PLUG AND PLAY SAFELY: RULES FOR BEHAVIOURAL COMPATIBILITY

A. J. H. Simons, M. P. Stannett, K. E. Bogdanov and W. M. L. Holcombe
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP
United Kingdom.

Abstract

The state of the art in component-based software development depends on the notion of interfaces and interface matching. This only partly solves the problem of safe component usage and substitution, dealing with *syntactic* compatibility. We present a specification method that describes under what conditions a component is *behaviourally* compatible with the expectations of an interface.

The method is based on the UML class and statechart diagrams, expressed as a set of rules for statechart refinement. Both refinement (UML *realisation*) and subtyping (UML *specialisation*) can be captured, allowing a designer to determine when a component matches the requirements of an interface, or when one component may safely be substituted for another. The rules are behaviourally safe under polymorphism with dynamic binding, up to the abstraction captured by the statecharts, and improve on previously published rules for behavioural compatibility.

Key Words

Components, behavioural compatibility, statecharts, subtyping, refinement.

1. Introduction

The eventual economic success of component-based software development will depend crucially on the willingness of developers to accept and trust third-party software components. Although the notion of interface matching is well-understood, no guarantees currently exist regarding the *behaviour* of third-party components and this represents a serious obstacle to component reuse. The "not invented here" syndrome is one aspect of developer mistrust. Experiences of component misbehaviour, such as the spectacular Ariane 5 disaster [1], seem to bear this out.

To improve the level of trust, an agreed specification method must be adopted by developers that not only describes when a component is type-compatible with a given interface, but which is capable of expressing when the component is *behaviourally* compatible with the expectations of an interface, at some suitable level of abstraction. Secondly, a testing method must be capable of guaranteeing that implemented components conform to the agreed specification.

MOTIVE (Method for Object Testing, Integration and Verification, EPSRC GR/M56777) is a combined state machine and algebraic approach to specifying and testing object-oriented and component-based systems. MOTIVE is based on the successful X-Machine specification and testing method [2, 3, 4, 5] and is further influenced by algebraic specification and testing methods [6, 7]. The fundamental goals of MOTIVE are:

- to preserve the scalability of the verification and testing obligations; and
- to provide guarantees of component and system correctness once testing is over.

This goal is reached through a novel combination of formal reasoning, design restrictions and a proven method for complete functional testing [3, 4].

In this paper, we concentrate specifically on the *refinement* [3] of statechart specifications. This can be used to support the UML [8] notion of *realisation* whereby abstract interfaces are realised by concrete components whose behaviour conforms to the interface. Object extension with subtyping is explained in the same formal framework, supporting the UML notion of *specialisation* and the matching of more specific components to more general interfaces.

2. Object Machines

In MOTIVE, the designer develops an Object Machine, a state machine describing the gross behaviour of an object or component in response to message requests, and later fills out a table of axioms in the associated Object Algebra, an imperative-flavoured algebra in the object-oriented style. For reasons of space, a discussion of the

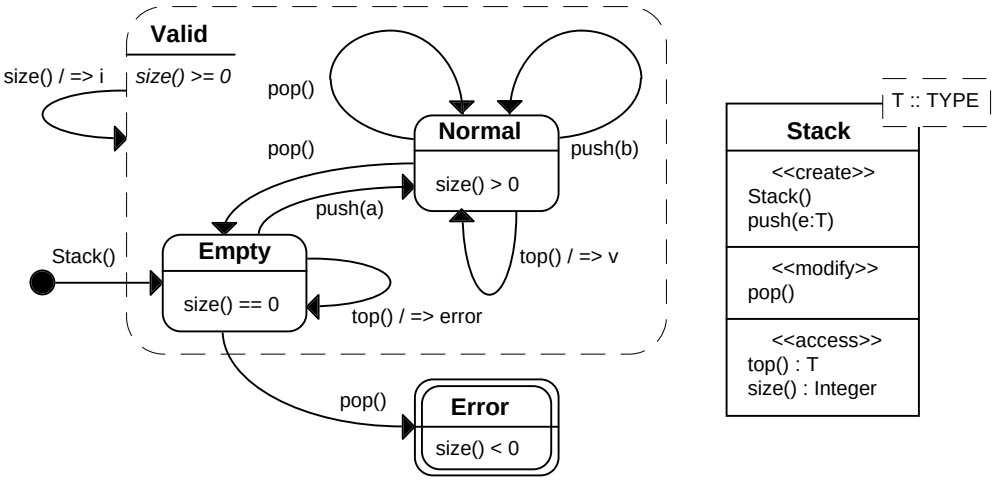

Figure 1: Object Machine Specification for a Stack

algebra is not possible in the current paper. Instead, we focus on the behavioural properties of component interfaces that can be captured solely using Object Machines.

Figure 1 illustrates a simple Object Machine specification for a *Stack*, which consists of a UML class diagram and a statechart diagram [8]. The class diagram partitions the public methods of *Stack* into algebraic constructors, transformers and observers, respectively labelled with the <<create>>, <<modify>> and <<access>> stereotypes. The statechart has *control states* chosen by the designer, corresponding to modes in which the *Stack* is perceived to react differently to messages. The *Empty* state encodes the limiting behaviour of *pop* and *top* (when they are undefined) and the *Normal* state is the regular state in which all methods are well-defined. The *Error* state is a halting state, representing an invalid object (in algebraic terms *undefined*). To avoid clutter, the diagram may group states into *regions* in which common reactions occur. The *Valid* region is shown by a dashed outline and transitions from the region boundary are considered *replicated for every enclosed state* [9].

The diagram has transitions describing the state-modifying behaviour of the <<create>> and <<modify>> methods. The nondeterminism of *pop* from *Normal* is resolved through postcondition guards, here the state-predicates governing entry to the next state. The self-transitions for <<access>> methods are also shown in the diagram, since they may yield state-contingent results (such as *top*) and so reveal the existence of interesting states.

There is a synergy between the Object Machine's state space and the algebraic category assigned to each method: <<create>> methods are the only ones that can reach new valid states; <<modify>> methods only ever reach previously visited states (the invalid *Error* state is discounted in this determination); and <<access>> methods never cause a state change. In an algebraic specification, axioms must be supplied to define precisely the meaning of each <<access>> and <<modify>> method in paired combination with every <<create>> method.

The *control states* chosen for an object are at a level of abstraction chosen by the developer, but they must always form a complete partition of its underlying *memory states*. In MOTIVE, the memory states of an object are defined abstractly as the Cartesian product of the ranges of its <<access>> methods under inductive construction, rather than concretely as the product of its attribute domains [10, 11], to facilitate the definition of abstract types.

3. Interface Realisation

The *Stack* specification developed in figure 1 captures the *abstract behaviour* of a *Stack* interface, describing not only the type signatures, but also the state-related behaviour that any implementation's methods must provide. A linked list could implement this state-space exactly, with an *Empty* state for the empty list and a *Normal* state for a chain of links.

By contrast, a self-resizing vector could implement the behaviour of the abstract *Stack*, but its state space would be different, since it would react differently when it was full and about to be resized in response to the next *push*. Realisation of an abstract interface frequently exposes further states in a more concrete Object Machine. This should always be in accordance with the designer's knowledge about the implementation strategy: the guiding principle is to model, as explicit states, those modes in which the object reacts differently to events.

Figure 2 shows the specification for a *DynamicStack*, which realises the abstract *Stack* interface. The statechart exposes new *Loaded* and *Full* states inside a *Normal*

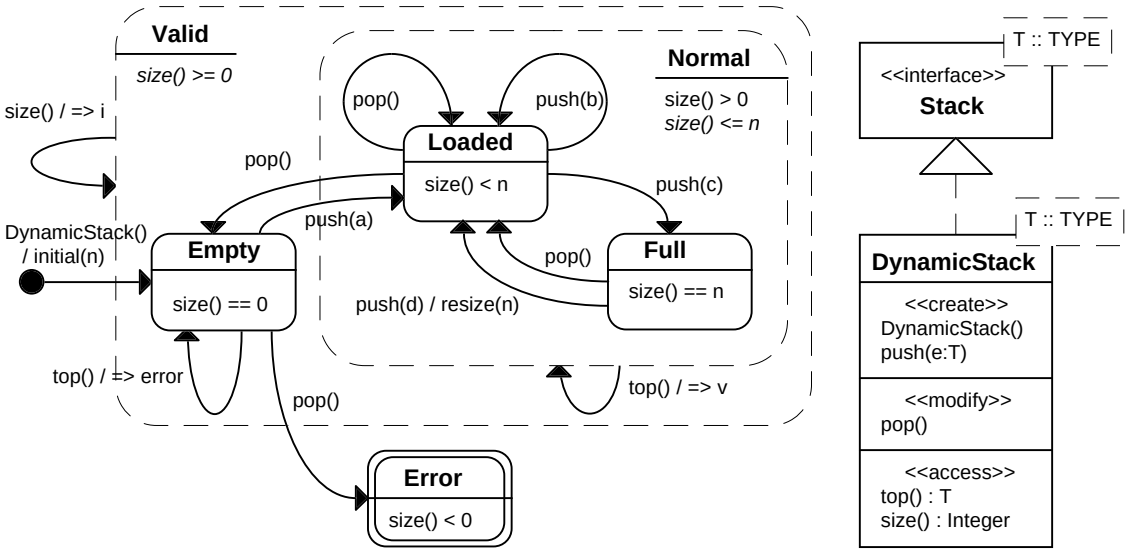

Figure 2: Object Machine Specification for a DynamicStack Realisation

region, which corresponds to the unrefined *Normal* state in figure 1. The states are motivated by the different resizing behaviour of the *DynamicStack*, when it reaches its *Full* state. The statechart observes several important rules of refinement:

1. newly-exposed states must completely partition a region corresponding to an unrefined state, up to any new assumptions made by the refinement;

2. entry and exit transitions crossing over the region boundary must similarly partition the entry and exit transitions to the old state;

3. transitions within the region must similarly partition the self-transitions of the old state, otherwise the refinement may deadlock;

4. and there may be no other transitions to or from states outside the region, otherwise the refinement may behave unexpectedly.

The three partitioning rules express more succinctly Cook and Daniels' state- and transition splitting rules for statechart refinement [12], but the extra prohibition rule 4 is a necessary addition. For example, if *push* from *Full* were to overflow and lead to the *Error* state, there would be some sequence of operations that would work correctly for a *Stack* but which would cause a *DynamicStack* to fail.

4. Component Substitution

In general, an interface may accept a component with more than the required methods. Compatibility is conventionally judged only in terms of providing methods matching the required signatures. However, now we must also consider extensions to *behaviour* and whether this violates the expectations of the interface. This is similar to the case in object-oriented programming where a variable of one type receives an object of some subclass type, and through which redefined methods are invoked by dynamic binding.

MOTIVE's underlying refinement model is a calculus of object types and subtyping. A subtype object should be usable in contexts where a supertype was expected [13]. An extended specification for a subtype must conform to the original type, both syntactically and behaviourally. In particular:

- the subtype may add methods to the original type, or replace these with methods having subtype (but typically, unchanged) signatures;

- the statechart of the subtype may expose new states and add extra transitions following the rules for statechart refinement.

A lending library example is developed in figure 3 as a counter-example to illustrate how breaking the rules of refinement from section 3 leads to extended types which are sytactically, but not behaviourally, compatible with their base types.

In figure 3, a *Reservable* type specialises a *Loanable* type by adding extra <<create>> methods and thereby introducing new states (cf the synergy between algebraic constructors and new object states noted in section 2). Assume that the original *Loanable* had two states, *Issued* and *Discharged*, which are now shown as regions partitioned exhaustively in the subclass *Reservable*.

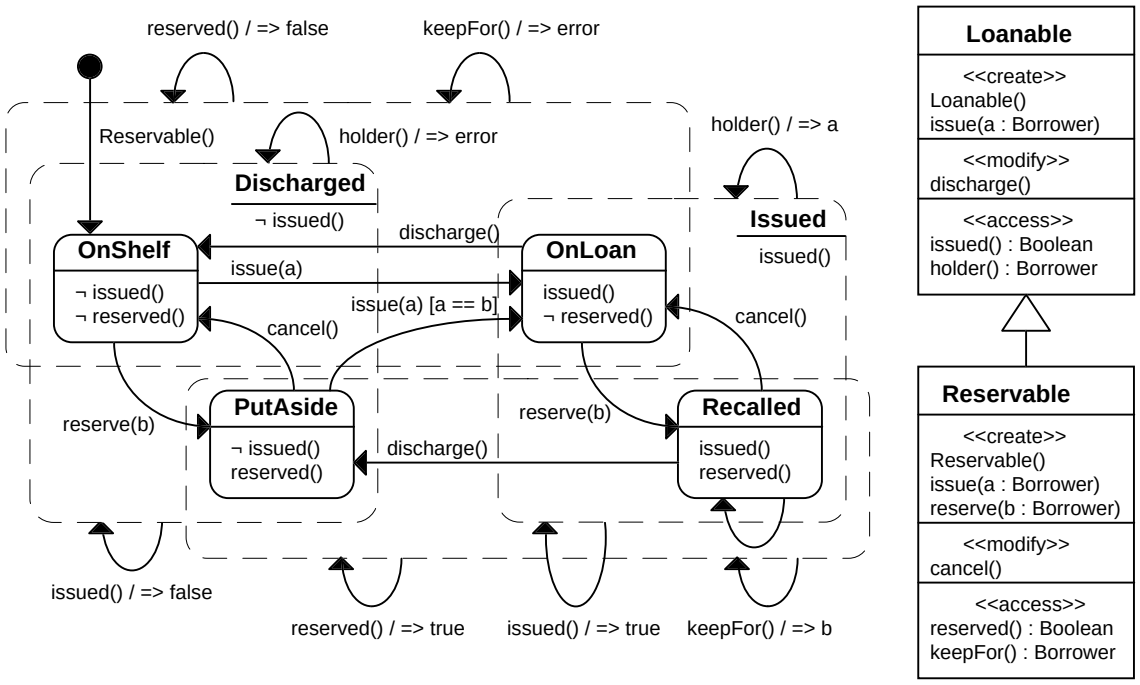

Figure 3: Object Machine for a Reservable Specialisation

We focus on the refinement of the *issue* and *discharge* transitions. In *Loanable*, these ran between the *Issued* and *Discharged* states. In the refined *Reservable*, these methods must now take into account the consequences of reservation.

Discharge is split into two transitions, running from distinct source to distinct target substates. These form a complete partition of the original transition, so are semantically equivalent (in fact, the splitting is a consequence of partitioning the source state space). However, the *issue* method is redefined to ensure that after a reservation, the item is only issued to the intended *Borrower*. Two transitions are explicitly given, but they do not completely partition the unrefined *issue* transition: there is an implicit self-transition: *issue(a) [a != b]* in *PutAside*, which is a null operation. So, in this one case a *Reservable* instance does not behave exactly like a *Loanable* instance.

It is important for subtyping that an instance of *Reservable* should appear as an instance of *Loanable*, when accessed through a *Loanable* reference variable. Syntactically, this is not a problem since *Reservable* merely adds extra methods to the base *Loanable* type and otherwise respects its interface [13]. Semantically, it must be possible to show that all sequences of method invocations to the *Reservable* instance leave it in states expected by the *Loanable* handle. So long as access is granted *only* through a base *Loanable* handle, there is no opportunity for *issue* and *discharge* to interact with *reserve* and *cancel*. However, if access is *also* granted through a *Reservable* handle, an interleaved *reserve* message may place the object in a state where *issue* is a null operation and, when dispatched through a *Loanable* handle, behaves unexpectedly.

The view of a *Reservable* object through a *Loanable* handle is obtained by collapsing the *Issued* and *Discharged* regions back to atomic states. Split transitions are lifted and recombined, but if they do not completely partition the original transition, then its behaviour has been altered. Lifting the implicit self-transition for *issue(a) [a != b]* in *PutAside* introduces a self-transition in the *Discharged* region which was not previously there. For this reason, we cannot consider a *Reservable* instance to be semantically a subtype of *Loanable*.

5. Conclusions

The basic premise of component substitutability is "no surprises", yet these examples show how difficult it can be to avoid unexpected behaviour or even failure. The syntactic rules for matching interfaces are well known: a component must provide at least as many methods as expected, and the signatures of those it provides must match (be subtypes of, with covariant results and contravariant arguments [13]) the expected signatures. However, previously published semantic rules for matching behaviour have ranged from the cautious to the liberal. We can compare some of these within our framework.

Liskov and Wing's 1993 revised notion of subtyping [14] highlighted the inadequacy of syntactic matching alone. It proposed one conservative form of subtyping in which all additional methods of subtypes were strictly definable in terms of existing methods. This ensured that properties proved for supertypes would also hold for subtypes. In our framework, it is clear why this works, since the restriction is equivalent to providing only <<modify>> and <<access>> methods in subclasses. No new states are reachable in the subclass that could not already be reached in the superclass. We allow more flexibility than this, by permitting extra <<create>> methods in subclasses, if these only generate exposed substates that are wholly-contained in existing states.

Cook and Daniels' rules for state- and transition splitting [12] and McGregor's rules for state machine refinement [15, 10] are close to our rules. They allow state partitioning, transition splitting, the addition of extra transitions and the addition of extra states in subclasses. The first two correspond exactly to our partitioning rules for statechart refinement. The third corresponds to the Liskov-style extension above. The fourth is too liberal and is not safe if subtype objects are aliased through both base and subtype handles. According to our rule 1, the new states introduced in a subtype must always be substates of some atomic state in the original type; according to our rule 4 there can be no further transitions over the region boundary to new, external states.

Why can we not introduce new external states, as allowed by McGregor, Cook and Daniels [10, 12, 15] ? Imagine if *Reservable* had added the states *PutAside* and *Recalled* externally to *Issued* and *Discharged*. Then, a message sequence: *r = new Reservable(); r.issue(); r.reserve(); r.discharge();* will put this object in the *PutAside* state (here, considered disjoint from *Discharged*). A *Loanable* handle aliasing a *Reservable* object always expects this to be in the *Discharged* state after a *discharge*; however, if an interleaved *reserve* message is sent through a *Reservable* handle, the next *discharge* sent through the *Loanable* handle will leave the object in the unexpected *PutAside* state. Not only is this unrelated to the *Discharged* state, but subsequent *issue* messages might not work as intended.

It should always be possible to view a refined message sequence as a base sequence, by forgetting the extra transitions introduced in the subclass. This cannot be done if the subclass introduces new external states, rather than exposing nested substates. Consider now that the situation is restored, as in figure 3, in which all new states are exposed substates. Here, the *putAside* state is wholly contained by *Discharged*. The significance of this is that an object will always be left in (some substate of) the *Discharged* state after a *discharge* message is received, even if the object receives an interleaved *reserve* message.

The rules given here for state machine refinement are therefore superior to previously-published rules. They allow the development of semantic specifications for components and capture the notions of interface realisation and also component specialisation (through subtyping). The specification rules ensure the semantic safety of systems in which components of more specific types are plugged into interfaces of more general types. In particular, they preserve the behavioural expectations of interfaces in the context of multiple handles of different types referring to the same component.

Acknowldegement

This research was sponsored by EPSRC GR/M56777 "MOTIVE".

References

[1] J. L. Lions, *Ariane 5 Flight 501 Failure, Report of the Inquiry Board*, http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, July 1996.

[2] G. Laycock, The theory and practice of specification based software testing, *PhD Thesis*, University of Sheffield, 1992.

[3] F. Ipate and W. M. L. Holcombe, An integration testing method that is proved to find all faults, *Int. J. Comp. Math.*, *63*, 1997, 159-178.

[4] W. M. L. Holcombe and F. Ipate, *Correct systems: building a business process solution*, Applied Computing Series (London: Springer Verlag, 1998).

[5] K. E. Bogdanov, Automated testing of Harel's statecharts, *PhD Thesis*, University of Sheffield, 2000.

[6] R. K. Doong and P. Frankl, The ASTOOT approach to testing object-oriented programs, *ACM Trans. Softw. Eng. Meth.*, *3*(4), 1994, 101-130.

[7] H. Y. Chen, T. H. Tse, F. T. Chan and T. Y. Chen, In black and white: an integrated approach to class-level testing of object-oriented programs, *ACM Trans. Software Eng. and Methodol.*, *7*(3), 1998, 250-295.

[8] Object Management Group, *The UML 1.4 Specification, http://www.omg.org/uml/*, 2002.

[9] A. J. H. Simons, On the compositional properties of UML statechart diagrams, *Electronic Workshops in Computing: Rigorous Object-Oriented Methods 2000*, series ed. C J van Rijsbergen, British Computer Society, 2000, 8.1-8.12.

[10] J. D. McGregor, Constructing functional test cases using incrementally-derived state machines, *Proc. 11th Int. Conf. Testing Computer Software*, Washington, USPDI, 1994.

[11] C. D. Turner and D. J. Robson, A state-based approach to the testing of class-based programs, *Software Concepts and Tools, 16*(3), 1995, 106-112.

[12] S. Cook and J. Daniels, *Designing object-oriented systems: object-oriented modelling with Syntropy*, (Englewood Cliffs, NJ: Prentice Hall, 1994).

[13] L. Cardelli and P. Wegner, On understanding types, data abstraction and polymorphism, *ACM Computing Surveys, 17*(4), 1985, 471-521.

[14] B. Liskov and J. M. Wing. A new definition of the subtype relation, *Proc. ECOOP '93, LNCS 707*, Springer-Verlag, 1993, 118-141.

[15] J. D. McGregor and D. M. Dyer, A note on inheritance and state machines, *Software Engineering Notes, 18*(4), 1993, 61-69.