Abstract

The rules for inheritance of classes with respect to data and function members are well defined. For example, the proposals for programming by contract in Eiffel ensure additional consistency between superclasses and subclasses. In object-oriented design, it is common to capture the behaviour of classes with lifecycles which are expressed in the form of finite state machines. In this context, there are very few proposals for what constitutes consistency between superclasses and subclasses.

This paper presents proposals for consistency between superclasses and subclasses in the context of the Petri Net formalism, which is a form of finite state machine with explicit provisions for concurrency. The paper cites the applicability of these proposals in the context of network protocols, and argues for a similar applicability in the context of object lifecycles.

1. Introduction

The concept of class hierarchies derived during object-oriented analysis and design has traditionally assumed consistency between superclasses and subclasses [13]. Such consistency is captured by the distinction between is-a and has-a relationships, which is made more precise by the use of assertions in Eiffel. This has been extended to the notion of programming by contract and embodied in the BON method [24].

Inheritance is also used for code sharing or code reuse, but this is primarily an implementation issue, rather than a design issue. Consequently, this paper will restrict its attention to class hierarchies with consistency between superclasses and subclasses. We do not, however, insist on full substitutability [25], since we envisage a software development context where a system is being progressively refined rather than merely substituting one component for another.

Object behaviour is traditionally captured by object lifecycles in the form of finite state machines. Notions of consistency between the lifecycles associated with superclasses and subclasses are scarcely formulated, let alone generally adopted. The current paper explores this issue and proposes a solution based on recent Petri Net research [10]. The essence of the solution is that every (complete) refined transition sequence (i.e. a transition sequence of a subclass lifecycle) has a corresponding abstract sequence (i.e. a transition sequence of the superclass lifecycle).

The paper first considers other proposals in this area. The Shlaer-Mellor method [21, 22] was perhaps the first to explore the relationship of object lifecycles between superclasses and subclasses, and hence it is discussed in §2. The Unified Modelling Language [1, 17] is becoming dominant in terms of method majority, market share, and industry standardisation. Currently, its specification has little to say on the matter of behavioural consistency — there are some comments which are more by way of descriptive observations rather than prescriptive recommendations. They are considered in §3.
As noted above, this paper advocates the use of behavioural consistency between the lifecycles of superclasses and subclasses in terms of being able to map refined transition sequences into abstract transition sequences. The forms of refinement which guarantee this consistency have been identified in the context of Petri Net research [10]. (Petri Nets are an extended form of finite state machine with standard support for concurrency. Therefore Petri Nets provide a close parallel with the extended finite state machines adopted in UML.) The forms of refinement are presented and discussed in §4.

The application of the Petri Net work to UML state machine refinement is discussed in §5, and the conclusions are found in §6.

2. The Shlaer-Mellor proposals

The Shlaer-Mellor method for object-oriented analysis and design can probably claim to be the first and most thorough in its use of object lifecycles to capture the dynamics of an object-oriented system [21, 22]. No doubt, this was prompted by its focus on real-time and embedded technology.

It has been noted that the great strengths of the method are: clear separation of subject matter (so that specifications and implementations can be separately reused); executable software specifications (so that software can be verified prior to implementation); implementation by translation (which supports optimisation without impacting on the specification); and a rigorous engineering process (where each development step has well-defined techniques, deliverables, and quality criteria) [15]. Certainly, the use of executable software specifications means that there is less room for incompleteness and inconsistencies.

Unfortunately, the Shlaer-Mellor method bears the marks of an early object-oriented approach. This is particularly apparent in its terminology and its identification of supertypes and subtypes. Thus, the sets of similar “things” or instances are referred to as objects rather than classes, and thus they refer to instances of objects rather than instances of classes. In capturing the relationship between a number of instances, the supertype is formed from the set of common attributes, while the subtypes are formed from the distinct attributes. This means that an instance is an instance of the supertype as well as of the subtype, with some attributes contributed by each.

Fig 2.1: The lifecycle of a simple microwave oven
As noted above, the Shlaer-Mellor method can lay claim to a careful consideration of the
dynamics of objects, as motivated by its interest in real-time and embedded systems. The format
of a typical object lifecycle (for a simple microwave oven) is shown in fig 2.1. The states are
uniquely identified by numbers, are drawn as rectangles, and are annotated with the actions
performed on entry to the state. The transitions are drawn as arcs, and are annotated with the
events that cause the transitions. Thus, the transition from state 1 to state 2 occurs when the event
\( V1 \) is received, namely when the button is pushed. The action(s) performed on entry to state 2
are to set a timer, turn on the light, and energise the power tube.

The flow of events between object lifecycles is captured in the \textit{Object Communication Model},
a simple example being found in fig 2.2. It shows that events \( V1, V3, V4 \) sent to the \textit{Oven}
originate externally (with the \textit{Cook}).

\begin{center}
\textbf{Fig 2.2: Object Communication Model for a microwave oven}
\end{center}

We do not consider the other dynamic models of Shlaer-Mellor — the \textit{Object Access Model}
(for synchronous communication between objects) and the \textit{Action DataFlow Diagrams} (for more
detailed elucidation of the actions performed on entry to states). These have been considered
elsewhere [8].

The Shlaer-Mellor method is unique in exploring the relation between the lifecycles of
superclasses and subclasses, though this is expressed in terms of supertypes and subtypes (as
noted above). Where subtypes share a common behaviour, this is captured in a lifecycle of the
supertype. Where subtypes have distinct behaviours, they are captured as separate lifecycles for
each subtype. Where there is a partial overlap of behaviours between subtypes, the common
behaviour is captured in the supertype and the distinct behaviours are captured in the subtypes. Since an instance is an instance of both supertype and subtype, so its lifecycle is formed by splicing together the supertype and subtype lifecycles. A simple example is shown in fig 2.3, where the common behaviour in the supertype is shown in bold. The splicing of events and/or states is carefully specified (though we do not consider it here).

![Splicing of lifecycles for supertype and subtypes](image)

**Fig 2.3: Splicing of lifecycles for supertype and subtypes**

Thus, the Shlaer-Mellor method addresses the composition of lifecycles which allows for consistency between superclasses and subclasses, even though it is not mandatory. It is a form of code sharing even if it does not enforce behavioural compatibility.

3. The UML proposals

The Unified Modeling Language (UML) has evolved out of the Booch, OMT and OOSE methods [17]. None of these had dynamic models as carefully defined as Shlaer-Mellor. OMT, for example, advocated the use of static (class) models, functional (dataflow) models, and dynamic (statechart) models [19]. These three models were seen as capturing different perspectives of the problem. They were combined in the first step of Object Design, but there was little in the way of guidance on how this should be done. In contrast to the detailed guidelines on deriving the static class models, the dynamic statechart models received minimal attention. Not surprisingly, there was no consideration of the relationship of state diagrams for superclasses and subclasses.

UML has given much more attention to dynamic models in general, and state machine models in particular. (Version 0.9 explicitly allowed the use of Shlaer-Mellor object lifecycles.) While the statechart notation has been adopted, it is stressed that the statecharts are prepared on a per-object basis rather than for modelling processes, as was originally the case for the statechart formalism [3]. Thus, a typical statechart might appear as in fig 3.1. This is hierarchical — states are drawn as rounded rectangles and may be nested. Thus, the *Active* state of a phone encompasses the substates *DialTone*, *Timeout*, etc.

UML version 1.3 includes a cursory consideration of the relationship of state diagrams for superclasses and subclasses — *State machine refinement* is the title of an example in subsection 2.12.5 Notes (for state machine semantics). The proposals are called *useful heuristics* since the topic is still the subject of research. Therefore, it is explicitly stated (page 2-155) that state machine refinement as defined here does not specify or favor any specific policy of state machine refinement. *Instead, it simply provides a flexible mechanism that allows subtyping (behavioral compatibility), inheritance (implementation reuse), or general refinement policies.* [17]
The least constrained refinement policy is referred to as general refinement. This allows the addition and removal of states and transitions; refined states may have different incoming and outgoing transitions; refined transitions may have different source and destination nodes; refined guards may have different guard conditions; refined action sequences may have different actions. In fact, this is so unconstrained that it is unclear why it is called refinement at all.

A more constrained refinement policy is referred to as strict inheritance. This policy allows the addition or replacement of features, but not their deletion. It is guided by the rather indirect principle of disabling refinements which may lead to non-strict inheritance once the state machine is implemented. It is not clear what kind of implementation is intended, and it appears that inheritance for code reuse is in mind. This is translated into requiring that refined states may have some of the same incoming transitions but more outgoing transitions; refined states may have more substates and may change their concurrency attribute; refined transitions should have the same source but may have a different target; refined guards may have different guard conditions; action sequences have some of the same actions. Following the general principle enunciated in the introduction, we consider this form of refinement inappropriate since it is based on the use of inheritance for code reuse at the analysis or design stage, and not just at the implementation stage.

The most constrained refinement policy is referred to as subtyping. This is intended to guarantee the substitutability principle and thus maintain behavioural compatibility. Behavioural compatibility is interpreted as preserving pre/post condition relationships of applying events/operations on the type, as specified by the state machine. Consequently, states and transitions are only added, not deleted. Further, refined states have the same outgoing transitions but may add others, and may have a different set of incoming transitions; refined states may have a larger set of substates and may change their concurrency property from false to true; a refined transition may have as destination a substate of the original destination state; a refined guard may weaken the original guard; a refined action sequence may extend the original sequence with additional actions.

It is apparent that these constraints are still motivated by some kind of implementation of the state machines rather than properties of the state machines themselves. Thus, the concentration...
on pre/post conditions is appropriate for the definition of classes and methods but not for finite state machine behaviour, where one should consider the occurrence of transition sequences. Thus, we consider that these proposals are also inappropriate. Mellor and Wilkie comment that actions in UML are incomplete, potentially inconsistent, and that subtleties are glossed over [12]. Perhaps this is hardly surprising given that the OMG issued a request for a precise action semantics in November 1998, and it is only due for voting in November 2000.

4. Behaviour refinement for Petri Nets

In this section, we present proposals for the refinement of lifecycles in the context of Petri Net theory. Petri Nets are a form of finite state machine with explicit support for concurrency. The particular form of Petri Net that we discuss is that of Coloured Petri Nets (CPNs) [4]. We introduce the notation of CPNs and illustrate with a simple example of a library loans system which is adapted from an example found in [2].

State components are called places and are drawn as circles or ovals. Places may contain tokens, which consist of values of a particular type. State changes are called transitions and are drawn as squares, rectangles or bars. The arcs connecting places to transitions and vice versa indicate how transitions affect the state, their direction indicating the direction of token flow. Thus, input arcs indicate the consumption of tokens, while output arcs indicate the generation of tokens. The arcs may be annotated with the particular tokens which need to be transferred. The transition firing modes will determine the particular values of these annotations.

In our simple library example, a book will have some associated information, such as the author(s), title, publisher, etc. In a CPN, each book can be represented by a token, and this information can be captured in the token type called Book. Each book will be in one of several states such as available, on loan, and overdue. These states can be represented by places in the Petri Net, with the presence of a book token in a place indicating that the book is currently in that state. Thus a CPN for the library books could be as shown in fig 4.1.

```
Available
  "b"
  "Pay fine" (b,u)
  "Loan expires" (b,u)
  "On loan" (b,u)
  "Overdue" (b,u)

Borrow
  "Return" (b,u)

\textbf{Colours:}
C(\text{Available}) = \text{Book}
C(\text{On loan}) = \text{Book} \times \text{User}
C(\text{Overdue}) = \text{Book} \times \text{User}
C(\text{Borrow}) = \text{Book} \times \text{User}
C(\text{Return}) = \text{Book} \times \text{User}
C(\text{Loan expires}) = \text{Book} \times \text{User}
C(\text{Pay fine}) = \text{Book} \times \text{User}
```

\textbf{Fig 4.1: CPN Books — the lifecycle of library books}

In this case, we have not shown all the details of the net. Thus, the token types are not indicated in full, nor is the relation between the transition firing modes and the variables inscribing the arcs. Clearly, however, the firing mode for transition Borrow will include sufficient information to identify the book \( b \) and the user \( u \).

A borrower or user of the library will have some associated information, such as their name, contact details, classification of membership, etc. Again, each user can be represented by a token type called User, and the various states of the user can be indicated by places. In this case, it is convenient to have only one place to indicate the state of the user with transitions indicating the possible actions such as borrowing or returning a book and paying a fine. Thus, a CPN for the library users could be as shown in fig 4.2.
The two CPNs above could be combined into a composite system by fusing the similarly-named transitions in the two nets. Clearly, many more details could be specified, but the example above will be adequate for our purposes of illustrating the forms of refinement which we wish to support.

4.1. Type refinement

The first and simplest form of refinement, which we call *type refinement*, is to incorporate additional information in the net in the tokens and firing modes. However, each value of the refined type can be projected onto a value of the abstract type. For example, it may be desirable to introduce a further classification of books to vary the loan period. As far as the subnet for the books is concerned, this will simply involve extending the token type for the places, and extending the corresponding type for the various transition firing modes. The changes will affect the firing of the *Loan expires* transition, especially in the composite system. It is certainly the case, however, that if there is a behaviour of the refined system, then there would be a corresponding (projected) behaviour of the abstract system. This is the generic behavioural constraint which we require for acceptable refinements.

4.2. Subnet refinement

The second form of refinement, which we call *subnet refinement*, is to augment a subnet with additional places, transitions and arcs. (We also classify as subnet refinement the extension of a token type or mode type to include extra values which are independent of previous processing. Here, these values of the extended type are not projected onto values of the abstract type (as in type refinement) but are ignored in the abstraction.) For example, it may be appropriate to cater for the reservation of books, as shown in fig 4.3.

---

**Fig 4.2: CPN Users — the lifecycle of library users**

The two CPNs above could be combined into a composite system by fusing the similarly-named transitions in the two nets. Clearly, many more details could be specified, but the example above will be adequate for our purposes of illustrating the forms of refinement which we wish to support.

4.1. Type refinement

The first and simplest form of refinement, which we call *type refinement*, is to incorporate additional information in the net in the tokens and firing modes. However, each value of the refined type can be projected onto a value of the abstract type. For example, it may be desirable to introduce a further classification of books to vary the loan period. As far as the subnet for the books is concerned, this will simply involve extending the token type for the places, and extending the corresponding type for the various transition firing modes. The changes will affect the firing of the *Loan expires* transition, especially in the composite system. It is certainly the case, however, that if there is a behaviour of the refined system, then there would be a corresponding (projected) behaviour of the abstract system. This is the generic behavioural constraint which we require for acceptable refinements.

4.2. Subnet refinement

The second form of refinement, which we call *subnet refinement*, is to augment a subnet with additional places, transitions and arcs. (We also classify as subnet refinement the extension of a token type or mode type to include extra values which are independent of previous processing. Here, these values of the extended type are not projected onto values of the abstract type (as in type refinement) but are ignored in the abstraction.) For example, it may be appropriate to cater for the reservation of books, as shown in fig 4.3.

---

**Fig 4.3: Subnet indicating the processing of book reservations**

Here, we have added a place to hold the reservation status for each book, and the transition *Borrow* will only fire if there is a compatible reservation status on the book for the given user. (We use the value *(b,−)* to indicate that no-one has a reservation for book *b*, and the value *(b,u)* to
indicate that user $u$ has a reservation on book $b$.) Again we satisfy the constraint that if there is a refined behaviour, then there is a corresponding (restricted) behaviour of the abstract system, but not necessarily vice versa.

4.3. Node refinement

The third form of refinement, which we call node refinement, is to replace a place (or transition) by a place (or transition) bordered subnet. We advocate the use of canonical forms of such refinements. The basis for a canonical place refinement is given in fig 4.4.

![Canonical place refinement](image)

**Fig 4.4: Canonical place refinement**

It has separate input and output border places — in this case there are two of each. Each input (output) border place may have more than one incident input (output) arc from (to) the environment. Each input border place has an associated accept transition which will transfer tokens from the border place to an internal place, here called buf. Similarly, each output border place has an associated offer transition which will transfer tokens from place buf to the output border place. All the border places and the place buf have the same token type, which is also the mode type shared by the accept and offer transitions. None of these transitions has a guard to constrain the flow of tokens. Clearly, such a canonical place refinement can be considered to have a corresponding abstract marking given by the sum of tokens in the border places and the internal place buf.

An arbitrary place refinement will be of the form of the basis of a canonical refinement (as above) augmented by subnet refinement which extends the accept and offer transitions.

In our running example, such an incremental change might be the identification of the details of processing a book once it has been returned. In other words, the place Available might be replaced by a subnet which takes into account the delay in reshelving a book, the possibility of repairs, etc. The node refinement for this place is shown in fig 4.5.

Note that this has one input border place called Returned and one output border place called On shelves. The Accept and Offer transitions, together with the internal place Buf constitute the basis of the canonical place refinement. Further activity is achieved by the subnet refinement which extends transitions Accept and Offer. This subnet retains the identity of the books, and hence this information (also) determines the abstract marking of the subnet. Thus, the place Buf is redundant, since its marking is equivalent to the sum of markings of places For checking, Under repair, Irredeemable, For shelving. (It is commonly the case that the place Buf is redundant in such place refinements.) Further, this abstract marking is not modified by the various actions internal to the subnet. Clearly, a refined behaviour of the net will have a corresponding abstract behaviour, though the reverse will not necessarily be the case. For example, it may be that a book is damaged to such an extent that its return to the shelves would be indefinitely delayed, in which case further borrowing of that book would be disallowed.
For transition refinement, the canonical basis is given in fig 4.6. It has separate input and output border transitions — in this case there are two of each. Each input (output) border transition may have more than one incident input (output) arc from (to) the environment. Each input border transition has an associated place recd, which receives a token equal to the abstract firing mode, when the input border transition has fired with that mode. The transition switch can fire when all the input border transitions have fired (with the matching abstract firing mode), thereby completing the input phase. It removes the matching tokens from the recd places and puts corresponding tokens into all the send places. There is one such send place associated with each output border transition. Once such a token is available the output border transition can fire (with the same abstract firing mode). Initially, all the recd and send places are empty. The abstract firing of the transition refinement commences with the firing of any of the input border transitions and is completed when all the matching output border transitions have fired and the recd and send places are again empty. Only such completed firing sequences will have corresponding abstract firing sequences. The canonical construction ensures that input border transitions fire before the corresponding output border transitions, ensuring the enabling of the corresponding abstract transition.
Fig 4.7: Refined Borrow transition

Even though the above three forms of refinement can be clearly identified and analysed in isolation, they will commonly be used in combination in practical applications.

The above proposals differ from those typically found in the context of High-Level Petri Nets (such as CPNs [4]). There, abstraction is used to aid the process of developing a Petri Net model, with the abstraction subsequently being discarded when the model is simulated or analysed [18]. Thus CPNs, as formalised by Jensen [4] and implemented in the Design/CPN tool [5], provide substitution transitions for building Hierarchical Coloured Petri Nets (HCPNs). Substitution transitions are like macros or textual substitution — they maintain structural compatibility, but there is no concept of abstract behaviour. The semantics of the construct are defined in terms of textual (or graphical) substitution.

Our proposals are therefore more constrained than substitution transitions, since they require behavioural consistency between a refinement and its corresponding abstraction. This is valuable for understanding a hierarchical model and also has benefits for analysis. Current indications seem to indicate that these forms of refinement are applicable in practice. Two network protocols which have been published in the literature [11, 14] have been easily reformulated in terms of these refinements [16, 23].

5. Implications for UML

The Coloured Petri Nets identified above have a lot in common with the state machines mandated for use in UML. There is a natural correspondence between state components in the two models and between state transition components. The inclusion of actions in a UML statechart is more involved but can be captured by token modification and additional transitions [8].

There are two significant differences with respect to state components. Firstly, the tokens resident in CPN places capture all the data of a system, whereas the data associated with a UML state machine is captured in three ways — as the current state of the machine, as the globally accessible data values associated with the object, and as the events in transit between state machines. Clearly, CPNs will wrap the first two together into one set of places, and will have a separate set of places to support event transmission. In the first case, the token type will include the values of object components, while in the second case, the token type will include the event data. Secondly, multiple tokens may correspond to multiple objects with the same lifecycle, whereas state machines normally assume one associated object. This is not a significant issue since the two views just represent different ways of folding Petri Nets [7].

The correspondence between state transition components is even more direct, given that a similar notation is adopted for so-called concurrent transitions in UML. The bar notation is only used for such transitions in UML, while they are always used in the Petri Net context. While UML transitions do not directly specify firing modes, nevertheless the event parameters and the access to the object state will determine the CPN transition firing modes.
It is also worth noting that the form of state machines supported by UML are more constrained than the possibilities available as CPNs. This is primarily apparent in the fact that UML requires complex states to be decomposable into an and-or tree. It is stated (p211) that such well-formedness rules have been adopted to avoid inconsistencies, including deadlocks, multiple occupation of a state, and other problems which have been extensively studied under Petri net theory. Thus a complex state may be decomposed into a set of mutually exclusive substates (an “or” decomposition) or into a set of concurrently held substates (an “and” decomposition) [20].

Given the above correspondences between CPNs and object lifecycles, it is appropriate to use the above forms of refinement to maintain behavioural consistency between the lifecycles of superclasses and subclasses, within the constraints of UML state machines, as detailed below.

**Type refinement** allows the replacement of types by subtypes provided that the subtype values can be projected onto supertype values. As noted above, such data values will arise as the value of an object and as event parameters. Thus, this form of refinement translates into UML as maintaining consistency between the refined and abstract value of an object, and by maintaining behavioural consistency between a refined and abstract lifecycle in the presence of refined events. Thus, if you consider the response to a refined event and then abstract the associated states, then you should get the same response to the corresponding abstract event.

**Subnet refinement** allows the addition of states and transitions provided that a refined transition sequence can be mapped to an abstract sequence by ignoring the additional components. In UML, this means that additional concurrent substates can be introduced provided that the associated transitions respond to (and produce) new events. Similarly, new self loops can be introduced (for new events) even if these involve a sequence of new states and new transitions. The important thing here is that the occurrence of that loop returns to an original abstract state without having traversed other abstract states or processing preexisting abstract events.

**Node refinement** allows a state to be refined into a hierarchical state with an arbitrary number of components. The key property is that the refined state always has a corresponding abstract state. In UML, this corresponds to replacing a simple state by a complex state, and the required property is already guaranteed by the well-formedness rules for such complex states (as noted above). Similarly, node refinement allows a transition to be refined into a sequence of transitions. (A concurrent transition can be refined into a more complex set of transitions.) Each complete firing of such a sequence of transitions will correspond to a firing of the original abstract transition.

### 6. Conclusions and further work

This paper has argued that the consistency between subclasses and superclasses which is normally required of class hierarchies (especially at the analysis and design stages) should be reflected by consistency between the corresponding object lifecycles. We have observed that little has been done in this regard even by the detailed Shlaer-Mellor method. The current UML specification recognises that this is an issue but contends that it is a matter for current research.

This paper has identified three forms of refinement which maintain behavioural consistency in the context of Coloured Petri Net research. Preliminary investigations indicate that behavioural consistency, as identified in this paper, occurs commonly in practice [16, 23]. The three forms of refinement we have identified are type refinement, subnet refinement and node refinement.

Given the natural correspondence between Petri Nets and generalised statecharts (as mandated for UML), these forms of refinement are easily transported into the UML context. They all maintain behavioural consistency in the sense that every complete refined transition sequence has a corresponding abstract transition sequence. This is more precise than behavioural consistency specified in terms of state machine implementation, as currently found in the UML specification.

There are a number of interesting avenues for further work. It would be appropriate to
formulate a more extensive and detailed set of rules appropriate to the UML specification. We are continuing to investigate the range of applicability of these forms of refinement to practical problems [6], particularly in the UML context. We are also implementing these forms of refinement in a Petri Net simulator [9] and are examining the benefits for analysis which can take advantage of the behavioural consistency between refinement and abstraction.

References