand Challenge and propose an agenda on how we will position our research to contribute to this world-wide collaborative research project.

1 Introduction

The goal of the Verifying Compiler Grand Challenge [17] is very easy to understand. It is to build a verifying compiler that

> "uses mathematical and logical reasoning to check the programs that it compiles."

The fundamental problems are how to obtain and document the correctness specification of a program and how to fully automate the verification/checking of the documented aspects correctness. While accomplishing this goal now requires effort from "(almost) the entire research community" including theoretical researchers, compiler writers, tool builders, software developers and users, these two interrelated problems have been the major concern of the formal methods research community in the past several decades. Each of these two problems is and will continue to be a focus in the research on the challenge.

For a certain kind of programs, some correctness properties, such as termination of sequential programs, mutual exclusion, divergence and deadlock freedom of concurrent systems, are common and have a uniformed specification for all reasonable models and codes. They can be generated from the specification and the code and there is no need to document them. Theories and techniques for verifying these kinds of properties are nearly mature, though a lot of work may still be required in the areas of efficient and effective decision procedure design.

* This work is partially supported by the project HighQSofD funded by Macao Science and Technology Development Fund, the NSFC projects 60673114 and 863 of China 2006AA01Z165.

These techniques are readily applicable to analysis and verification of these properties of legacy code and open source. This is also the case for typing correctness for programs written in typed languages.

However, for general correctness properties, the problem of how to specify and document them is far more challenging. The techniques for their verification depend on the model that is used for the specification. In some frameworks, one may translate the code into a specification notation or model and then use the techniques and tools developed for that notation or model to analyse the translated code. The model checking tools are based on this approach. With an approach of this kind, the main difficulties lie in

1. the automated, or even manual, procedure of abstraction in the translation is difficult, especially with the constraints that the resulting abstract model can be used for efficient checking,
2. one has to study what, where, when and how the correctness properties, i.e. "assertions and annotations", are produced and documented,
3. it is still challenging to identify properties that can be verified compositionally, and to make the specification notation and model to support more compositional analysis and verification,
4. there is a need of great of research to make the tools effective and efficient even with specified correctness criterion.

In our view, there is quite a long way for theories and techniques to be mature for solving the first three problems, and solutions to these problems will be useful for dealing with the fourth problem.

Some other paradigms have been developed from the idea of *proof outlines* based on Hoare Logic. There, correctness properties are documented as assertions and annotations at certain points of a program. These approaches to some extent avoid the first problem described above, and either a deductive proof (e.g. with a theorem prover) or a simulation proof (e.g. by a model checker) can be applied. It also allows a combination of these two verification techniques. However, the other three problems remain.

A conclusion that we can draw from the above discussion is that working towards a Verifying Compiler still needs a great amount of investigation in, among other areas discussed in the description of the Grand Challenge [17], new ways of modelling to provide better support to

1. separation of concerns, specification and analysis at different levels of abstraction and better compositionality,
2. integration of formal methods with the effective practical engineering methods,
3. unifying different formal theories of programming and verification to make it possible for the verifying compiler to "work in combination with other program development and testing tools" [17],
4. development of design techniques to ease the difficulties in identification and generation of correctness criterion and the analysis and verification procedures.

This in fact has been the main theme of the research at UNU-IIST, and we can now focus on this even better directed goal of the Grand Challenge.

2 The Grand Challenge Related Research at UNU-IIST

We outline in this section our approach to the challenge with a discussion on the research problems, and a summary of progress we have made so far.

Our overall research will be centered on the issues listed at the end of the previous section in the conclusion of our understanding about the challenge. However, we will organise the research within the UNU-IIST project on Component-Based Modelling and Design.

2.1 The theme

We are developing an approach that allows a system to be designed by composing *components*. The aim is compositional design, analysis and verification. To achieve this aim, it is essential that the approach supports multi-view modelling, and allows separation of concerns. Different concerns are described in different viewpoints of a system at different levels of abstraction, including interfaces, functional services, synchronization behaviour, interaction protocols, resource and timing constraints. Our approach integrates a state-based model of functional behaviour and an event-based model of inter-component interaction [15, 14]. The state-based model is for white-box specification in support of component design, and the event-based model is for black-box specification used when composing components [15].

Multi-view modelling It is crucial that the model supports abstraction with information hiding so that we can develop refinement and transformation based design techniques. This will provide theoretical foundation for integration of the formal design techniques with practical engineering development methods. Design in this way can preserve correctness to a certain level of abstraction, and good design techniques and models even support code generation that ensures certain correctness properties (i.e. being correct by construction [28]) and helps in generation and documentation of assertions and annotations. Refinement in this framework characterises the *substitutability* of one component for another. It involves all the substitutability of all the aspects, but we should be able to define and carry out refinement for different features separately, without violating the correctness of the other aspects. We are investigating different design techniques for different correctness aspects supported by the refined calculus. We hope that the refinement calculus permits *incremental and iterative* design, analysis and verification. This is obviously important for scaling up the application of the method to large scale software development, and for the development of efficient tool support. We believe being incremental and iterative is closely related and complementary to being compositional, and important for lowering the amount of specification and verification and the degree of automation.

However, the different aspects of correctness are often related. A big challenge is to solve the problems of consistency among the specifications of the different views, and provide solutions for their consistent integration and transformation. The solutions to these problems are needed to provide theoretical support to development of tools for checking the consistency and carrying out the transformation and reasoning about the correctness of the transformation. Formal specifications of different aspects and their conditions of consistency are give in our papers [15, 14].

Multi-view analysis and verification techniques Analysis and verification of different aspects of correctness and substitutability have to be carried out with different techniques and tools. Operational simulation techniques and model checking tools are believed to be effective for checking correctness, consistency and refinement of interaction protocols, while deductive verification and theorem provers are found better suited for reasoning about denotational (or pre and postcondition based) functionality specification. Where fully automated verification is not possible one has to rely on approximated results. Here, testing can play a role to confirm the approximately verified property. Another important aspects of testing is the validation of the abstract properties to be verified. Fault-based testing adds an additional dimension to verification. Here, test cases are designed by mutating the specification to detect faults that violate the specification [2]. Also, different modelling notations will affect the way of test case generation. Therefore, The combination of verification, testing and design needs further exploration.

Integrating component-based and object-oriented techniques A component may not have to be designed and implemented in an object-oriented framework. However, the current component technologies such as COM, CORBA, and Enterprise JavaBeans are all built upon object-oriented programming. Object programs are now widely used in applications and many of them safety critical. This project also studies the techniques of modelling, design and verification of object systems and the construction of component systems on underlying object systems [15, 23].

2.2 Initial progress

A number of formal notations and theories have been well-established and proved themselves effective as tools in dealing with different aspects of computing systems. For component-based systems, analysis, design and verification can thus be carried with different techniques and tools. However, integration of components requires the integration of the theories for ensuring correctness and substitutability. In particular, we need an underlying execution model of component systems. UNU-IIST has developed a model and calculus, called **rCOS** [12], for component systems [31], The calculus is applied to formal use of UML in requirement analysis [22, 20], design [33], and consistent code generation [25]. In

rCOS, we define a component with provided and required interfaces and their functional specifications [21, 15]. Composition and refinement of these kinds of components are also defined. The research is based on Hoare and He's Unifying Theories of Programming (UTP) [18] and aim to advance UTP to for analysis of object-oriented programs and component systems. Challenging as it is, the initial results show that it is promising that "many aspects of correctness of concurrent and object-oriented programs can be expressed by assertions" [17].

Our approach facilitates assurance of global refinement by local refinement via integration of the event-based simulation and the state-based refinement. Global refinement is usually defined as a set containment of system behaviours [14], and can be verified deductively within a theorem prover. Local refinement is based on specification of individual operations [15, 14], and can be established by simulation techniques using a model checker. Promising results have also been achieved in unifying different verification methods [1, 10, 24].

The research also indicates what kind of language mechanisms, such as inheritance with attribute hiding, method overwriting and method reentry calls, are likely to cause bugs in a program. They should be avoided if possible and when they are used assertions should be inserted and verification effort should be concentrated on these assertions. To fully verify different kinds of correctness aspects, we require different verification methods (simulation, deduction and testing) and tools (model checkers, theorem provers and test case generators).

Component-based systems When we specify a component, it is important to separate different views about the component [21, 15, 14]. From its user's (i.e. external) point of view, a component C consists a set of *provided services* [31]. The syntactic specification of the provided services is described by an interface, defining the operations that the component provides with their signatures. This is also called the *syntactic specification* of a component. Such a syntactic specification of a component does not provide any information about the effect, e.g. the *functionality* of invoking an operation of a component or the *behavior*, i.e. the temporal order of the interface operations, of the component. However, it describes the syntactic dependency on other components.

rCOS contains notations for the description of the following notions for component-based systems, which serve different purposes for different people at different stages of a system development:

- Interfaces: describe the structure nature of a component and are only used for checking syntactic dependency and compositionality.
- Guarded Designs: specify the behaviours of service operations of an interface. It describes both the condition 9as the guard) imposed on the environment for use of a service operation (the design), and the behaviours of execution of a service once invoked.
- Protocols: impose the order on use of services of an interface. They describe the way by which clients can interact with an interface, and are used to avoid deadlock when putting components together.

– Contracts: are specifications of interfaces. A contract associates an interface to an abstract data model plus a set of functional of its services specified as guarded designs, and as well as an interaction protocol.
– Components: are implementations of contracts. The designer of a component has to ensure that it satisfies its contract. Its code is used by the verifier to establish this satisfaction relation.
– Combinators: are defined for interfaces, contracts and components so that they can be composed in different ways. Their algebraic properties are used to verify design of middlewares (such as CORBA) and design patterns.
– Substitutivity: is defined in terms of refinement that integrates both state-based refinement and event-based simulation.
– Coordinators: are modelled as *processes* and used to coordinate the activities and interactions of a group of components [7].

The model provides a definition of consistency among the elements of a contract and a method for consistency checking. It also allows to extend a contract by adding more services and imposing more constraints on use of services.

Component are *passive* entities that provide functional services to be invoked. Active entities are modelled as processes that are to coordinate (schedule and glue) components. We have proven that components composed with processes that coordinating them form another component. For details, we refer the reader to the technical report [7].

Design and verification of object-oriented Programs We use **rCOS** to define an object-oriented language with subtypes, visibility, reference types, inheritance, type casting, dynamic binding and polymorphism. The language is sufficiently similar to Java and C++ and can be used in meaningful case studies and to capture certain difficulties in modelling object-oriented designs and programs.

Our semantic framework is *class-based* and refinement is about *correct* changes in structure and methods of classes. The logic is a *conservative extension* of the standard predicate logic. We define the traditional programming constructs, such as conditional, sequential composition and recursion, in the exactly same way as their counterparts in the imperative programming languages without reference types. This makes our approaches more accessible to users who are already familiar with the existing imperative languages. Also, all the laws about imperative commands remain valid without the need of reproving.

The calculus relates the classic notions of refinement and data refinement [3, 16, 27] in imperative languages to refactoring and object-oriented design patterns for responsibility assignments [9, 19] This takes the initial attempts in formalisation of refactoring in [29, 32] a step forward by providing a formal semantic justification of the soundness of the refactoring rules, and advance the theories in [4–6, 30] on object-oriented refinement to deal with large scale object-oriented program refinement with refactoring, functionality delegation, data encapsulation and class decomposition. Within the calculus, we have already proved the

soundness of several design patterns, including Expert pattern, Low Coupling pattern, and High Cohesion pattern [12].

Object-oriented models of requirement and design **rCOS** is also applied to formal use of UML in requirement analysis [22, 20], design [33], and consistent code generation [25]. We have provided a unified semantic definition for UML models of requirement and design. This semantics framework covers

- Use case model
- Class model
- Object diagram
- Interaction diagram

The unification is important and useful in dealing with consistency between different models. It in fact deals with the most informal aspects of UML, including description of use cases and the links between different UML diagrams in system development. On the other hand, the formalism still keeps the roles and views of these models clearly separate: class models correspond the program state, while the use case model describes the required services and external behaviour and the sequence diagram realise the external behaviour with internal object interactions.

The refinement calculus developed in **rCOS** is used for transformation of models that preserve certain properties. The calculus deals with refinement of class diagrams to increase its capacity in supporting more use cases. This implies the support to an incremental system development such as the Rational Unified Process. The formalisation of UML in our notation allows us to transform UML diagrams consistently and to formally define and reason about transformations of UML diagrams, such as decomposing a class into several classes, adding classes, associations, changing multiplicites of associations, etc. Moreover, **rCOS** also supports the following refinements to a class diagram:

- adding a new class
- introducing inheritance
- moving an attribute or a method from a class to its direct superclass
- introducing a fresh superclass to an existing class
- copying a method of a class to its direct subclass

Combination of the object-oriented and component-based methods **rCOS** [12–14, 23] also provides a consistent combination of the object-oriented and component-based methods. In general, a component in our proposed model can be realized by a family of collaborating classes. Therefore, for a component C, we treat the interface methods of C and the protocol as the specification of the use cases of the component and the components in environment of C as the actors of these use cases. The design and implementation of this component can then be carried out in a UML-based object-oriented framework.

The types of the fields (or attributes) of components can be classes. The classes and their associations form the information (data) model. This model can be represented as a UML class diagram and formalized as class declaration in **rCOS** [12–14, 23]. The implementation of a component is based on the implementation of the class model. For example, in UML2.0, a port of a component is realized by a class too.

2.3 The research problems and program

The research program of the project is also incremental and iterative and it is roughly outlined below:

1. start with modelling, design and verification of sequential object systems,
2. deal with components with only interfaces and specification functional services
3. add synchronous interactions
4. add asynchronous interactions
5. add coordinators and containers to manage interaction among components
6. deal with resource and timing constraints of embedded systems
7. add features of fault-tolerance, security and survivability
8. extend it to internet-based programming
9. test the techniques with case studies (including analysis of middlewares and design patterns), and evaluate the results by looking at how it can support tool development
10. more foundational research include a logic for object-oriented reasoning and component-based reasoning

In fact the project started two years ago and it will be a long lasting project. The aims and focus may be adjusted along the progress of the research.

3 International Collaboration

The research will be conducted in a close collaboration with Tata Research Development and Design Centre, the largest industry research development and design centre in India. They have a great interest in applying formal methods in their tool development. UNU-IIST is now establishing a collaboration project on Scaling up Formal Methods for Large Software Development. We will investigate how the research results at UNU-IIST in theories and techniques of program modelling, design and verification can be used in the design of software development tools at TRDDC. A separate submission focusing on tool support is also accepted for presentation at this working conference to be considered for a presentation about the collaboration [26].

UNU-IIST has recently joined the NoE of ARTIST II and the collaboration with the other partners, such as Aalborg University (Denmark) and Uppsala University (Sweden) on component-based development and verification will become closer.

We have also a long tradition of collaboration with the University of Macau, Peking University, Nanjing University and Software Institute of the Chinese Academy of Sciences, Oxford University, University of Minho (Portugal) and the University of Leicester (UK).

Acknowledgement: This proposal is made on behalf of all members of the academic staff at UNU-IIST. We would like to thank our colleagues, Chris George, Dang Van Hung and Tomasz Janowski for the discussions.

References

1. B. Aichernig and J. He. Testing for design faults. Submitted to Formal Aspect of Computing. 2005.
2. B.K. Aichernig. Mutation Testing in the Refinement Calculus. Formal Aspect of Computing. 2003.
3. R. Back and L.J. von Wright. Refinement Calculus. Springer, 1998.
4. R. Back, *et al.* Class refinement as semantics of correct object substitutability. Formal Aspect of Computing 2, 18-40, 2000.
5. P. Borba, A. Sampaio and M. Cornelio. A refinement algebra for object-oriented programming. In Proc of ECOP'2003, Lecture Notes in Computer Science 2743, 457–482, 2003.
6. A. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language. Lecture Notes in Computer Science 1709, 1439–1460, 1999.
7. X. Chen, Z. Liu, and J. He. A theory of contracts. Technical Report UNU-IIST Report No 335, UNU-IIST, P.O. Box 3058, Macao, May 2006.
8. Martin Fowler. Refactoring, Improving the Design of Existing Code. Addison-Wesley, 2000.
9. E. Gamma, *et al.* Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
10. J. He. Linking simulation with refinement. In Proc of the 25th Anniversary of CSP, Lecture Notes in Computer Science 3525, 61–75, 2005.
11. J. He and C.A.R. Hoare. Unifying theories of concurrency, in Proc of ICTAC'2005, 2005.
12. J. He, Z. Liu, X. Li and S. Qin. A relational model of object oriented programs. In Proc of APLAS'2004, Lecture Notes in Computer Science 3302,415-437, 2004.
13. J. He, Z. Liu, and X. Li. rCOS: A refinement calculus for object systems. Technical Report UNU-IIST Report No 322, UNU-IIST, P.O. Box 3058, Macau, March 2005.
14. J. He, Z. Liu, and X. Li. A theory of contracts. Technical Report UNU-IIST Report No 327, UNU-IIST, P.O. Box 3058, Macau, July 2005.
15. J. He, Z. Liu and X. Li. Component-based software engineering – the Need to Link Methods and their Theories. In Proc of ICTAC05, Theoretical Aspects of Computing, the International Colloquium, Lecture Notes in Computer Science 3722, 72-97, 2005.
16. C.A.R. Hoare, *et al.* Laws of Programming. Communications of the ACM 30: 672–686, 1987
17. C.A.R. Hoare. The verifying compiler. Journal of ACM, 50(1): 63–69, 2003.
18. C.A.R. Hoare and J. He. Unifying theories of programming. Prentice Hall, 1998.
19. C. Larman. Applying UML and Patterns. Prentice-Hall International, 2001

20. X. Li, Z. Liu, J. He and Q. Long. Generating prototypes from a UML model of requirements. In Proc of ICDIT'2004, Lecture Notes in Computer Science 3347, 255–265, 2004.
21. Z. Liu, J. He and X. Li. Contract-oriented development of component systems. In Proc of IFIP WCC-TCS'2004, 349–366, 2004.
22. Z. Liu, J. He, X. Li and Y. Chen. A relational model for object-oriented requirements in UML. In Proc of ICFEM'2003, Lecture Notes in Computer Science 2885, 641–665, 2003.
23. Z. Liu, J. He, and X. Li. rCOS: A refinement calculus for object systems. In *Proc. FMCO 2004, LNCS 3657*, pages 183–221. Springer, 2005.
24. Z. Liu, A. Ravn and X. Li. Unifying proof methodologies of Duration Calculus and Linear Temporal Logic. Formal Aspect of Computing 16(2), 140–154, 2004.
25. Q. Long, Z. Liu, J. He and X. Li. Consistent code generation from UML model. In Proc of ASWEC'2005, IEEE Computer Press. 2005.
26. Z. Liu and R. Venky. Tools for formal software engineering. In Proc of IFIP Working Conference on Program Verifier Challenge, 2005.
27. C.C. Morgan. Programming from Specifications. Prentice Hall, 1994.
28. A. Pnueli. Looking ahead. Workshop on the Verification Grand Challenge, SRI International, 2005.
29. D.B. Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champain, 1999.
30. K.R.M. Leino. Recursive object types in a logic of object-oriented programming. Lecture Notes in Computer Science 1381, 1998.
31. C. Szyperski. Component Software. Addison Wesley, 1998.
32. L.A. Tokuda. Evolving Object-Oriented Designs with Refactoring. PhD thesis, University of Texas at Austin, 1999.
33. J. Yang, Q. Long, Z. Liu and X. Liu. A predicative semantic model for integrating UML models. In Proc of ICTAC'2004, Lecture Notes in Computer Science 3407, 170–186, 2005.