Tools for Formal Software Engineering

Zhiming Liu[1] and R. Venkatesh[2]

[1] International Institute for Software Technology
United Nations University, Macao SAR, China
Z.liu@iist.unu.edu
[2] Tata Research and Design Development Centre, Pune, India
r.venky@tcs.com

Abstract. We propose a collaboration project to integrate the research effort and results obtained at UNU-IIST on formal techniques in component and object systems with research at TRDDC in modelling and development of tools that support object-oriented and component-based design. The main theme is an integration of verification techniques with engineering methods of modelling and design, and an integration of verification tools and transformation tools. This will result in a method in which a *correct* program can be developed through transformations that are either proven to be correct or by showing that the transformed model can be proven correct by a verification tool. Transformations include those for model construction and those that invoke verification tools.

1 Formal Software Engineering and the Grand Challenge

The goal of the Verifying Compiler Grand Challenge [14] is to build a verifying compiler that

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

This implies that "a program should be allowed to run only if it is both syntactically and semantically correct" [28]. To achieve this goal, the whole computing community have to deal with a wide range of issues and to overcome a great deal of difficulties, among which are [13]

1. arriving at automated, or even manual, procedures of abstraction that enables a compiler to work in combination with different program development and testing tools,
2. studying what, where, when and how the correctness properties, i.e. assertions and annotations, are identified and specified,
3. identifying properties that can be verified compositionally, and designing specification notations and models to support more compositional specification, analysis and verification.
4. making tools that are scalable even with specified correctness criteria,

In our view, theories and techniques are a long way from being able to solve the first three problems, and solutions to these problems will be useful in dealing with the fourth problem.

In this position paper, we propose the development *Formal Software Engineering* as a method to develop large software systems using engineering methods and tools that are verifiable. We propose formal modelling of requirements and design, and the automatic generation of code to achieve this. We believe that this effort will contribute towards a solution to the problems stated earlier. In particular, we propose a collaborative project on software development technology and tools that helps in *correctness by construction* [28].

1.1 The state of the art in software engineering

Software engineering is mainly concerned with the systematic development of large and complex systems. To cope with the required scale traditional software engineers divide the problem along three axes - development phases, aspects and evolutions. The development phases are - Requirements, Design and Implementation. Each development phase is divided into different aspects, such as:

- static data model, control flow or processes and operations in the requirements phase;
- design strategies for concurrency, efficiency and security in the design phase. These strategies are commonly expressed as design patterns [6]; and
- databases, user interface and libraries for security in the implementation phase.

The third axis is that of system evolution [15, 16] where each evolutionary step enhances the system by iterating through the requirements - implementation cycle. Unfortunately all aspects are specified using informal techniques and therefore this approach does not give the desired assurances and productivity.

The main problems are:

- Since the requirements description is informal there is no way to check for its completeness, often resulting in gaps.
- The gaps in requirements are often filled by ad-hoc decisions taken by programmers who are not qualified for the same. This results in rework during testing and commissioning.
- There is no traceability between requirements and the implementation, making it very expensive to accommodate changes and maintain the system.
- Most of the available tools are for project management and system testing. Although these are useful, they are not enough to ensure the semantic alignment of the implementation w.r.t a requirements specification and semantic consistency of any changes made in the system.

1.2 The state of the art of formal methods

Formal methods, on the other hand, attempt to complement informal engineering methods by techniques for formal modelling, specification, verification and refinement [30,

7]. In principle, a formal system development starts with an abstract specification and transforms it into a program through a number of refinement steps. The method is supported by a sound logical framework but it is only suited for the development of relatively small programs. In practice, only some significant properties of a part of the system are formally specified and verified for an abstract model of the implementation by a model checking tool or a theorem prover or even by hand. It is still a great challenge to scale up formal methods to industry scale because of the problems listed below.

- Each development is usually a new development with very little reuse of past development.
- There is no clear separation between requirements, design and implementation making it difficult for domain experts, architects and programmers to collaborate towards a single solution.
- Because of the theoretical goal of completeness and independence, refinement calculi provide rules only for a small change in each step. Refinement calculi therefore do not scale up in practice. Data refinement requires definition of a semantic relation between the programs (their state space) and is hard to be applied systematically.
- Given low level designs or implementations it is not easy for software engineers to build correct and proper models that can be verified by model checking tools.
- There is no explicit support for productivity enhancing techniques such as component-based development or aspect-oriented development.

Both formal methods and the methods adopted by software engineers are far from meeting the quality and productivity needs of the industry, which continues to be plagued by high development and maintenance costs. Complete assurance of correctness requires too much to specify and verify and thus a full automation of the verification is in feasible. However, recently there have been encouraging developments in both approaches. The software engineering community has started using precise models for early requirement analysis and design [26, 5]. Theories and methods for object-oriented, component-based and aspect-oriented modelling and development are gaining the attention of the formal methods community. There are attempts to investigate formal aspects of object-oriented refinement, design patterns, refactoring and coordination [3, 12, 4, 20].

1.3 Objective

The aim of this project is to combine the strengths of software engineering techniques and formal methods thus enabling the development of systems that have the assurances possible due to formal methods and productivity and scale-up achievable by methods adopted by software engineers. This will be achieved by

1. Identifying development steps and precise artifacts for each step.
2. Stating correctness criteria for each artifact.
3. Building tools that will verify the correctness of given artifacts or generate artifact guaranteeing correctness.

2 Formal Modelling of Complex Systems

This section gives a brief outline of the technique and solution to be investigated by this project. The techniques are explained using a simple example of a library system, that maintains a collection of books. Members belonging to the library borrow and return books. In order to keep the explanation simple and readable we have not been rigorous in the specification of the library system. In [25], a Point of Sale (POST) system was formally developed, including a C# implementation.

2.1 Requirements modelling

For an object system, the development process begins with the specification of functional requirements. Functional requirements of a system consists of three aspects - the state, a set of operations through which external agents may interact with the system and a set of global properties that must be satisfied by the state and operations. This can be modelled as a triple $RM = \langle S, O, I \rangle$ where S is a model of the state, O is a set of operations that modify the state and I is a set of global invariants. O is expressed a pre- post-condition pair [20]. A requirements model is consistent if each operation in O is consistent with the state model and preserves the global invariant. The model can be used to specify both object and component systems. The model can be further enhanced by adding descriptions of interaction protocols with the environment [11, 9], timing aspects, features of security, etc. A multi-view and multi-notation modelling language, such as a formalized subset of the Unified Modelling Language(UML) [27, 12], can be used to specify this model and analyzed for inconsistencies using model-checking techniques as demonstrated in [29]. The difficulty is in carrying out the analysis incrementally, a small number of use cases at a time that only involve a small number of domain classes [22].

Library requirements The state space of the library system is represented by the tuple $\langle \mathit{Shelf}, \mathit{Book}, \mathit{Member}, \mathit{Loan} : \mathit{Book} \times \mathit{Member}, \mathit{isIn} : \mathit{Book} \times \mathit{Shelf} \rangle$ where, *Book*, *Member* and *Shelf* are set of books, members and shelves in the library. *Loan* is a set of tuples representing the books that have been currently loaned to members. The association *isIn* is a set of tuples representing books that are currently on some shelf. This state space corresponds to a UML diagram and can be formalized as a class declartion section of an OO program [22, 12].

The set of operations will be $\{\mathit{Borrow}(\mathit{Member}, \mathit{Book}), \mathit{Return}(\mathit{Member}, \mathit{Book})\}$. These operations are identified from the use cases [18, 22]. The *Borrow* operation can be described as

signature : $\mathit{Borrow}(S, S' : \mathit{State}, b : \mathit{Book}, m : \mathit{Member})$
pre-condition: $\neg \exists m_1 : \mathit{Member} \bullet \langle b, m_1 \rangle \in S.\mathit{Loan}$
post-condition: $S'.\mathit{Loan} = S.\mathit{Loan} \cup \langle b, m \rangle \wedge S'.\mathit{isIn} = S.\mathit{isIn} - \langle b, m \rangle$

Return can be defined similarly.

A sample invariant is *BookInvariant*, which states that every book in the library is either on the shelf or loaned to a member. This can be stated as follows.

$$\begin{aligned} \textit{BookInvariant}(S : \textit{State}) \stackrel{def}{=} \quad & \forall b : S.\textit{Book} \bullet \exists m : \textit{Member} \bullet \langle b, m\rangle \in S.\textit{Loan} \wedge \\ & \neg\exists s : \textit{Shelf} \bullet \langle b, s\rangle \in S.\textit{isIn} \\ \vee \ & \neg\exists m : \textit{Member} \bullet \langle b, m\rangle \in S.\textit{Loan} \wedge \\ & \exists s : \textit{Shelf} \bullet \langle b, s\rangle \in S.\textit{isIn} \end{aligned}$$

More formal details about formalisation of a use-case mode and its consistency relation with a class model (i.e. the state space) can be found in [18, 22].

2.2 Design

Design involves transforming the requirements model of a system to a model for a platform or family of platforms and in the process imparting some non-functional properties such as - support for concurrent or parallel execution, performance and usability. The platform may be modelled by a tuple, $\langle S_p, O_p\rangle$ where S_p is a meta-model of the platform state and O_P is a set of platform operations which maybe combined using a set of available operators. Given a platform model a system is designed by transforming requirements state model, *S* to a design state model S_d that is an instance of the platform state meta-model, S_p and transforming each operation *o* ∈ *O* to an operation o_d, which is expressed as a composition of operations in O_P. The design step also specifies a set of design invariants, I_d that the design operations must preserve. Thus the design model is a triple, $\langle S_d, O_d, I_d\rangle$ where O_d is the set of all transformed operations and the design process consists of two transform functions $\langle T_s, T_o\rangle$ where $T_s : S \rightarrow S_d$ is the state transformation function and $T_o : O \rightarrow O_d$ is the operations transformation function. A design is correct if the two transformation functions are consistent that is the diagram in figure 1 commutes and the design operations preserve the design invariants.

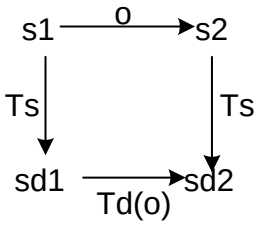

Fig. 1. Design Transformations

Library design Assume the library system is to be implemented on a platform consisting of a relational database supporting concurrent access. The platform state can be modelled by the tuple

⟨*ShelfTable*:*Table*, *BookTable*:*Table*, *MemberTable*:*Table*, *BookShelfTable*:*Table*, *BookMemberTable*:*Table*, *Process*, *hasLock*:*Table* × *Process*⟩

where, *ShelfTable*, *BookTable* and *MemberTable* are relational tables. The tables *BookShelfTable* and *BookMemberTable* are database tables having two columns each and store the relations *isIn* and *Loan*. *BookShelfTable* has columns book and shelf and *BookMemberTable* has columns book and member. *hasLock* represents the tables that have been locked by a process. The platform operations are

$\{Select, Insert, Delete, Abort, Commit\}$

These can be combined using the operators $\{;, :=, if, =, != \}$ where ';' is the sequencing operator, ':=' is the assignment operator, 'if' is the normal if, '=' and '!=' are the equality and inequality comparison operators. The platform operations have the following informal semantics.

- *Select* returns a row from the given table satisfying the given condition.
- *Insert* and *Delete* are updating operations To perform such an operation, a lock is first produced on the table that is to be updated and then performs the appropriate operation to insert or delete of a row. The operation returns `true` if the update is successful and returns `false` if a lock cannot be obtained or the update is not successful for some reason.
- *Abort* rolls back all the changes made so far by this operation and releases all the locks obtained and exits the operation.
- *Commit* releases all the locks that have been taken by this operation.

As part of the design step we define a total order $' <'$, on the tables. For the library system the order can be as given by the tuple above. The design step transforms each operation in the requirements specification into a corresponding operation in the design specification such that the updates are ordered preserving the total order on the tables thus ensuring that there are no deadlocks.

The specification for Borrow will get transformed to

```
Borrow( b : Book, m : Member)
bm := Select from BookMemberTable
      where BookMemberTable.book = b;
if ( bm != nil ) then Abort;
bs := Select from BookShelfTable
      where BookShelfTable.book = b;
b := delete bs from BookShelfTable;
if (b = false ) then Abort;
b := insert (b, m) into BookMemberTable;
if ( b = false ) then Abort; Commit;
```

A function that automatically transforms all library requirements specification to the a design specification implementing the chosen design strategy can be written. The correctness of the transformation as required by figure 1 can be shown. In addition deadlock freedom can also be shown. Since the design has been implemented as a transformation we will not have to prove correctness of the design specification for each operation, instead we prove correctness of the transformation function.

Design Patterns Different systems adopt similar design transformation functions. Therefore the process of formal design can be scaled up by abstracting away from individual design transformation functions to a design pattern. A design pattern is a meta-function that maps a requirements model to a design transformation function for that requirements model. A design pattern is correct if the mapped design functions are correct as described above. Design patterns can be proved correct independent of the requirements model making them scalable. In the presence of design pattern a design step will involve selecting and applying the appropriate design patterns.

Fotr the Library example, the design strategy of imposing a total order on the tables can be abstracted out into a transformation function. The transformation function takes the total order and a requirements specification as input and transforms the requirements of an arbitrary system into a corresponding design specification. MasterCraft [1] implements a few such design patterns for some select platforms and design strategies. MasterCraft however does not support formal specification of these design patterns. If implemented as a design pattern the atomicity preservation and deadlock freedom will not have to be proved for each application of the transformation. All we will need to show is that for a given application there exists a total order which can be conformed to while performing the transformation. We believe that is is achievable in the frameowrk of $rCOS$. The General Responsibilibity Assignment Software Pattern (GRASP) [17] are formalized as refinment rules in $rCOS$ [12]. More design patterns and pattern-directed refactoring are also studied and applied to the case study POST [2, 25].

3 Research Problems

The previous section presented an overview of a proposed method for formal development of large scale systems. To realize this method, we first need to define it more formally. We aim at a logically sound and systematic method (that we are tempted to call a *formal engineering method*) and tools that themselves are provably correct for supporting the method. The method includes:

1. *A language and a logic for specifying and reasoning about a system at different levels of abstractions.* The main task is to develop a notation for describing each aspect of correctness of a model. This will allow a developer to split a model of a system into several aspects making it more manageable. This is important for tool development too. The notation for a particular aspect should be expressive enough for describing all the concerns about that aspect. However, overlapping features among different notations should be kept to a minimum else, problems of inconsistency and integration will become overwhelming[3].
   The logic should provide a sound link among the different notations to deal with the problems of model consistency and integration. It should support compositional reasoning about the whole model by reasoning about the sub-models of the aspects. Different verification techniques and tools maybe applied to models of different aspects of functionality, interaction and structure of the system.

[3] This is a serious problem in the application of UML.

2. *A Language and logic for specifying the transform functions and reasoning about correctness.* The language should preferably be composable. That is, it should be possible to specify various design transformations independently and compose them to get a design from requirements. The techniques and tools will include formally proved pattern-directed transformations of specifications to scale up the classical calculi of refinement. We will also investigate the use of model checking and static analysis techniques and tools for consistency and analysis of properties of models. For specification and analysis of coordination among components, simulation techniques and tools can be used. Transformation of different sub-models may need different verification techniques and tools. Data refinements will be realized by structural transformations following design patterns that are scaled up from object-oriented design.
3. *Automatic code generators that implement the implementation functions for various platforms.* Refactoring transformation of designs and implementations will be studied and implemented in the tool support.
4. *Techniques and tools for domain-specific languages and their programming* (such as web-based service and transaction system based on internet).

The main theme of the project is to integrate formal verification techniques and tools with design techniques and tools of model (or specification) transformations. Verification and transformation will work complementary to ensure the correctness of the resultant specification. The design techniques and transformation tools are essential in the development to transform the requirements specification to a model that is easy to be handled with the verification techniques and tools. The design and transformation have to be carried interactively between the designer and the tool. Verification tools can be also invoked during a transformation.

This project will be conducted in a close collaboration between UNU-IIST and TRDDC. UNU-IIST is particularly strong in theories and techniques for program modelling, design and verification, and TRDDC is the largest industry research development and design centre in India. 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 position paper by UNU-IIST is also presented at this conference [13].

Related Work at UNU-IIST and TRDDC

TRDDC and UNU-IIST have been approaching the above problem from two different ends. TRDDC has expertise in software engineering techniques and has been researching this area for several years now. These efforts have resulted in MasterCraft [1], a tool that generates code for different platforms from design specifications. Current research activities at TRDDC include graph-based languages for specifying requirements [29] and transformations. The requirements group has successfully used model checking to verify correctness of requirements of a few projects. The work on transformation specifications has resulted in a proposal as a standard in response to an OMG request. The proposal is in an advanced stage of acceptance.

UNU-IIST has been working on formalizing object-oriented development. This work has resulted in a relational model for object-oriented design and an associated refinement calculus [12, 21, 10]. The refinement calculus supports incremental and iterative development [22]. The model is current being extended to support component-based development [20, 11, 9]. Initial progress have been made in experimental development of tool support [19, 24]. Promising results have been achieved in unifying different verification methods [8, 23].

4 International Collaboration

UNU-IIST has now joined as a partner 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. UNU-IIST also has 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, and the University of Leicester.

TRDDC too is involved in a lot of collaborative work. The list of currently active collaborators include - University of Aalborg, Denmark, King's College, London, University of Illinois at Urbana Champaign, Indian Institute of Science, Bangalore, Indian Institute of Technology, Mumbai and University of Wisconsin, Milwaukee.

References

1. Mastercraft. Tata Consultancy Services. http://www. tata-mastercraft.com.
2. Q. Long andJ. He and Z. Liu. Refactoring and pattern directed refactoring : A formal perspective. Technical Report UNU-IIST Report No. 318, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005. http://www.iist.unu.edu/newrh/III/1/page.html.
3. P. Borba, A. Sampaio, and M. Cornélio. A refinment algebra for object-oriented programming. In L. cardelli, editor, *Proc. ECOOP03, LNCS2743*, pages 457–482. Springer, 2003.
4. M. Broy. A theory for requirements specification and architecture design of multi-functional software systems. In Z. Liu and J. He, editors, *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*. World Scientific, To appear.
5. M. Fowler. Invited talk: What is the point of UML. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. UML 2003, Lecture Notes in Computer Science 2863*. Springer, 2003.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addlison Wesley, 1994.
7. J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
8. J. He. Link simulation with refinement. In *Proc of The 25th anniversary of CSP*, 2004.
9. J. He, X. Li, and Z. Liu. Component-based software engineering – the need to link methods and their theories. In H.V. Dang and M. Wirsing, editors, *Proc. of ICTAC05, International Colloquium on Theoretical Aspects of Computing, Lecture Notes in Computer Science 3722*, pages 72–97. Springer, 2005.
10. 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. http://www.iist.unu.edu/newrh/III/1/page.html.

11. 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. http://www.iist.unu.edu/newrh/III/1/page.html.
12. J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Pro. APLAS'2004, Lecture Notes in Computer Science*, Taiwan, 2004. Springer.
13. J. He, Z. Liu, and M. Reed. Theories and techniques of program modelling, design and verification: positioning the research at UNU-IIST in the collaborative research on the program verifier challenge. Submitted to IFIP Working Conference on Program Verifier Challenge, 2005.
14. C.A.R. Hoare. The verifying compiler: A grand challenge for computer research. *Journal of the ACM*, 50(1):63–69, 2003.
15. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
16. P. Kruchten. *The Rational Unified Process – An Introduction (2nd Edition)*. Addison-Wesly, 2000.
17. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
18. X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
19. X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML model of requirements. In *International Conference on Distributed Computing and Internet Technology(ICDIT2004), Lecture Notes in Computer Science*, Bhubaneswar, India, 2004. Springer.
20. Z. Liu, J. He, and X. Li. Contract-oriented development of component systems. In *Proceedings of IFIP WCC-TCS2004*, pages 349–366, Toulouse, France, 2004. Kulwer Academic Publishers.
21. Z. Liu, J. He, and X. Li. A model of refinement for object-oriented and component systems. In *FMCO 2004: International Symposuim on Formal Methods of Component and Object Systems. Lecture Notes in Computer SCience*, page to appear, Leiden, the Netherlands, 2004. Springer.
22. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for object-oriented requirement analysis in UML. In *Proc. of International Conference on Formal Engineering Methods, Lecture Notes in Computer Science*, Singapore, November 2003. Springer.
23. Z. Liu, A.P. Ravn, and X. Li. Unifying proof methodologies of Duration Calculus and Linear Temporal Logic. *Formal Aspects of Computing*, 16(2), 2004.
24. Q. Long, Z. Liu, J. He, and X. Li. Consistent code generation from uml models. In *Australia Conference on Software Engineering (ASWEC)*. IEEE Computer Scienty Press, 2005.
25. Q. Long, Z. Qiu, Z. Liu, L. Shao, and J. He. POST: A case study for an incremental development in rCOS. In H.V. Dang and M. Wirsing, editors, *Proc. of ICTAC05, International Colloquium on Theoretical Aspects of Computing, Lecture Notes in Computer Science 3722*. Springer, 2005.
26. S.J. Mellor and M.J. Valcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, 2002.
27. OMG. The Unified Modeling Language (UML) Specification - Version 1.4, September 2001. Joint submission to the Object Management Group (OMG) `http://www.omg.org/technology/uml/index.htm`.
28. A. Pnueli. Looking ahead. Workshop on The Verification Grand Challenge February 21–23, 2005 SRI International, Menlo Park, CA.
29. U. Shrotri, P. Bhaduri, and R. Venkatesh. Model checking visual specification of requirements. In *International Conference on Software Engineering and Formal Methods (SEFM 2003)*, page 202209, Brisbane, Australia. IEEE Computer Society Press.
30. J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.