Monotonicity and Lattices as a Foundation of Component-Based Systems

Philipp Heuberger*
Turku Centre for Computer Science and Åbo Akademi
DataCity, 3rd floor
Lemminkisenkatu 14A
FIN-20520 Turku, Finland
Tel: (358) 2 265 4044
Fax: (358) 2 265 4732
Email: pheuberg@abo.fi

Abstract

Software components are modules that are replaceable by other modules in a composed software system. The same substitutability occurs in mathematics in the form of replacing a function argument by an other.

Refinement calculus introduces a refinement relation as a substitution criteria and is more general than the equivalence approach. This generality allows the *programming by a development contract.*

The mathematical structure of a lattice is subsumed within the notion of *programming by a development contract.* A lattice has two binary choice operations called meet and join. The choices may be interpreted as the respective choices of two contract parties, respectively one contract partner and all others during the development of a program. The combination of a lattice structure with the monotonicity of refinement yields a good candidate for the foundation of component-based systems.

Keywords: programming by contract, object-oriented development, algebra, lattice theory, refinement calculus

Workshop Goals: encourage interdisciplinary work between discrete mathematics and software engineering, analyze the background of research communities, identify and propose new research topics, render the referee process more transparent.

*This work has been financed by the Swiss-TMR-Fellowship of the Federal Office for Education and Science and is supported by The Swiss National Science Foundation.

1 General Introduction

Software composition is a very mathematical discipline. Nevertheless it appears that mathematicians and computer scientists have deviating goals. Whereas mathematicians formalize a problem to understand it, computer scientists specify a problem to solve it. A common interest is the formalization or specification aspects of languages. Composability definitely qualifies as a language aspect of common interest.

In the development of object-oriented programs in a team, every member has to distinguish between independent tasks and interfering activities. Software interfaces have to be communicated among the members in order to provide the basis for the distinction. Only by means of efficient communication, cooperate software development becomes possible.

I would like to propose the use of *development contracts*, which define the development tasks and assign them to the contract parties. A typical concrete situation is that software architects design abstract classes which are handed over to developers who implement the extended concrete classes. The mutual dependencies of the architect and the developer have to be identified and settled in a development contract.

A development contract is not limited to the behavioral description of the abstract class. It must also contain the requirements of the extension. Although practiced in object-oriented development, this major shift of paradigm is not reflected in formal specification methods, yet.

New trends in the software industry adds importance to development contracts. Company's boundaries squeeze in between architect and developer partners and informal interface descriptions gain a new status. There is a rising need for making the description more precise and standard. The benefit is a finer grained competition, which has the potential of destructing monolithic software firms to the advantage of the customer. In future a text processing system will be composed of components from different manufacturers and the user himself or a third party composer will even have the choice between components from different manufacturers for the same purpose.

Another expected application of development contracts is to facilitate cooperation between strong competitors. In Europe, for example, the railway industry is divided into competing groups of companies. In a sector of desirable cooperation, imprecision hampers the development of new technologies that could enable smooth transportation. Development contracts may facilitate the development of compatible software systems, which would strengthen the position of the railway in the transportation market. The existing railway system is a European asset in the world-wide competition.

2 My Background

My scientific background is both in component-based software engineering and formal methods. Already in my diploma thesis at the Swiss Institute of Technology in Zurich, I worked on the formal development of a distributed deadlock detection algorithm [SH89]. In my M.Sc. thesis [Heu91] at the University of Victoria (Canada), I have derived an algorithm in computational geometry. The derivation has been strongly influenced by the work of Dijkstra [Dij76, DS89]. The thesis presents the development from a formal specification to an executable program.

Initially, I wrote the executable program in C++, but rewrote it as soon as the programming environment Oberon was released on an available computer. Oberon[1] is a modular and therefore a component-based language, which has anticipated many of today's developments in object-oriented programming already in the 80's: type extension, type safety, dynamic loading, garbage collection, just to name the most important ones.

My enthusiasm for Oberon brought me back to the institute for computer systems, where Oberon had been developed by Niklaus Wirth and Jürg Gutknecht [Wir88, NW89, WG92] with their assistants. In my four years appointment, I had contact with colleague assistants, for example with Clemens Szyperski, Sepp Templ, Wolfgang Weck and Régis Crelier, who performed early experimental work in the now emerging field of component-based software systems.

Even though involved with practical programming exercises in student's assignments and projects, our international group headed by Beverly Sanders and consisting of Spiros Lalis, Martin Gitsels and myself was supposed to be mainly concerned with formal methods and its application to distributed systems. Spiros Lalis extended Oberon to a prototype distributed system [Lal94] and Martin Gitsels, whom I shared the office with, worked on a proof editor and a network protocol. My own work on various areas in distributed systems and object-oriented programming was mostly of theoretical nature using first UNITY [CM88] and later the refinement calculus [Mor94]. After that Sanders quit the institute, I joined the programming methodology group of Ralph Back in Finland. Ralph Back has been a pioneer in refinement calculus, which has a strong foundation in higher order logic and weakest precondition semantics.

Currently, I am in charge of the project "Building Correct and Reusable Software Components in Oberon using the Refinement Calculus" [Heu95a]. The main publications of the project are [Heu95b, Heu97e]. As a case study and practical by-product, I continue the development of the refinement tool RefStep [HH95, Heu97c]. To gain contact to other communities and experience in related methodologies, I have carried out a project [Heu96a] with the method B [Abr96] and I have published the result in [Heu96b]. Subsequently, I have been appointed to be a member of the program and steering committee of the second B conference.

Since two years, I am enrolled as a Ph.D. student at the Turku Centre of Computer Science (TUCS) and Åbo Akademi in Finland. TUCS is a graduate school that has received much attention for encouraging interdisciplinary work in computer science, mathematics and business administration. By my experience, a promising exchange of ideas exist between the disciplines of programming methodology and discrete mathematics. I argue in this position paper that a common interest focuses on the foundation of component-based systems.

3 Workshop Goals

My recent experience in the submission of drafts [Heu97b, Heu97f, Heu97a, Heu97d] is that it is difficult to find a suitable conference for my contributions. Returned referee reports either indicate that the referees have insufficient background knowledge in order to appreciate the submitted material or claim that my theoretical discussions are confused and refer to nontrivial work[2] on the subject.

[1] Oberon is the successor of Pascal and Modula-2

[2] Remark: I am not so much interested in the nontrivial work of others rather than in their comprehensible results.

My conclusion is that my background is incompatible with the background of some research communities and not welcome to others. My goal in participating in the FoCBS-workshop is to discuss the problem of publishing interdisciplinary results in particular between discrete mathematics and component-based software engineering. I want to analyze the background of the participants and explain my background in order to reach common grounds.

The necessity to exchange background knowledge also struck me when participating at ECOOP'97 in Jyväskylä - a conference with a topic of wide range and highly specialized presentations. Even contributions on closely related fields - one would expect to understand easily - turned out to consume much more effort for profound appreciation than anticipated.

Additionally, a one day workshop on component-oriented programming before the conference showed a worrying conceptual imprecision in the definition of a component. A discussion group that specifically searched for a definition was unable to come up with something sharper than "a component is anything". I agree that "a component *may be* anything", but is there no more precise definition?

It is often advised to read a couple of relevant papers from previous conferences in order to understand the background of the targeted research community and I am in favour in doing so in a sincere way. However, I believe that the publication selection process in computer science has a severe problem due to favouritism. My experience is that working out ideas myself often is more efficient than reading through the bulk of literature. Of course, there are excellent papers. - They are only hard to find! We think that the situation could be improved by rendering the referee process more transparent.

4 An analogy of software composition and function application

Software components are modules that are replaceable by other modules in a composed software system. The substitutability of a module by another sometimes is and sometimes is not valid. There are two questions: What is the difference between a module and a component and when does a module substitute another?

Substitutability is the essential concept in (discrete) mathematics. It occurs whenever a function argument is replaced by another argument. Translated to the replacement of modules, this gives an answer to the first question.

Let me identify a module with an argument and the composed software system with a function. Then, I characterize the difference between a module and a component as follows.

- A module corresponds to an argument that is considered in isolation and
- a component corresponds to an argument that is considered in the context of a function application.

After having answered the first question, can I now answer the second one? - When does the substitutability of two modules hold? Yes, I can. The answer is: when they are equivalent! The answer sounds trivial. Somebody, who is not familiar with mathematical equivalence, will think that this is just a play on words. No, this is more than a pun. The trick is that equivalence needs to be defined, which involves more than a simple paragraph in a position paper.

In fact, the approach pursued is refinement calculus is more general. Equivalence is weakened to refinement (implication) and substitutability to monotonicity. Equivalence is symmetric and symmetry is unnecessarily restrictive. If two components A and B are equivalent, it means that A may be substituted for B *and* B may be substituted for A. This is not necessary. Refinement is an order relation that is not symmetric. If component A is a refinement for B then A may substitute B in a composed software system, but not necessarily vice-versa. With refinement, a wider space of feasible replacements than with equivalence is achieved.

Why this generalization? The reason is that the replacement of abstract specifications by concrete implementations is asymmetric. Abstraction and refinement are directed in opposite directions. Abstraction omits details and refinement adds them. The abstraction level or similarly the refinement level corresponds to the degree of non-determinacy.

5 Programming by development contracts

A development contract is an abstract specification, which is agreed on at the beginning of a development. The contract contains nondeterministic choices (implicit or explicit) which are assigned to developers. The assigned choices may be decided upon by its developer separately. Separate programming (done in parallel) is the rationale for having a development contract in the first place.

A lattice has two explicit binary choice operations and until now, I have only seen the use for one or for as many as there are contract parties. How does it then fit together? - To answer this question, I need to understand the intuitive interpretation of the choices in program semantics. The essential difference is that one choice is determined benevolently in order to establish a desired goal. The other choice may be resolved arbitrarily, i.e. each variant establishes the desired goal anyways.

The purpose of having both choices in the programming by a development contract is as follows. After the completion of the contract, each developer replaces the choices assigned to him by "benevolent choice" and the choices assigned to others by "arbitrary choice". He then refines the altered contract to a concrete implementation, resolving the benevolent choice according to his own goals and criterias.

When all contract parties finally have completed their separate development, they combine their implementations together. Because each developer assumed an arbitrary choice of his contract partners, his own development is valid irrespectively of the others' choices. Provided the development was carried out according to valid refinement laws, the combined implementation satisfies the original abstract specification. A system test then should cause no trouble.

6 Software composition

There is a complication that arises from the fact that a component itself may recursively refer to (extracted) subcomponents. Because a subcomponent does not make part of the component, I have introduced what I call an *application type*, which may be considered to be a default more abstract substitute for the subcomponent. Hence, a component not only consists of what I call a *specification type*, but also of an application type for each extracted subcomponent. This is a sharp characterization of a component and enables me to describe two different tasks in component-based programming: composition and replacement (or re-implementation).

- A composition is a mapping from application types to specification types and arrows indicate the mapping. If the mapping is in conformance with the lattice order, then the composition is valid.
- In order to replace a component, its specification type has to be a refinement of each application type. A constructive approach is the calculation of the GLAT (greatest lower application type), which is the meet of the application types associated by the mapping. From the calculated GLAT, an implementation may be derived by refinement.

7 Problems to be tackled

In this section, I present two lists. The first list contains enumerated subproblems and the second list proposes actions of how to deal with them. The numbers of subproblems and proposed actions match.

P1 There exist already precise and powerful description formalism. The problem is that there are many of them. What has to be focused on is to isolate the common features of them and compare their expressive power.

P2 Having a precise formalism is not sufficient. Practical small and large projects of model character have to be carried out. Such examples projects have to be communicated effectively.

P3 A widely distributed problem is the asymmetric perception of development contracts coined by the asymmetric nature of subroutine calls. The caller is the master and the callee is the slave. Rooted in a long tradition of not accepting upcalls, libraries only offer functionality and do not demand functionality from the user - and if they demand, then they do not state it in the documentation.

P4 There exists an ignorance of practitioners (and researchers) against mathematical aspects of notations. The relevant bits of mathematics are unknown or poorly understood.

P5 The community of C,C++, and Java programmers seem to have missed the significant step of introducing modules[3]. The step from Pascal to Modula has not found much appreciation and only finds acceptance in the realm of component-based systems.

A1 My work has been influenced by papers of America and Liskov & Wing and my draft paper [Heu97b] was an attempt to bridge the gap. There is a good possibility to combine the efforts of the refinement calculus community and the behavioral subtyping community. Another candidate is the theory of object by Abadi & Cardelli.

[3] At least they did in the beginning of the 90's.

A2 I am working on a refinement tool, which itself is a project that gives material for a case study. The examples treated by the tool are small and have model character.

A3 In order to balance the contracts and make them symmetric in the sense of programming by a development contract, I associate three types to a programming language type. Two types (the implementation and the application type) reflect the views of the two contract partners and one type (the specification type) reflects the impartial contract view.

A4 The possibilities to communicate mathematical aspect of notations through computer user interfaces has not been exploited. I expect that tools as the one I am currently developing are suitable to educate high-school and lower-grade computer science and mathematics students. Mathematical education needs to be aimed at application in software engineering.

A5 Wirth and Gutknecht have carefully avoided ruining the achievements of procedure-oriented and structured programming. In the project Oberon, they added little to obtain an object-oriented language. Experience from this project has to reach a larger audience.

8 Comparison with related work

Back has been an early pioneer of refinement calculus [Bac78] and research in this topic has lead to a textbook [Mor94] and a comprehensive foundation [BvW97]. The calculus has emerged from Dijkstra's seminal work on weakest preconditions and an influential paper by Wirth [Wir71]. A monograph by Dijkstra and Scholten [DS89] approaches the underlying mathematics in an unconventional way and Wim Hesselink discloses more insight in [Hes92]. A predominant quality of refinement calculus is the elaborated interplay of specified and executable code. Different from its origins, Morgan and Back allow miraculous statements [Mor88], which is a source of an ongoing controversy. Miraculous statements are based on the lattice join operation and have no executable interpretation. Therefore, they have been excluded by the healthiness laws [Dij76] in the very beginning.

The basic concept of object-orientedness is subclassing (inheritance). An important reinforcement is *behavioral subtyping*, which has been introduced e.g. in POOL [Ame90]. Subclassing is code inheritance in its simplest form, where method signatures are compared for type compatibility. A more sophisticated comparison is achieved by the contra/covariance rules. Subtyping goes deeper and a comparison includes, similar to the refinement relation, the semantic behavior of the compared methods. Liskov and Wing have given a precise subtype definition [LW94]. To capture the semantics, they specify class methods in the two tiered style of Larch [GH93]. The subtype definition is high-level and involves method names and signatures.

Refinement calculus identifies programs with the elements of the predicate transformer lattice. The partial order of the lattice, i.e. the refinement order, relates two programs, which are at different abstraction levels. In the development of object-oriented programs, the basic elements are classes rather than programs. An obvious but often unnoticed novelty is that classes at different abstraction levels coexist: abstract and concrete classes. In the Oberon project [WG92], I have experienced how abstract classes are designed by software architects and handed over to developers who elaborate concrete classes. When only the definition[4] without an additional description of the

[4] signature of procedures in Oberon

specification and application types is given, however, developers have no means to perform their task. [Szy92] was an exceptional project.

Lattice theory is a well-understood discipline in mathematics [DP90]. Lattices are well suited to formalize models with two parties. This is particularly the case for software user and programming interfaces which can be compared with a contract between two parties [Heu95b]. Guided by the duality of lattices, the notion of an *application type*[5] is introduced. Application types are expected to play a major rôle in the formal specification of modular software components [DL95].

The draft [Heu97b] presents in a simplified setting a lattice-theoretical approach to the development of object-oriented programs. It introduces a class construct so that classes become elements in the lattice. As for action systems [BS94] and higher constructs in the B method [Abr96], syntactic sugar is added on top of a primitive language based on a primitive language as presented in [BvW97].

In difference to [BvW97], the lattice structure rather than the refinement order is postulated and the interpretation of meets and joins is initially searched for, thereby inverting the interpretation of the refinement order, which coincides with the subtype definition. The lattice order relates classes, which are at different abstraction levels. The inverted interpretation enables the calculation of abstract common superclasses as the join of implementations. This shows that meaning, which is useful in the development of object-oriented programs, can be assigned to the join operation.

9 Concluding remarks

For researchers that have committed themselves to a research topic for a long time, interdisciplinary exchange of background knowledge is a distressing activity that has to be honoured appropriately. The voluntary promotion of self-competing research should be highly appreciated and the self-critical attitude helped to become a virtue of every researcher in computer science.

The proposed idea of programming by development contracts is my own. If it has been documented before, the better.

10 Acknowledgment

I would like to thank Ralph Back and Jockum van Wright for having had the opportunity to read their manuscript. I am also grateful to the referees for the suggested literature.

References

[Abr96] J-R. Abrial. *The B-Book*. Cambridge University Pressy, 1996.

[Ame90] Pierre America. Designing an object-oriented programming lanugage with behavioural subtyping. In *REX School/Workshop*, volume 489 of *LNCS*, pages 60–90, 1990.

[5] An application type was originally called "client type", but I prefer "application type" to avoid a confusion with the client's specification type.

[Bac78] R. J. Back. *On the correctness of refinement in program development.* PhD thesis, University of Helsinki, 1978.

[BBN95] Marie Larsson Bror Bjerner and Bengt Nordström, editors. *7th Nordic Workshop on Programming Theory*, volume 86 of *Göteborg University and Chalmers University*, November 1995.

[BS94] R.J.R. Back and K. Sere. From action systems to modular systems. Technical Report Ser. A. No 154, Åbo Akademi, 1994.

[BvW97] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus.* to appear, 1997. Manuscripts, June 26th 1996 and February 18th, 1997.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DL95] Drishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report TR 95-20a, Iowa State University, 1995.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[DS89] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Hes92] Wim H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge Tracts in Theoretical Computer Science, 1992.

[Heu91] Philipp Heuberger. A constructive approach to sweep algorithms for voronoi diagrams. Master's thesis, University of Victoria, May 1991.

[Heu95a] Philipp Heuberger. Building correct and reusable software components in oberon using the refinement calculus. Swiss TMR-project 83EU-045389, May 1995.

[Heu95b] Philipp Heuberger. Embedding oberon type extension into refinement calculus. In Bror Bjerner and Nordström [BBN95], pages 145–155.

[Heu96a] Philipp Heuberger. Exercise: Equivalence relation. In *Specification Case Studies with B*, volume Ser. B, No 26 of *Reports on Computer Science and Mathematics.* Åbo Akademi, 1996.

[Heu96b] Philipp Heuberger. Proving on a reasonable level of abstraction with programmer-designed theories. In Henri Habrias, editor, *1st B International Conference: Putting into Practice Methods and Tools for Information System Design*, volume 86, pages 297–302. Institut de Recherches en Informatique de Nantes, November 1996.

[Heu97a] Philipp Heuberger. Application types and the definition of software components. submitted draft, May 1997.

[Heu97b] Philipp Heuberger. A lattice-theoretical subtype definition and the calculation of an abstract common superclass in a global state space. submitted draft, March 1997.

[Heu97c] Philipp Heuberger. The minimal user-interface of a simple refinement tool. In Yves Bertot, editor, *User Interfaces for Theorem Provers*. Sophia Antipolis, France, September 1997. to appear.

[Heu97d] Philipp Heuberger. Notational convenience of the "everywhere" operator in structural induction proofs. submitted draft, March 1997.

[Heu97e] Philipp Heuberger. Oberon and refinement calculus. In Jukka Paakki, editor, *Fifth Symposium on Programming Languages and Software Tools*, C-1997-37, pages 49–60. University of Helsinki, June 1997.

[Heu97f] Philipp Heuberger. Strategies for data-refining an equivalence. submitted draft, March 1997.

[HH95] Philipp Heuberger and David Hofmann. The refinement tool: RefStep. In Bror Bjerner and Nordström [BBN95], pages 565–570.

[Lal94] Spyridon Lalis. *Hermes - Supporting Distributed Programming in a Network of Personal Workstations*. PhD thesis, Swiss Institute of Technology in Zurich, 1994.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 1994.

[Mor88] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10:403–419, 1988.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1994.

[NW89] J. Gutknecht N. Wirth. The Oberon system. *Software Practice and Experience*, 19(9):857–893, september 1989.

[SH89] Beverly A. Sanders and Philipp A. Heuberger. Distributed deadlock detection and resolution with probes. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, volume 392 of *LNCS*, pages 207–218, 1989.

[Szy92] C. A. Szyperski. Write-ing applications: Designing an extensible text-editor as an application framework. In *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems*, 1992.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison Wesley, 1992.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *CACM*, 14:221–227, 1971.

[Wir88] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7), july 1988.