

# Call-Ordering Constraints

Nam Tran, David Abramson, Christine Mingins  
*Faculty of Information Technology, Monash University*  
*{nam.tran, david.abramson, christine.mingins}@infotech.monash.edu.au*

## Abstract

*Several kinds of call-ordering problems have been identified, all of which present subtle difficulties in ensuring the correctness of a sequential program. They include object protocols, synchronisation patterns and re-entrance restrictions. This paper presents call-ordering constraints as a unifying solution to these problems. These constraints are new classes of contracts in addition to traditional preconditions, postconditions and invariants. They extend the traditional notion of behavioural subtyping. The paper shows how constraint inheritance can almost ensure behavioural subtyping conformance. The paper also shows how these constraints may be monitored at run time. Call-ordering constraints are included in the BECON contract system, which has been implemented on the Common Language Infrastructure (CLI).*

## 1. Introduction

The ordering of method calls in a sequential program can present subtle difficulties in ensuring its correctness. This problem manifests in a number of variants. Existing research [5, 6, 9, 15-18, 20, 23] has studied these variants using such designations as synchronisation, mandatory calls, re-entrance and various versions of “protocols”. Prior work only addressed these cases individually and they remain isolated solutions.

We have earlier shown the significance of the problem and classified the problem cases in a unifying taxonomy [21]. The taxonomy identifies three main classes of call-ordering problems, namely protocols, synchronisation patterns, and re-entrance restrictions. They affect the most important classes of programs, namely sequential, concurrent, object-oriented and component-based programs. Therefore, it is clearly beneficial to develop a unifying solution to all the problem cases. In this paper, we deal with sequential object-oriented and component-based programs.

Contract systems [2, 5, 14, 20] are a common approach to software specification and verification. Typical contract systems support the specification of method preconditions, postconditions and class invariants. They usually support the dynamic monitoring of code’s conformance to contracts. However, a method’s precondition and postcondition cannot express, let alone ensure, ordering constraints between it and other methods. Checking a class invariant can help prevent some dangerous re-entrant calls. However, it is not always practical to evaluate, or even to write, invariants of the required complexity.

We have developed a unifying solution to call-ordering problems as part of BECON, a language neutral contract system for binary components [22]. It presents an integrated model for contract representation and verification. Contracts are represented as first-class entities in binary components. An execution environment implements contract verification. Call-ordering constraints are new classes of contracts. BECON unifies and integrates them with traditional contracts. Static verification can establish if a type’s call-ordering constraints conform to another’s. Dynamic monitoring can detect if a program violates specified constraints.

The remainder of the paper is structured as follows. First, we describe the classes of call-ordering constraints in section 2. We then show in section 3 how they extend the traditional notion of behavioural subtyping. In section 4, we describe dynamic monitoring algorithms. In section 5, we briefly describe how the BECON contract system and its implementation on the CLI [10] support call-ordering constraints. We discuss related work in section 6, and conclude in section 7.

## 2. Classes of call-ordering constraints

Call-ordering constraints include protocols, mandatory calls and forbidden calls. An interface may specify a protocol and a method may specify a list of mandatory calls and a list of forbidden calls. (In this paper, we use the term “interface” to denote an interface to a component or an object.) The new concept of forbidden calls helps solve the re-entrance problem.

## 2.1. Protocols

A protocol specifies the non-uniform service availability of an interface. In other words, a protocol dictates when a call to a method of the interface is allowed. In contracting terms, the effect is similar to that of a precondition. As such, a protocol is an obligation on the clients of the interface, and a client must obey the protocol when issuing calls to the interface.

A protocol is a regular process that can be modelled by a non-deterministic finite state automaton (NFSA). An NFSA is usually described by a non-deterministic finite state labelled transition system (LTS). A protocol may include state predicates to resolve some of the internal non-determinism introduced by the LTS model. This concept of protocol is similar to protocols in regular types [16], in Java [6], and in Fugue [9]. NFSA can approximate actual process behaviours more precisely than regular languages, which can only describe process traces, not process failures [16].

Figure 1 depicts the protocol of a Var interface.

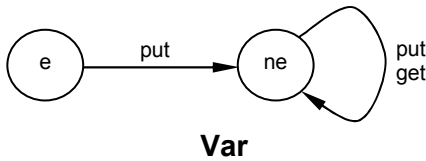


**Figure 1. Protocol of Var interface**

For each interface that specifies a protocol, an implementing component includes its protocol description. This description is an LTS, modelling that interface’s protocol, with methods of the interface as transition labels and with explicitly declared abstract protocol states. The transition labels may include methods that the interface inherits from others, as well as locally declared methods. The description consists of a list of protocol states, and a list of transitions.

Each protocol state has an identity, and optionally a predicate. A state predicate is a method that takes no parameters other than the target object (“this”) and returns a Boolean value. A state may also be marked as ‘start’ or ‘final’. An object starts its life in a ‘start’ state, and ends its life in an ‘end’ state.

Each transition has a source state, a destination state, and a transition method. A method may participate in multiple transitions. There may also be multiple transitions from one state to another. When a source state and transition method pair does not identify a unique transition, the protocol is non-deterministic. Some non-determinism may be resolved at run time by state predicates, whose evaluation may indicate that a resultant state is actually invalid. However, in general, there may be external non-determinism that is part of the actual behaviour and cannot be eliminated.

Methods not mentioned in a protocol description represent implicit, reflexive transitions. These methods may be called in any protocol state, and do not change the state. This characteristic is important for protocol conformance, as explained below, and for keeping protocol descriptions compact.

The following pseudo-code illustrates the description of the protocol of Var, with state predicates, and ‘start’, ‘final’ markers. This protocol is deterministic.

```

protocol Var {
  states {
    empty {start/end, isEmpty()}
    not-empty {end, !isEmpty()}
  }
  transitions {
    <empty> put <not-empty>
    <not-empty> put <not-empty>
    <not-empty> get <not-empty>
  }
}
  
```

**LTS description of Var**

## 2.2. Mandatory calls

Mandatory calls specify synchronisation patterns. A method’s list of mandatory calls specifies the methods that its implementation must invoke. The method can make these calls directly or indirectly. They can be calls to the specified methods, or methods that override them. In contracting terms, it is an obligation on the method implementation for the benefit of its clients. It may be seen as a postcondition that the implementation has invoked those specified methods. A mandatory call specifies only a method that must be invoked, without a specific target object or specific arguments. A list of mandatory calls may have a strict sequential order, or the order among those calls may be insignificant.

A list of mandatory calls may specify arbitrary methods of the same interface, of other interfaces in the same component, and of other components. If some components are not loaded at run time, the mandatory calls targeting their methods do not resolve, therefore, are obviously not satisfied.

Mandatory calls fully cover typical synchronisation patterns. For example, for the Observer pattern [12], the method Subject.change() can specify the method Observer.update() as a mandatory call. For the Model-View-Controller pattern, alternatively, a method of Controller can specify Model.change() and View.update() as mandatory calls, strictly in that order. The method must also execute these calls as an atomic action.

## 2.3. Forbidden calls

Forbidden calls are opposite to mandatory calls and they are a new innovation to help solve the re-entrance

problem. A method’s list of forbidden calls specifies the methods that its implementation must not invoke at all, even indirectly. A method implementation also must not invoke methods that override specified forbidden ones. In contracting terms, similar to mandatory calls, it is an obligation on the method implementation for the benefit of its clients. It may be seen as a post-condition that the implementation has not invoked those specified methods. A forbidden call specifies only a method that must not be invoked, without including a specific target object or specific arguments. The order among forbidden calls in a list is not important.

Similar to mandatory calls, a list of forbidden calls may specify arbitrary methods of the same interface, of other interfaces in the same component, and of other components. If some components are not loaded at run time, the forbidden calls targeting their methods do not resolve, therefore, obviously cannot be violated.

Forbidden calls can help solve the typical re-entrance problem because a method may specify forbidden calls to prevent re-entrance. These re-entrance restrictions, by definition, involve only methods of the same object. This technique can be used to supplement, or replace, invariant checking, especially where using invariant is difficult or too costly. Writing an executable invariant may be difficult due to the limitation in expressive power. Evaluating a complex invariant may incur an unacceptably high overhead.

Forbidden calls targeting methods of other objects are also useful for specifying and enforcing requirements of non-interference between different parts of a program. They can also be used for security constraints such as code access control. However, this usage for security applications requires further research, and is beyond the scope of this paper.

### 3. Behavioural subtyping

Behavioural conformance relations between call-ordering constraints extend the traditional notion of behavioural subtyping. Constraint inheritance can also largely guarantee these conformance relations.

#### 3.1. Traditional behavioural subtyping

The notion of behavioural subtyping [1, 13, 14] requires that a subtype’s contract specifies behaviours conforming to the behaviours specified by a supertype’s contract. This is to ensure that an instance of a subtype can substitute for an instance of a supertype without observable changes in program behaviours. This conformance includes structural subtyping constraints such as co-variance, contravariance, and in-

variance rules for parameters and return types [7]. However, we are only concerned with behavioural contracts here. Following traditional behavioural subtyping, a subtype’s invariant implies a supertype’s invariant. A method’s precondition is implied by the precondition of a method it overrides. A method’s postcondition implies the postcondition of a method it overrides.

#### 3.2. Extended behavioural subtyping

The extended notion of behavioural subtyping requires also that a subtype’s call-ordering constraints conform to a supertype’s constraints. First, a subtype’s protocol conforms to a supertype’s protocol. Second, a method’s list of mandatory calls includes the list of mandatory calls of a method it overrides. If ordering is significant, it maintains the order between items of the overridden method’s list. Similarly, a method’s set of forbidden calls includes the set of forbidden calls of a method it overrides.

These conformance relations are devised according to the same substitutability principle.

**3.2.1. Protocol conformance.** The protocol conformance relation ensures the observable equivalence of method availability. As discussed earlier, a protocol is an obligation on the clients of an interface, similar in spirit to method preconditions. A client that obeys a type’s protocol expects its calls to the type’s methods to succeed. Therefore, for substitutability, a subtype’s protocol must satisfy any client who obeys the supertype’s protocol.

This conformance relation also allows a subtype’s protocol to be less demanding than its supertype’s protocol. Newer clients of the subtype can take advantage of the new protocol. This is similar to weakening a precondition. For example, compared to the Var interface above, a DefVar interface that supports default initialisation allow a ‘get’ in an ‘empty’ state. This DefVar’s protocol is less demanding than, but conforms to, Var’s protocol. Newer clients of DefVar need not make an initial call to ‘put’ before calling ‘get’. However, Var’s existing clients, which always make that initial call, continue to work with DefVar.

This protocol conformance relation is the same as request substitutability of regular types [6, 16]. This conformance relation requires the traces of a subtype’s protocol (specified as an LTS) to include the traces of a supertype’s protocol. In addition, the failures of the subtype’s LTS, relatively to the supertype’s LTS, must be included in the failures of the supertype’s LTS. The second requirement is only significant for protocols that are non-deterministic. Intuitively, these two requirements ensure that a client obeying the supertype’s protocol can always issue calls to the subtype. In addi-

tion, any call deemed invalid by the subtype's protocol could also possibly be deemed invalid by the supertype's protocol. Therefore, a client is never "surprised" by the substitution of an instance of a subtype for an instance of a supertype.

This protocol conformance relation also means that the set of implicit, reflexive transitions, for unmentioned methods, of a subtype must include that of the supertype. This is really a special case of trace inclusion but has great significance in practice. A subtype, introducing a protocol for the first time in a type hierarchy, must not place any constraint on the inherited methods, by not mentioning them in the protocol description. A subtype thereby can introduce a protocol useful for its specialised behaviour without breaking clients relying on the unrestrained behaviours of super-types. This is especially useful for type hierarchies where all types are subtypes of a default root type, e.g. Object in .NET, Java or ANY in Eiffel.

**3.2.2. Mandatory call conformance.** A method's list of mandatory calls must include the list of mandatory calls of a method it overrides. In addition, if the order between mandatory calls of the overridden method is significant, it must be preserved in the overriding method. This inclusion ensures that a client of the overridden method is always satisfied regarding its expectation for mandatory calls. The overriding method fulfils at least as much as the overridden method, and possibly more. In other words, the subtype's method preserves and possibly strengthens mandatory call requirement of an overridden method. The extra mandatory calls that an overriding method specifies, if any, are of no concerns to the existing clients. If the extra mandatory calls clash with the previously declared forbidden calls, the conflict must be resolved at the time of declaration, as explained below.

**3.2.3. Forbidden call conformance.** Similarly, a method's list of forbidden calls must include the list of forbidden calls of a method it overrides. This inclusion ensures that a client of the overridden method is always satisfied regarding its expectation for forbidden calls. The overriding method fulfils at least as much as the overridden method, and possibly more. In other words, the subtype's method preserves and possibly strengthens forbidden call requirement of an overridden method. The extra forbidden calls that an overriding method specifies, if any, are of no concern to the existing clients. If the extra forbidden calls clash with the previously declared mandatory calls, the conflict must also be resolved at the time of declaration.

### 3.3. Forcing behavioural subtyping through constraint inheritance

It is useful to guarantee behavioural subtyping by disciplined inheritance of constraints, as has been done for DbC assertions [8, 14]. When behavioural subtyping is guaranteed, verification of constraint conformance is unnecessary. Such verification may be computationally expensive or even undecidable. In addition, with such guarantee, all "hierarchy errors" [11] are completely avoided. Moreover, only the "extra" contracts need to be written for a subtype.

**3.3.1. Mandatory call inheritance.** A method inherits all mandatory calls declared by the methods it overrides when the order between mandatory calls in a list is not important. It declares only the extra mandatory calls, if any, which are appended to the lists of mandatory calls it inherits. Clearly, this inheritance ensures that an overriding method's mandatory calls include an overridden method's mandatory calls, as required by the conformance relation explained above.

When the order between mandatory calls is important, all mandatory calls from overridden methods must be copied to an overriding method's list of mandatory calls. Extra mandatory calls may be inserted into the list at the desired position while preserving the order of the copied mandatory calls. Given two lists of mandatory calls, it is easy to verify if one includes and preserves the order of the other.

Detecting if the extra mandatory calls of an overriding method conflict with its inherited forbidden calls is straightforward. A method cannot be both a mandatory call and a forbidden call of another method. A compiler can perform this detection and signal a compile time error. This detection can also be performed after compile time if binary components contain mandatory call specifications (as in BECON). Such a conflict is a semantic error that must be resolved by a developer.

**3.3.2. Forbidden call inheritance.** A method inherits all forbidden calls declared by methods it overrides. (The order between methods specified in a list of forbidden calls is not significant.) It declares only the extra forbidden calls, if any, that are appended to the lists of forbidden calls it inherits. Clearly, this inheritance ensures that an overriding method's forbidden calls include an overridden method's forbidden calls, as required by their conformance relation explained above.

Conflicts between newly declared forbidden calls and inherited mandatory calls are treated in the same manner as conflicts between inherited forbidden calls and newly declared mandatory calls.

**3.3.3. Limited protocol inheritance.** Protocols may be inherited unchanged. Obviously, a subtype’s protocol conforms to its supertype’s protocol in that case. A type that does not specify a protocol inherits a supertype’s protocol. A root type that does not specify a protocol has an implicit one that places no constraints over its method availability.

If changes to a protocol are required, it cannot be inherited but must be rewritten instead. A type that specifies a protocol must obey the protocol conformance relation explained above. Conformance between protocols can be verified by a relatively simple algorithm, proposed by Nierstrasz [16].

### 3.4. Method identities and overriding

We also extend behavioural subtyping by supporting overriding in call-ordering constraints. Matching method identities specified as protocol transitions, mandatory calls, or forbidden calls, takes into account overriding. A specified method matches an actual call to itself or to any method overriding it.

Because a method and the methods that override it are deemed behaviourally equivalent, this feature prevents contract violation due to overriding and is very convenient. If calling a method ‘m’ violates a protocol, or a forbidden call, calling any method that overrides ‘m’ is also a violation. Similarly, if calling ‘m’ satisfies a mandatory call, calling any method that overrides ‘m’ also satisfies that mandatory call. This extended coverage ensures that the contracts involving ‘m’ as a protocol transition, a mandatory call, or a forbidden call are “future-proof”. In addition, a constraint only needs to specify ‘m’, and not a set of methods, to cover ‘m’ and all the known methods that override ‘m’, but not any method that ‘m’ overrides.

### 3.5. Multiple inheritance

We support constraint inheritance (except with protocols) from multiple supertypes and static detection of conflicts, which must be resolved by developers.

The detection of conflicts between mandatory and forbidden calls from multiple supertypes is similar to that for single inheritance of mandatory and forbidden calls as explained above. Similarly, source compilers can do this detection, and in BECON, detection may be performed on binary components as well.

For protocols, if there are multiple protocols from multiple supertypes, which are not in subtyping relations among themselves, these protocols are not inherited. Instead, a subtype must specify a protocol that conforms to the protocol of each supertype.

The requirement to resolve all such conflicts does not present undue restrictions. Intuitively, multiple supertypes that are not in subtyping relations among themselves are orthogonal. Therefore their protocols should also be orthogonal. Multiple overridden methods that do not override one another should be semantically similar. Therefore, their constraints should compose without conflicts. If these methods are conceptually different but happen to have the same signature, an overriding method should not combine them. Explicit renaming and overriding, such as supported by the CLI [10], should be applied instead.

## 4. Dynamic monitoring

### 4.1. Monitoring protocols

An execution environment (EE) monitors method calls to an object according to the protocol of the object’s actual type. For each object subject to protocol monitoring, the EE maintains a transition system simulating the protocol of the object’s type. For each method call to the object, it checks if the object’s transition system is in a state allowing that method call. If it is not, an error is raised to the caller. Otherwise, when the method completes its execution, the EE updates the object’s transition system to simulate the corresponding transition. This update includes evaluating any predicate of the transition’s resultant states to possibly eliminate some non-determinism in the protocol. However, in general, a protocol may contain external non-determinism that cannot be eliminated.

**4.1.1. Monitoring ‘end’ states.** The EE also monitors if an object ends its life when its transition system is in an ‘end’ state. If an object does not, an error is noted, indicating this violation. An implementation of this monitoring mechanism depends on the object finalisation model of the underlying platform. For platforms with deterministic finalisation, e.g. by automatic or programmer-coded calls to object destructors, the EE may integrate with this mechanism to perform its check. On such a platform, an implementation may also choose to provide more useful error reporting. For platforms with non-deterministic object finalisation, e.g. with automatic garbage collection, the EE may leverage the garbage collector to perform its check. On such a platform, there is not a good mechanism to handle violations meaningfully. This is because the detection of a violation might occur in an entirely different thread, e.g. the garbage collector, from threads that created or used the object. Those threads and the relevant scopes may no longer exist at the time of detection.

**4.1.2. Monitoring actual types.** The BECON EE monitors a method call to an object according to the protocol of the object's actual type, not that of the object reference's static type or that of the type declaring the method. In some cases, these may be three distinct types with the object's actual type being a subtype of the other two.

In general, an object reference's static type is not known at run time, or in the binary components on native platforms. Even when all static types may be known, aliasing makes it difficult to monitor method calls according to protocols of object references' static types. For example, a new object reference may be assigned an existing reference of some other static type, i.e. not a freshly constructed object. It is not clear how a transition system for the new reference may be initialised because it must start in some state corresponding to the existing reference's current transition state or states. We specify a protocol conformance relation, but not yet a mechanism for explicit mapping between states of conforming protocols.

Monitoring a method call according to the protocol of the type declaring the method is similarly problematic. This is because there may be calls to methods of different declaring types but to the same target object.

On the other hand, monitoring according to the protocol of the object's actual type is simple and sufficient. First, each object has only one actual type throughout its lifetime. Therefore, it can be associated with a single transition system to consistently simulate exactly one protocol. Second, the protocol of the object's actual type conforms to a protocol declared by any supertype. Therefore, method calls obeying the protocol of the target object's actual type never violate any client's expectation.

## 4.2. Monitoring mandatory and forbidden calls

An EE monitors if a method implementation fulfils its mandatory and forbidden call obligations. The monitoring matches all method calls occurring in the same thread, during the execution of the method, against the method's lists of mandatory and forbidden calls. It covers both direct and nested method calls made by the method. Matching of actual calls against the list of expected calls respects ordering if required. If an impending call matches a specified forbidden call, the EE aborts the impending call and immediately reports an error to the caller of the executing method. At the end of the method execution, before a return completes, the EE checks if all mandatory calls have been satisfied and raises an error if not.

**4.2.1. Compressed call stack algorithm.** An elegant algorithm that performs the traversal of a com-

pressed call stack can be implemented for monitoring mandatory and forbidden calls. For each thread, there is a compressed call stack holding information about invocations of methods that have mandatory and forbidden calls to be monitored. Invocations of methods that do not have mandatory or forbidden calls need not be represented. Therefore, the stack corresponds to a compressed version of the real call stack. Each entry on the stack holds a copy of the complete lists of mandatory and forbidden calls of the corresponding method being invoked. Figure 2 illustrates this algorithm.

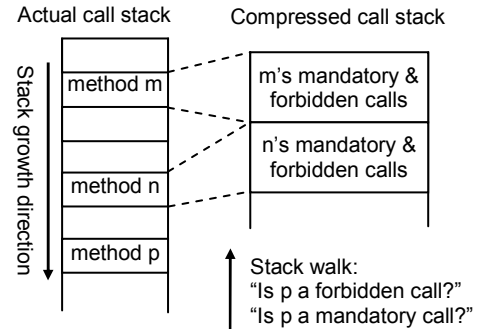


**Figure 2. Monitoring mandatory and forbidden calls**

For each impending method invocation, the EE walks the thread's compressed call stack, matching the method against mandatory and forbidden calls of outer invocations. This matching takes into account method overriding as specified in section 3.4. If a mandatory call matches the impending method, it is removed from the list holding it. For a list whose order is significant, only its first entry may match the impending method. If a forbidden call matches an impending method, an error is reported immediately. As a method invocation completes its execution, if it has specified mandatory or forbidden calls, its corresponding entry is popped from the compressed call stack. If the popped entry has a non-empty list of mandatory calls, then the returning method has not fulfilled its mandatory call obligation, and the EE reports an error to the method's caller.

This algorithm works well in covering both direct and nested calls with arbitrary depth. In addition, the traversal of the compressed call stack is relatively efficient because usually the number of calls to methods with mandatory or forbidden calls is a small portion of all method calls. Moreover, the EE may maintain a record of methods that are known to be targets of mandatory or forbidden calls. Then only a call to one of these methods requires a traversal of a compressed call stack. A further optimisation may seek to reduce stack traversal efforts when a method is called multiple times from the same call-frame.

## 5. Implementation

We have implemented support for call-ordering constraints on Rotor [19], as part of an implementation of the BECON contract system [22]. We augmented the ilasm assembler language to write call-ordering constraints. Our augmented assembler generates binary CLI components containing these constraints as metadata. We have also extended the CLI binary format to accommodate the new metadata. We modified the Rotor execution engine to run contracted components with configurable contract monitoring. The modified execution engine implements the algorithms for constraint monitoring as described above. The following ilasm snippet shows example constraint specifications.

```
.class public SavingsAccount extends Account {  
  // protocol specification  
  .contract _start _end state closed {}  
  .contract state open {}  
  .contract transit {  
    .eval instance void BeginSession()  
    .from closed  
    .to open  
  }  
  .method Deposit(float32 amount) cil managed {  
    .contract mandatory { // mandatory call  
      .eval BankAdmin::NotifyBalanceChange(...)  
    }  
    .contract notcall { // forbidden call  
      .eval void Account::Debit(float32)  
    }  
    // code for Deposit  
  }  
  // rest of class SavingsAccount  
}
```

**Figure 3. Example constraints in ilasm**

Our implementation remains fully backward compatible with other CLI implementation. First, contracted components run on other CLI execution engines, e.g. Microsoft’s .NET CLR, as if without contracts. Second, the modified Rotor execution engine runs non-contracted components with no side effects and negligible overhead.

## 6. Related work

Existing work has addressed cases of the call-ordering problem individually. In most instances, they present language-specific solutions. Our unifying solution addresses all call-ordering problems in a language-neutral manner.

Nierstrasz [16] studied the problem of object method availability. He proposed modeling active objects as processes. Method availabilities are captured in regular types, which are NFSA. An NFSA is described by an LTS. Nierstrasz also gave an algorithm for verifying that a regular type conforms to another. Regular types are the theoretical basis for our protocols. Nierstrasz did not provide an implementation.

Butkevich et al [6] also based their work on protocols for Java on Nierstrasz’s regular types. They added state predicates for resolving non-determinism. Their protocols for Java are written in an extended Java syntax. Their implementation involves instrumenting Java source code to insert wrapper objects that maintain transition machines simulating specified protocols. They monitor protocols according to statically declared types. This is problematic as we discussed earlier.

DeLine and Fähndrich [9] implemented support for protocols for .NET programs in the Fugue protocol checker. Their state machine protocols are essentially similar to protocols based on regular types, but without state predicates. Fugue performs static analysis of .NET code to check if usage of an object conforms to its protocol, but it does not check if two protocols are substitutable.

AsmL [2] includes mandatory calls in model programs, as a way to express synchronisation constraints such as a method *must* call another in its implementation. Runtime monitoring checks if an implementation is correct with respect to the model program. At run time the model program is executed in parallel with the actual implementation to maintain a simulated call stack. The actual implementation is instrumented to notify the runtime service of actual method calls and returns. These are checked against the simulated call stack to see if the mandatory calls are actually made.

AsmL’s mandatory calls are different from ours in several ways. First, AsmL mandatory calls specify target objects and arguments as model variables. This is more precise, but requires significantly more overhead to maintain the correspondence between model and actual variables. Second, AsmL mandatory calls require direct calls from an implementation, unlike ours, which may be matched by indirect calls. Finally, AsmL mandatory calls are not matched against over-riding methods.

Existing work [3, 4, 15, 20] deals with the re-entrance problem by using class invariants. This approach requires an object’s invariant to be satisfied for an incoming call to proceed. This is substantially different from our solution of using forbidden calls. As discussed earlier, forbidden calls are especially useful where invariant checking is difficult or too costly.

## 7. Summary

The temporal ordering of method calls in a program can present subtle problems to ensuring its correctness. The constraints on this ordering must be specified and enforced to help programs attain correct semantics.

We have presented a unifying solution using call-ordering constraints, namely protocols, mandatory calls

and forbidden calls. We have extended the notion of behavioural subtyping to include conformance relations between call-ordering constraints. We showed elegant algorithms for dynamic monitoring of programs' conformance to specified constraints. Our unifying solution has been implemented for an industrial platform.

## 8. Acknowledgement

We are grateful to the anonymous reviewers for their helpful comments. This work has been supported in part by a Monash Graduate Scholarship and a DSTC Top-up Scholarship for Nam Tran, and a Rotor award from Microsoft.

## 9. References

- [1] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School Workshop*, Noordwijkerhout, The Netherlands, May/June 1990, LNCS 489, Springer-Verlag, NY, 1991, pp. 60-90.
- [2] M. Barnett, W. Schulte, "Contracts, components and their runtime verification on the .NET platform", Technical Report MSR-TR-2002-38, Microsoft Research, April 2002.
- [3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte: "Verification of object-oriented programs with invariants", *Journal of Object Technology*, vol. 3, no. 6, June 2004, pp. 27-56.
- [4] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction, Lecture Notes in Computer Science*. LNCS 3125, Springer, July 2004. pp. 54-84.
- [5] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware", *Computer*, July 1999, pp. 38-44.
- [6] S. Butkevich, M. Renedo, G. Baumgartner, M. Young, "Compiler and Tool Support for Debugging Object Protocols", In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, San Diego, California, United States, 2000, pp. 50-59.
- [7] Luca Cardelli. Typeful programming. Technical Report 45, DEC SRC, 1989.
- [8] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'18)*, Berlin, Germany, March 1996, pp. 258-267.
- [9] R. DeLine, M. Fähndrich, "The Fugue protocol checker: Is your software Baroque?", Technical Report MSR-TR-2004-07, Microsoft Research, January 2004.
- [10] ECMA, *Standard ECMA-335: The Common Language Infrastructure*, December 2001.
- [11] R. B. Findler and M. Felleisen. Contract Soundness for Object-Oriented Languages. In *Object Oriented Programming, Systems, Languages, and Applications*, 2001.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [13] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, November 1994, pp. 1811-1841.
- [14] B. Meyer, *Object-Oriented Software Construction*, 2<sup>nd</sup> ed., Prentice Hall, Upper Saddle River, 1997.
- [15] L. Mikhajlov, E. Sekerinski, L. Laibinis, "Developing Components in the Presence of Re-entrance", In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume II*, 1999, pp. 1301 - 1320.
- [16] O. Nierstrasz, "Regular Types for Active Objects," In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), Prentice Hall, 1995, pp. 99-121.
- [17] R. H. Reussner, "An Enhanced Model for Component Interfaces to Support Automatic and Dynamic Adaption", In J. Hernández, A. Vallecillo, J.M. Troya (Eds.), *New Issues in Object Interoperability, Proceedings of the ECOOP 2000 Workshop on Object Interoperability*, Cannes, France, June 12-16, 2000, pp. 33-42.
- [18] B. Selic, "Protocols and Ports: Reusable Inter-Object Behavior Patterns", In *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2-5, 1999, Saint Malo, France, pp. 332-339.
- [19] D. Stutz, T. Neward, G. Shilling. *Shared Source CLI Essentials*. O'Reilly 2003.
- [20] C. Szyperski with D. Gruntz and S. Murer, *Component Software – Beyond Object-Oriented Programming*, 2<sup>nd</sup> ed., Addison-Wesley and ACM Press, 2002.
- [21] N. Tran, D. Abramson, C. Mingins, "A taxonomy of call ordering problems", In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, Busan, Korea, December 2004, pp. 120-129.
- [22] N. Tran, *Language Neutral Behavioural Contracts for Binary Components*, PhD thesis, Monash University, 2005.
- [23] D. M. Yellin and R. E. Strom, "Protocol Specifications and Component Adaptors", *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, March 1997, pp. 292-333.