Reuse by Contract

Bernd Fischer
Gregor Snelting

Abt. Softwaretechnologie, Technische Universität Braunschweig
Bültenweg 88, D-38106 Braunschweig, Germany
Tel: +49-531-391-7579
Fax: +49-531-391-8140
Email: {fisch,snelting}@ips.cs.tu-bs.de

Abstract

Reuse by contract is the application of formal methods to software reuse: software components are associated with *contracts*—formal models of their functional behaviour—and administered, retrieved, and reused by these. We argue that reuse by contract is necessary for safe reuse in a formal process model, but is helpful even for more traditional software development. We discuss some obstacles against the use of formal component specifications, and propose some solutions in order to make reuse by contract practical.

Keywords: formal methods, reuse, software component retrieval, specification matching, automated deduction.

Workshop Goals: Discuss the design of reusable component libraries; discuss ways to mitigate acceptance problems; discuss tool architectures; study relation with OO design methods; study combination with structural code reuse mechanism (e.g., patterns or functors.)

1 Background

Reuse by contract is the application of formal methods to software reuse: software components are associated with *contracts*—formal models of their functional behaviour—and administered, retrieved, and reused by these.

Similar approaches have been proposed before (e.g., [KRT87, RW91, MM91]) but without convincing success. The goal of our project NORA/HAMMR[1][FKS95c, FKS95b, FKS95a] is thus to make reuse by contract *practical*. We investigate

- scalable and efficient architectures for reuse by contract,
- reuse-friendly specification techniques,
- library organization techniques based on contracts,
- deductive component retrieval,
- its interaction with formal software development processes, and
- the integration of reuse by contract and conventional reuse mechanisms.

Our long-term goal is to build a system which smoothly integrates component design, implementation, and verification with the systematic reuse of a fully specified and verified component library.

Our work on NORA/HAMMR started in 1994. Our main topic so far has been the application of automated deduction techniques to solve the proof tasks emerging from deductive component retrieval. A large number of experiments done in collaboration with colleagues from the German joint research project on deduction show that current theorem provers are capable to solve enough of the emerging tasks. Our implementation is tailored towards these experiments. It uses VDM as contract language but can generate proof tasks for different theorem provers. We currently work on an improved library organziation and the integration of a program verification system.

2 Position

2.1 What is Reuse by Contract really?

B. Meyer has coined the phrase *design by contract* [Mey92] to denote a software development style which emphasises the importance of formal specifications and interleaves them with actual code. Reuse by contract is an attempt to lift this style to code reuse. Its basic idea is to turn the contracts into the actual medium by which client and provider can negotiate reuse:

- The client states his side of the contract as a pair of granted pre- and required postcondition.

[1]NORA is no real acronym; HAMMR is the highly-adaptive multi-method retrieval tool. This work has been sponsored by the DFG under grant Sn11/2-3.

- The provider formalizes his bid (i.e., each library component) the other way round, as a pair of required pre- and granted postcondition.
- Negotiation is defined in terms of the contract: an offered component is suited for reuse if the involved pre- and postconditions satisfy a well-defined logical relation.

The negotiation process, also called deductive component retrieval, is the most important technical problem to be solved and as such sometimes identified with reuse by contract. In our opinion, however, only its integration into a formal software development process will lead to significant reuse effects and thus justify the name "reuse by contract." In a formal setting, the contracts arise naturally (e.g., from refinements) and do not impose any extra work on the developers. Consequently, reuse can be built into program design from the very beginning.

On the other side, design by contract is *not per se* reuse by contract as the existence of a library does not automatically imply its re-use. The difference is in the roles the contracts play. In the design approach, contracts are passive and confined to the library. They do not only describe the components properties but are also their possessions. The reuse approach removes this asymmetry and "activates" the contracts for prospective clients. It is a way to exploit the full power of contracts.

2.2 Proof Tasks and Code Reuse

Components can be reused if they bridge the gap between the clients stated pre- and postcondition. Proof tasks formalize this relation; their exact nature determines form and effects of reuse.

The most efficient form of reuse takes place if the gap between the client's offer (the precondition) and his wishes (the postcondition) is bridged completely. A component c can be plugged in and thus close the gap if it has a weaker pre- and a stronger postcondition than the client requires in his query q. This ideal situation is usually formalized in the following condition, or *proof task*:[2]

$$(pre_q \Rightarrow pre_c) \wedge (post_c \Rightarrow post_q) \tag{1}$$

This proof task is however not adequate if q is a partial function but not c. If we want c to match q even if its results on the extended domain do not fit the original query, we must restrict the implication between the postconditions on the domain given by pre_q. We thus work with proof tasks of the form

$$(pre_q \Rightarrow pre_c) \wedge (pre_q \wedge post_c \Rightarrow post_q) \tag{2}$$

This so-called *plug-in compatibility* supports safe reuse. The retrieved components may be considered as black boxes and may be reused "as is", without further proviso or modification.

Another, weaker form of the proof tasks emphasizes the clients postcondition and retrieves all components which satisfy it at least on their own domain:

$$pre_c \wedge post_c \Rightarrow post_q \tag{3}$$

[2]Actually, the proof tasks are universally closed wrt. the formal input and output parameters of the component and the query and also contain equations relating the parameters. Likewise, the pre- and postconditions are of course logical functions of the respective parameters. However, to improve the legibility, we use this traditionally abbreviated formulations.

Generally, code reuse based on this *conditional compatibility* is potentially unsafe because the client has to satisfy the open obligation pre_c. In a formal framework, however, (3) is justified because it describes a normal refinement step: the client trades his own open contract $post_q$ against the usually simpler pre_c by re-using c. Note that (3) is also more efficient to check, as only one implication has to be proven.

To increase the recall, conditional compatibility can be relaxed further by again taking the client's precondition into account. But in contrast to (2) it is now added to the premise:

$$pre_c \wedge pre_q \wedge post_c \Rightarrow post_q \tag{4}$$

Hence, *partial compatibility* retrieves all components which do "the right thing" at least on a domain restricted by $pre_c \wedge pre_q$. By varying pre_q, clients can control recall and granularity of reuse. The stronger it is, the more components are retrieved but the smalller is their respective benefit, simply because pre_q acts as an additional open obligation.

2.3 Significance

The investigation [Lio96] of the recent Ariane 5 disaster revealed that it was caused by the reuse of an unmodified Ariane 4 software component which led to an uncaught exception crashing the software and hence the spacecraft. In [JM97], however, Jézéquel and Meyer argue that the ultimate reason for the crash was the components failure to state its assumptions, i.e., the absence of a contract. They conclude

> "There is a more simple lesson to be learned from this unfortunate event: *Reuse without a contract is a sheer folly.* From CORBA to C++ to VisualBasic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naïve hopes are doomed to produce results *far worse* than a traditional, reuse-less software process. To attempt to reuse software without Eiffel-like assertions is to invite failures of potentially disastrous consequences."

We share this conclusion as motivation for our work.

2.4 Benefits

For the users, the biggest visible benefit is of course the ability to retrieve components which *provably* match their needs. Provably matching components increase the overall quality of the software. They also improve the software process, the productivity and other aspects of software development. Depending on the kind of compatibility used (plug-in, conditional or partial), several benefits can be identified.

Plug-in compatibility – and, to a smaller extent, conditional compatibility – are most useful in a formal development process. Here, it can be used for the *safe composition* of components. A component which satisfies a proof task is guaranteed not to compromise the overall correctness. This is true even for conditional compatibility, if – as is often the case – the refinement process can generate the component's precondition. Safe composition prevents reuse disasters like the Ariane case. Safe composition is a must for any safety-critical software project which wants to utilize component reuse, and reuse by contract is the only available technology.

In some cases software composition from formally specified components can be done *automatically.* By means of constructive type theory, a formal specification can be transformed into executable code which may also contain calls to (formally specified) library components. Needless to say, the resulting code is provably correct.

Once a new piece of software has been constructed by safe composition, automatically or not, it can be added to the library. Hence the basis for reuse is increased, without any additional overhead: in a formal process, the specification of a new component or subsystem must be supplied anyway. Furthermore, reuse by contract offers additional support for a formal development process. Development steps are larger, and an actual implementation for subsystems can be obtained early, thereby supporting vertical prototyping. If new components are added to the library, a positive feedback is obtained which encourages the use of formal specifications and dedicated technology and tools (e.g. verifiers) for a formal development process.

In a less formal software process, reuse by contract still improves software development. In such a context, partial compatibility will be most useful. Partial compatibility has highest recall, but may require manual checks or even component modifications in order to match the components precondition. For a software developer whose primary interest is to find reusable components, this is not an obstacle. If the primary goal is to reuse code, the developer will be ready to provide the stronger precondition needed for conditional compatibility, or even extend the component's functionality in order to utilize partial compatibility. Still, use of a formal postcondition considerably increases the precision of component search, that is, reduces the probability of finding irrelevant components.

Reuse by contract is attractive even for those who prefer commercial success over safety or improved component retrieval. Today's libraries (for example, the Standard Template Library for C++) are not formally specified. If different vendors offer the same library, there will be subtle differences between implementations. As a consequence, a user will not be able to switch from one library implementation to another – enabling library monopolies and preventing innovative vendors with small market share from commercial success. Hence market transparency requires that there is an implementation-independent specification – a *standard* – for a component library. Today, such standards have been established in other software and hardware areas; usage of formal specifications as standards for libraries will result in better return on investment for independent library vendors.

2.5 Obstacles

There are a couple of problems which make a successful implementation of reuse by contract a hard task. The major impediment is the general acceptance problem of formal methods. As one colleague put it: "If I need a sort routine, I say `grep sort`!"

Without a formal software development process, the up-front costs become fairly high. Programmers are not used to contracting and may consider it merely as an additional burden which remains without any benefit as long as there are no or not enough specified libraries to be reused. Library construction, however, is time-consuming and expensive, especially when it is not supported by the feed-back described above. Worst of all, due to the general indifference in formal methods, the market offers only very few specified libraries to begin this process with.

The "look and feel" of a reuse system also can impair its usefulness. If the end user has to deal with complicated parameter settings for the prover, specify details he considers irrelevant, or provide postconditions in some cryptic prover language, reuse by contract will not be successful.

Another source of problems is the computational complexity of deductive component retrieval. Long response times due to insufficient deductive power can easily render the entire concept impractical; this in particular affects scale-up for big libraries or complicated components. Contracts for larger components which in turn promise larger pay-offs may become too large or too complicated. A more technical aspect may lead to even more complicated proof tasks. If provider and client use different mathematical concepts (e.g., sets and lists) the resulting "view mismatch" can only be solved if the prover deduces the necessary mappings.

In a non-formal software process, the use of formal specifications can even hamper the retrieval abilities. The reason is that recall may suffer from the overwhelming precision of formal specifications. If there is a component which differs only slightly from what the user wants, it will not be found, because the proof obligation can not be fulfilled ("near miss"). Theorem provers do not have a notion of an "almost provable" statement, and it is this sharp distinction between true and false statements which may backfire in a reuse context. Note that in a formal process, this problem does not occur as partial or "fuzzy" contract fulfillment is not acceptable.

2.6 Possible Solutions

There is no medicine for people who reject formal methods. The only argument which might appeal to them is that reuse by contract is not going to replace existing, established retrieval methods, but to *augment* them. For all the other problems, solutions can be outlined as follows.

First of all, a retrieval system based on formal specifications must hide the deductive machinery completely. Any details for setting prover parameters, synchronizing parallel-running provers, generating prover input etc. must be invisible to the user. Instead, the retrieval tool should offer an interface which utilizes the end user's language and concepts; in particular, it should also offer access to more traditional retrieval algorithms. As somebody *has* to take care of preprocessing and tuning the formal specifications and tune the deductive engine, we propose that this is done by an expert. The end user must not be bothered by this.

In order to tackle the performance and scale-up problems, we utilize two mechanisms: *abstraction* and *incrementality*. Abstraction means that not always a traditional formal specification will be needed, somtimes a more compact component description is sufficient. Multiple layers of specifications can be used to separate the core functionality from non-functional implementation aspects as e. g. structure sharing [PH95]. This can be achieved by a domain-specific logic which would not only improve the deductive abilites of the system but would also be beneficial for the end user.[3]

Incrementality means that several processes must cooperate in order to achieve an increasing reduction of the problem space. NORA/HAMMR uses a *filter chain* in order to reduce the burden of the theorem prover. The chain consists of a series of filters of increasing power, the prover is only the last element in the chain. Chain configuration may vary; a typical filter chain includes signature matching and model checking. Signature matching selects components according to a

[3]It is technically easy to create a domain-specific extension of a specification language: this only requires that a set of predefined function and predicate names is defined, whose meaning is given by some additional axioms.

specification of their interface alone. Model checking is used in order to discover non-theorems: if a counterexample in some small model can be found, the proof obligation is considered a non-theorem and the component rejected. Both techniques may negatively affect recall as well as precision, as a counterexample in a finite model of, say, the integers may be invalid, and demanding identical interfaces is too restrictive.[4] But they greatly reduce the burden of the prover, as only a small number of proof obligations survives the preliminary filters.

Signature matching can also help with the view mismatch problem. It identifies the structural similarities between the types in question which in turn can be used to construct some of the abstraction functions automatically.

In order to reduce the risc of not finding components due to overspecification or "near misses" of the prover, we again propose abstraction and incrementality. The user interface must allow to incrementally sharpen the postcondition (or weaken the precondition), thereby incrementally filtering the set of surviving components. Furthermore, use of a domain-specific specification language offers the appropriate abstractions to the user; near misses due to erroneous low-level specification details are avoided.

3 Comparison

Most work on library design (e.g., [MS96, Knu93]) follows the traditional style of informal or stylized descriptions and reference implementations. The industrial-strength example of a contract-based library design we know of is Meyer's work [Mey94]. There is, however, some more research work, e. g. the RESOLVE project [SW94]. Both [JC93] and [MMM94] use formal specifications to determine a subsumption relation between components and structure their libraries accordingly. Similarly, [LW94] define the notion of *behavioral subtype* as an extended means to organize class libraries and [Lea91] has developed techniques to support the specification and verification of object-oriented programs.

Deductive component retrieval has also been investigated by [RW91] which used λProlog to specify the components and its built-in higher-order unification as retrieval mechanism. Moorman Zaremski and Wing [MW95] were the first to explore different match relations; our own work (cf. section 2.2) expands on their results. They also introduced the use of a "real" specification language (Larch/ML) for component description. With the exception of [Ste91] which works with algebraic specifications, most other approaches now also use languages which are some sugared variant of first order logic. However, while [MW95] applies the associated interactive LarchProver to solve the proof tasks, their sheer number requires an automated theorem prover as e.g., Otter [MMM94] or SETHEO [FS97] to make it practical.

A more pragmatic approach to deductive retrieval is to use the components types as their specifications (e.g., [Rit91].) This *signature matching* allows the application of more efficient reasoning mechanisms (e.g., order-sorted theory unification) but also a very concise query formulation. It is thus a successful tool for functional languages which offer rich type structures. The behavioral abstraction which is inherent to types makes an unmodified type-based retrieval unsuited for reuse by contract but it can still be used as a fast pre-filter.

[4] In fact, the latter problem can be tackled by use of additional axioms which allow e.g. interchanges of parameters, and for the former we propose the use of abstract model checking.

There has also been some earlier work to integrate deductive retrieval into software development environments. The PARIS system [KRT87] supported the semi-automatic construction of programs over a library of so-called schemes, i.e., program fragments which are enriched by assertions about their combination and instantiation possibilites. The construction process then generated proof tasks which were solved by the Boyer-Moore theorem prover. The Inscape system [Per87] aimed at the development of large software systems based on specifications; it also offered some retrieval support. However, both systems worked with severely restricted logics and inference mechanisms and never left the prototype stage.

Lowry et al. [LP$^+$94] utilized a formally specified library which contains functions for celestial mechanics and spaceship course computations. After providing a formal specification of e.g. a space vehicle's destination point and time, the system automatically composes a program consisting of calls to appropriate library routines, which compute the flight data. Lowry's system uses a domain-specific logic, which in turn is hidden from the user by a sophisticated graphical user interface.

4 Research Topics

Although specified component libraries are a necessary requirement for *any* code reuse mechanism working with formal methods and not only for reuse by contract, they are a nearly extinct species. We think that the community should take up the lead of Meyer and work towards *realistic*, formally specified libraries which must also cover non-functional aspects of components which traditionally matter for the users, e. g. structure sharing. This work should also include the development of appropriate domain-specific logics.

Larger benefits are expected from reusing components of a much coarser granularity than simple functions ("megaprogramming", [WWC92].) Scaling-up specification methods to such megacomponents requires a lot of further research. First of all, flexible components are parameterized (generic packages, C++ templates etc). Thus even signature matching requires higher-order unification. A next step could be an investigation about specifications of design patterns or parameterized modules/functors. Having a fully specified pattern library would be a nice argument.

Scale-up also concerns deductive retrieval. First of all, the provers must be adapted to reflect a situation where there are thousands of simple proof obligations, and almost all of them are non-theorems. Furthermore, methods have to be developed for fast rejection of non-theorems which do not compromise recall. One promising candidate is abstract model checking [Jac94]. Probably the combination of behavioural subtyping and signature matching also works to this end.

Finally, to increase the number of reuse opportunites, the strict compatibilities defined in 2.2 could by relaxed by introducing some approximate reasoning. However, this requires appropriate automatic component adaption mechanisms to maintain the integrity of reuse by contract.

References

[FKS95a] B. Fischer, M. Kievernagel, and G. Snelting. "Deduction-Based Software Component Retrieval". In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, (eds.), *Working*

Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, pp. 1–5, Montréal, August 1995.

[FKS95b] B. Fischer, M. Kievernagel, and W. Struckmann. "High-precision retrieval for high-quality software". In I. M. Marshall, W. B. Samson, and D. G. Edgar-Nevill, (eds.), *Proc. 4th Software Quality Conf.* , pp. 80–88, Dundee, July 1995. University of Abertay Dundee.

[FKS95c] B. Fischer, M. Kievernagel, and W. Struckmann. "VCR: A VDM-based Software Component Retrieval Tool". In M. Wirsing, (ed.), *Working Notes of the ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pp. 30–38, Seattle, Wash., April 1995.

[FS97] B. Fischer and J. M. P. Schumann. "SETHEO Goes Software Engineering: Application of ATP to Software Reuse". In W. McCune, (ed.), *Proc. 14th CADE*, *LNAI* **1249**, Townsville, July 1997. Springer.

[Jac94] D. Jackson. "Abstract Model Checking of Infinite Specifications". In M. Naftalin, T. Denvir, and M. Bertran, (eds.), *Proc. 2nd FME*, *LNCS* **873**, pp. 519–531, Barcelona, October 1994. Springer.

[JC93] J. Jeng and B. H. C. Cheng. "Using formal methods to construct a software component library". In I. Sommerville and M. Paul, (eds.), *Proc. 4th ESEC*, *LNCS* **717**, pp. 397–417, Garmisch-Partenkirchen, September 1993. Springer.

[JM97] J.-M. Jézéquel and B. Meyer. "Design by Contract: The Lessons of Ariane". *IEEE Computer*, **30**(1):129–130, January 1997.

[Knu93] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, New York, 1993.

[KRT87] S. Katz, C. A. Richter, and K. S. The. "PARIS: A System for Reusing Partially Interpreted Schemas". In *Proc. 9th ICSE*, pp. 377–385, Montery, CA, March 1987. IEEE Computer Society Press.

[Lea91] G. T. Leavens. "Modular Specification and Verification of Object-Oriented Programs". *IEEE Software*, **8**(4):72–80, July 1991.

[Lio96] J. L. Lions et. al. Ariane 5 Flight 501 Failure Report, 1996.

[LP+94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. "AMPHION: automatic programming for scientific subroutine libraries". In Z. W. Raś and M. Zemankova, (eds.), *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, *LNAI* **869**, pp. 326–335. Springer, October 1994.

[LW94] B. Liskov and J. M. Wing. "A Behavioral Notion of Subtyping". *ACM TOPLAS*, **16**(6):1811–1841, November 1994.

[Mey92] B. Meyer. "Applying "Design by Contract"". *IEEE Computer*, **25**(10):40–51, October 1992.

[Mey94] B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice-Hall, Englewood Cliffs, 1994.

[MM91] P. Manhart and S. Meggendorfer. "A knowledge and deduction based software retrieval tool". In *Proc. 4th Intl. Symp. on Artificial Intelligence*, pp. 29–36, 1991.

[MMM94] A. Mili, R. Mili, and R. Mittermeir. "Storing and Retrieving Software Components: A Refinement-Based System". In B. Fadini, (ed.), *Proc. 16th ICSE*, pp. 91–102, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[MW95] A. Moorman Zaremski and J. M. Wing. "Specification Matching of Software Components". In G. E. Kaiser, (ed.), *Proc. 3rd FSE*, pp. 6–17, Washington, DC, October 1995. ACM Press.

[Per87] D. E. Perry. "The Inscape Environment". In *Proc. 11th ICSE*, pp. 2–12. IEEE Computer Society Press, May 1987.

[PH95] A. Poetzsch-Heffter. "Interface Specification for Program Modules Supporting Selective Updates and Sharing and their Use in Correctness Proofs". *Softwaretechnik-Trends*, **15**(3):116–125, October 1995. Proc. Softwaretechnik 95, G. Snelting (ed.).

[Rit91] M. Rittri. "Using types as search keys in function libraries". *JFP*, **1**(1):71–89, January 1991.

[RW91] E. J. Rollins and J. M. Wing. "Specifications as Search Keys for Software Libraries". In K. Furukawa, (ed.), *Proc. 8th Intl. Conf. Symp. Logic Programming*, pp. 173–187, Paris, June 24-28 1991. MIT Press.

[Ste91] R. A. Steigerwald. *Reusable Software Component Retrieval via Normalized Algebraic Specifications.* PhD thesis, Naval Postgraduate School, December 1991.

[SW94] M. Sitaraman and B. W. Weide. "Special Feature: Component-Based Software Using RESOLVE". *ACM SIGSOFT Software Engineering Notes*, **19**(4):21–22, October 1994.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. "Toward megaprogramming". *Communications of the ACM*, **35**(11):89–99, November 1992.

Biographies

Bernd Fischer is researcher at the Department of Software Technology at the Technical University of Braunschweig. His interests include formal specification, automated deduction and functional programming. He received his diploma in computer science from the TU Braunschweig in 1990.

Gregor Snelting is a professor for software technology at the Technical University of Braunschweig. His main interest is to utilize deductive and algebraic techniques in order to improve software design, configuration management, component reuse, software reengineering, and software validation. He received a diploma in computer science and mathematics (1982) and a PhD in computer science (1986) from the TU Darmstadt, and became professor and leader of the software technology group in 1991.