Compositionality and Refinement in Model-Driven Engineering

Jim Davies, Jeremy Gibbons, David Milward, and James Welch

Department of Computer Science, University of Oxford, Oxford OX1 3QD, UK
{firstname.lastname}@cs.ox.ac.uk

Abstract. Model-driven engineering involves the automatic generation of software artifacts from models of structure and functionality. The use of models as 'source code' has implications for the notions of composition and refinement employed in the modelling language. This paper explores those implications in the context of object-oriented design: establishing a necessary and sufficient condition for a collection of classes to be treated as a component, identifying an appropriate notion of refinement for the generation process, and investigating the applicability of data and process refinement to object models.

Keywords: formal methods, model-driven, object orientation, compositionality, refinement, inheritance

1 Introduction

Compositionality is a fundamental notion in software engineering, and an important property of design methods and modelling languages. A language is compositional for a notion of meaning M when the meaning of a compound expression is determined by the meanings of its components. That is, for every means of composition $\oplus$ in the language, there is a function $f_\oplus$ such that

$$M \llbracket A \oplus B \rrbracket = f_\oplus (M \llbracket A \rrbracket, M \llbracket B \rrbracket)$$

This is an essential tool for tackling complexity: a system may be designed, implemented, and analysed as a collection of smaller components.

Refinement describes the intended relationship between specification and implementation, or between a given component and a suitable replacement; the intention being that the meaning of the implementation should be consistent with that of the specification. In this context, meaning is often described in terms of the range of possible behaviours or effects, and B is a refinement of A, written $A \sqsubseteq B$, if and only if every behaviour of B is also a behaviour of A: that is

$$A \sqsubseteq B \Leftrightarrow M \llbracket A \rrbracket \supseteq M \llbracket B \rrbracket$$

Refinement is another essential tool for tackling complexity, allowing the comparison of descriptions at different levels of abstraction, and checking that one component may be safely replaced with another.

Model-driven engineering is the automatic generation of software artifacts from models of structure and functionality. Where the artifacts in question are at a lower level of abstraction than the models then this may be seen as a process of automatic refinement. The additional information needed is introduced by transformations that provide context or describe implementation strategies within a particular domain. This affords a significant factorisation of effort: the same transformations can be used in the development of many different systems, or many different versions of the same system.

A model-driven approach allows the developer to work at a higher level of abstraction, with concepts and structures that are closer to the users, or the processes, that the software is intended to support. The *model-driven architecture* (MDA) proposed for object-oriented development [1] has been characterised as "using modeling languages as programming languages" [2]. For such an approach to work, the concepts and structures of the modeling language must admit a precise, formal interpretation within the chosen domain, even if this is expressed only in terms of the transformation and the generated code.

A considerable amount of research has been published concerning the formal interpretation of the most widely-used object-oriented modelling language, the Unified Modeling Language (UML). However, code generation from UML models is typically limited to the production of data structures and default, primitive methods for structures such as JavaBeans [3], and the implementation of more complex, user-defined methods remains a manual task—error-prone, and time-consuming. The principal reason for this is the lack of any suitably-abstract means of describing intended behaviour: in most cases, it is easier to express and understand design intentions directly in terms of executable code.

In a sequential context, behaviour can be described in the transformational or state-based style characteristic of formal techniques such as Z [4], and the Refinement Calculus [5], and adopted in more recent developments such as the Object Constraint Language (OCL) [6]. Here, operations are specified in terms of the relationship between the state of the system before and after the operation has been performed, together with the values of any inputs and outputs. The specification is usually given as a pair of constraints: a precondition and a postcondition. The Z notation differs, notably, in regarding the precondition as a derived property, calculated as the domain of the resulting relation.

Where formal techniques are used in the design of novel programs or algorithms, the specifications may describe precisely what is to be achieved, but are unlikely to support the automatic generation of a suitable implementation. Within a specific domain, however, it is entirely possible to establish a useful set of heuristics, transformations, or strategies for translating abstract specifications into program implementations: this is formally-based, model-driven engineering in practice. For the domain of *information systems*, in particular, most postconditions are readily translated into combinations of guarded assignments: for example, the constraint that $a \in S$ could be translated to the action `S.insert(a)`.

In earlier work [7, 8], we presented a formal language for the model-driven development of information systems; we have applied this language, and the corresponding model transformation techniques, to the production of several, large systems, including a secure, online patient monitoring system. In the course of this work, it became clear that the original characterisation of the generation process in terms of data refinement, presented in [8], was problematic. It became clear also that a suitable notion of composition was required for models, in order that a large system might be designed and maintained in several parts.

In this paper, we identify a suitable notion of composition for object models in this context. We revisit our characterisation of the generation process, concluding that data refinement is an unrealistic expectation, and arriving at an improved characterisation in terms of trace refinement or partial correctness. We consider the question of when one object model might usefully refine another in this context, and the related question of when a class might usefully be defined as a subclass of another. These points are illustrated using a small example model, and placed in the context of related work.

2 Object models and abstract data types

In object-oriented programming, a class may be seen as "an implementation of an abstract data type" [9]. In object-oriented modelling, the situation is not so straightforward: the interpretation of a particular class may depend upon information presented elsewhere in the model; consideration of the class declaration itself may not be enough. For example, consider the two classes, described using the notation of UML and OCL, shown in Figure 1. The operation increment on A should increase the value of attribute m by 1; however, its applicability may be constrained by the value of n in any corresponding instance of class B.

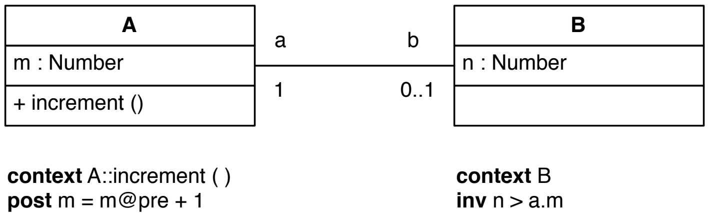


Fig. 1. A constraint between associated classes

A mapping of A and B to separate abstract data types may not admit an adequate interpretation of operation increment. Instead, the operation increment should be considered as an operation on a component whose state encompasses both A and B objects, and this component should be mapped to a single abstract data type, whose state is an indexed collection of A and B objects.

We may use the schema notation of Z [4] to describe the corresponding data type. In this description, the given set I denotes the set of object references, and $[0\,..\,1]$ a postfix, generic abbreviation for sets of cardinality at most one—in combination with the unique selection operator μ, a simple way of representing optionality. The state of the data type is described by the schema *System_State*, in combination with the local schemas *A_State* and *B_State*, and the single data type operation by the schema *A_Increment*.

$$
\begin{array}{|l}
\hline
\textit{A\_State} \\
m : \mathbb{N} \\
b : I[0\,..\,1] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{B\_State} \\
n : \mathbb{N} \\
a : I \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{System\_State} \\
as : I \pfun \textit{A\_State} \\
bs : I \pfun \textit{B\_State} \\
\hline
\forall\, b : \operatorname{ran} bs \bullet (as\ b.a).m < b.n \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{A\_Increment} \\
\Delta \textit{System\_State} \\
\textit{this} : I \\
\hline
\textit{this} \in \operatorname{dom} as \\
(as'\ \textit{this}).m = (as\ \textit{this}).m + 1 \\
\hline
\end{array}
$$

In this description, the constraint upon the applicability of the operation is captured implicitly within the global state schema. We could make it explicit by adding the conjunct $(as\ \textit{this}).b \neq \emptyset \Rightarrow (as\ \textit{this}).m < (bs\,(\mu\,((as\ \textit{this}).b))).n$ to the operation schema. The complexity of this conjunct, even in such a simple example, is representative of the difficulty posed by constraints that extend across associations.

Nevertheless, we should expect to find this kind of constraint in object models. For example, the *opposite* property for "mutually-constrained attributes" is part of the core UML language definition; the principal reference texts for OCL, including that for OCL in MDA [6], include many examples of constraints upon attributes of associated classes; and the class-responsibility-collaboration approach developed by Beck and Cunningham [10] insists that "objects do not exist in isolation" [11].

As a consequence, we should expect specifications of operations, given in the context of individual classes, to be less applicable than their precondition part would suggest. In the above example, the specification **post** m = m@pre + 1 does not tell the whole story; the given constraint cannot apply when the resulting value of m would be equal to or greater than the value of n in an associated object of class B. Although the object-oriented approach affords the convenience of defining operations within classes—within the context of the most relevant data, or the most obvious reference point—a complete understanding of the operation may require a consideration of other parts of the model.

It should be clear that the ADT corresponding to the model containing both A and B cannot be derived from the ADTs corresponding to A and B: for our implicit mapping M from models to data types, there is no function $f_{\oplus}$ such that

$$M(A \oplus B) = f_{\oplus}(M(A), M(B))$$

where $\oplus$ denotes the combination of class declarations within a model.

Observation 1 *Classes are not necessarily components in the context of model-driven development. In particular, they may not be an appropriate unit of composition for behavioural information.*

3 Model-driven development

Where the model is to be used as *source code*, as the basis for the automatic generation of a working system, the specification provided for each operation is *final*: the constraint information provided in the model is all that the compiler has to work with. In particular, then the compiler will need to determine what is to happen if the operation is called in circumstances where the constraint is not applicable: that is, for combinations of state and input values that lie outside the calculated precondition.

If the generated system holds data of any value, then it would not seem sensible to allow an arbitrary update to the state: in the absence of any default action, the effect of calling an operation outside its precondition should leave the state of the system unchanged. Further, if we wish to adopt a compositional approach, in the sense that a composite operation should be inapplicable whenever one or more of its components is inapplicable, then it is not enough for the operation to leave the state unchanged; instead, its inapplicability must be recorded or communicated.

Within the precondition, the specification is applicable, and the intended effect of the operation is known. However, it may be that the compiler does not know how to achieve this effect: that is, part of the constraint information may lie outside the domain of the model transformation rules that are used to generate the implementation. For example, the constraint

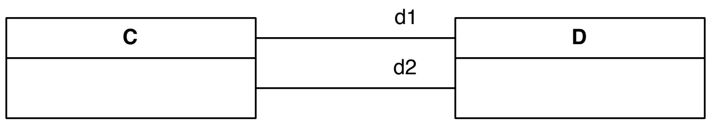
$$x = y - 1 \land y = 2x - 3$$

describes a condition achievable by the assignment $x, y := 4, 5$, but it is quite possible that the model transformations used in the compiler do not address the solution of such a system of simultaneous equations.

Where this is the case, then the intended effect of the operation is known, but is not achievable; in the generated implementation, the operation should not be allowed to proceed—unless, of course, the desired condition already holds, in which case the effect can be achieved simply by doing nothing. Again, the inapplicability of the specification should be reflected by the exceptional, or blocking, behaviour of the implementation.

In practice, we are more likely to encounter a constraint that readily admits two or more different implementations, any of which could be easily generated, but for which the intention behind the specification is unclear. That is, although an implementation could be generated that would satisfy the constraint, it seems more likely that the user would prefer to extend or qualify the specification, rather than accept—or be surprised by the behaviour of—the generated implementation.

Consider, for example, the situation illustrated by the class diagram of Figure 2, in which the operation cleanUp has the effect of ensuring that the two associations d1 and d2 are disjoint. If an object d is present in both associations when the operation cleanUp() is called, the intention in the model is unclear: should we remove d from d1 or from d2?



context C :: cleanUp ()
post d1->intersection(d2)->isEmpty()

Fig. 2. A postcondition admitting multiple implementations

While it could be the case that either alternative is equally acceptable, it is more likely that the designer has failed to make their intentions clear. Deleting one of these links may have consequences for other data: it may even be that, to achieve a new state in which the model constraints are satisified, deletions need to be propagated across the whole system. Is this what the designer intends? For information systems, where the data may be of considerable value, it may be better to generate an implementation that blocks when intentions are unclear, instead of making unexpected or unintended modifications.

The nature of refinement associated with code generation for model-driven engineering should now be clear: it is neither failures refinement, where the concurrent availability of interactions is preserved; nor is it data refinement, where sequential availability is preserved. We argue instead that it should be *trace refinement*: if the implementation is able to perform an operation, then its postcondition is achieved; however, it may be that the implementation blocks in some, or even all, circumstances where the precondition applies.

$$\begin{tikzcd}
\begin{array}{c}\textit{Model}\\ M\end{array} \arrow[r, "R_A"] \arrow[d, "T"'] & \begin{array}{c}\textit{Abstract Data Type}\\ A\end{array} \arrow[d] \\
\begin{array}{c}\textit{Implementation}\\ I\end{array} \arrow[r, "R_C"] & \begin{array}{c}\textit{Abstract Data Type}\\ C\end{array}
\end{tikzcd}$$

Fig. 3. Abstract data type semantics of model and implementation

To see this, consider the commuting diagram of Figure 3, in which T denotes the code-generating transformation, and A and C denote the representation of the model and the implementation, respectively, as abstract data types. As the model is intended as source code for system generation, it is reasonable to assume that the data model contained within it will be reflected in the implementation, to the extent that data types A and C have equivalent state components and initialisations. It is reasonable to assume also that the model and implementation present the same interface in terms of operation names, inputs, and outputs.

We will write R_A to denote the mapping from a model operation to the corresponding operation on an abstract data type, and R_C to denote a similar mapping for implementation-level operations, then the correctness constraint upon our model transformation is simply that

$$R_C \llbracket\, T\,(op)\,\rrbracket \subseteq R_A \llbracket\, op\,\rrbracket$$

for every operation op: the transformation should respect the precondition and postcondition, along with any related model constraints. There is no requirement that $\mathrm{dom}\, R_A \llbracket\, op\,\rrbracket$ should be contained within $\mathrm{dom}\, R_C \llbracket\, T\,(op)\,\rrbracket$, and hence no guarantee that C is a data refinement of A [4, 5].

However, if we consider the processes PA and PC, defined using the notation of Communicating Sequential Processes (CSP) as follows:

$$\begin{aligned}
&PA(s) = \\
&\quad \Box\, op : Op \bullet \\
&\qquad s \in \mathrm{dom}\, R_A \llbracket\, op\,\rrbracket \;\&\; \underline{op} \rightarrow \sqcap\, s' : R_A \llbracket\, op\,\rrbracket (\!|\, \{s\}\, |\!) \bullet PA(s') \\
&PC(s) = \\
&\quad \Box\, op : Op \bullet \\
&\qquad s \in \mathrm{dom}\, R_C \llbracket\, T(op)\,\rrbracket \;\&\; \underline{op} \rightarrow \sqcap\, s' : R_C \llbracket\, T(op)\,\rrbracket (\!|\, \{s\}\, |\!) \bullet PC(s')
\end{aligned}$$

where the event $\underline{op}$ represents a successful, completed execution of the operation op, chosen from the set of all operations Op defined in the model, and & denotes the guard operator: in the expression $g \;\&\; P$, the actions of process P are available only if g is true. For any code-generating model transformation T satisfying the correctness constraint above, it should be clear that

$$PA(init) \sqsubseteq_T PC(init)$$

where $init$ represents the initial state of the system, and $\sqsubseteq_T$ denotes trace refinement.

Observation 2 *The correctness of a code-generating model transformation may be characterised as trace refinement between specification and implementation: guaranteeing safety, but not liveness.*

4 Model refinement

In model-driven development, improvements are made to an implementation by updating the model used to generate it. Some updates can be characterised as formal refinements, in the sense that data type corresponding to the new model is a refinement of the data type corresponding to the old. In this case, it would be useful to know whether the old and new implementations are related in the same way: if they are, then any testing, integration, or further development based upon the generated code, rather than the model, need not be repeated. (Of course, we would prefer an approach to development in which any such activity is based solely upon the model, but this is not always possible.)

In a sequential context, the notion of model refinement that we will consider is based upon data refinement of the corresponding abstract data types. A model M_2 will refine another model M_1 precisely when the effect of any sequence of operations upon the corresponding data type A_2, in terms of the possible changes in state and outputs generated, is contained within the effect of the same sequence upon the corresponding data type A_1. The fact that such a refinement relationship exists is shown most often by exhibiting a forward simulation.

If schema s_1 denotes the set of all states—the state space—of data type A_1, s_2 denotes the state space of A_2, and i_1 and i_2 are subsets representing the initial configurations of each data type, then f is a forwards simulation precisely when $i_2 \subseteq i_1 \mathbin{⨾} f$ and

$$\begin{aligned}&\mathrm{dom}\, R_A \llbracket\, Op_1 \,\rrbracket \lhd (f \mathbin{⨾} R_A \llbracket\, Op_2 \,\rrbracket) \subseteq R_A \llbracket\, Op_1 \,\rrbracket \mathbin{⨾} f\\&\mathrm{ran}(\mathrm{dom}\, R_A \llbracket\, Op_1 \,\rrbracket \lhd f) \subseteq \mathrm{dom}\, R_A \llbracket\, Op_2 \,\rrbracket\end{aligned}$$

for every corresponding pair of operations Op_1 and Op_2, where R_A denotes the relational semantics of the operation at the model level, $\lhd$ denotes domain restriction, and $⨾$ denotes forward relational composition. This is the characterisation of [4], with the omission of the identity relation for input and output.

In the case where the two models have precisely the same classes, attributes, associations, and initialisation, this reduces to a constraint upon the updates made to the specification of each operation:

$$\begin{aligned}&\mathrm{dom}\, R_A \llbracket\, Op_1 \,\rrbracket \lhd R_A \llbracket\, Op_2 \,\rrbracket \subseteq R_A \llbracket\, Op_1 \,\rrbracket\\&\mathrm{dom}\, R_A \llbracket\, Op_1 \,\rrbracket \subseteq \mathrm{dom}\, R_A \llbracket\, Op_2 \,\rrbracket\end{aligned}$$

where Op_2 represents the updated version of Op_1. Thus we may produce a refinement of the model by weakening the precondition of an operation—that is, extending the domain of the corresponding relation—while strengthening the postcondition. To guarantee that the generated implementation is refined in the same way, we need to know also that

$$\begin{aligned}&\mathrm{dom}\, R_C \llbracket\, T(Op_1) \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket \subseteq R_C \llbracket\, T(Op_1) \,\rrbracket\\&\mathrm{dom}\, R_C \llbracket\, T(Op_1) \,\rrbracket \subseteq \mathrm{dom}\, R_C \llbracket\, T(Op_2) \,\rrbracket\end{aligned}$$

where R_C denotes the semantics of the operation at the implementation level.

The argument of the previous section tells us that any suitable code-generating model transformation T will guarantee that

$$R_C \llbracket\, T(Op) \,\rrbracket \subseteq R_A \llbracket\, Op \,\rrbracket$$

for any operation Op, but this is not enough. The following monotonicity property of T would suffice:

$$R_A \llbracket\, Op \,\rrbracket \subseteq R_A \llbracket\, Op' \,\rrbracket \Rightarrow R_C \llbracket\, T(Op) \,\rrbracket \subseteq R_C \llbracket\, T(Op') \,\rrbracket$$

for any pair of operations Op and Op'. However, this property is unlikely to hold in practice: it requires that the refinement proposed by the designer is one that is performed automatically in the course of code generation. While this would produce a model in which more of the corresponding implementation is made explicit, it seems unlikely that we would wish to propose such a refinement in the context of model-driven engineering.

To see why, consider the definitions of Op_1 and Op_2 presented as operation schemas upon a state $State \mathrel{\widehat{=}} [x : \mathbb{N}]$, with their implementations $T(Op_1)$ and $T(Op_2)$ written in an extended language of guarded commands. In a data type with state $State$, and attribute x accessible, replacing Op_1 with Op_2 would produce a data refinement.

$$\begin{array}{|l}\hline Op_1 \\ \Delta State \\ \hline x = 0 \land x' \in \{0, 1\} \\ \hline \end{array} \qquad \begin{array}{|l}\hline Op_2 \\ \Delta State \\ \hline x = 0 \land x' = 1 \\ \hline \end{array}$$

$$T(Op_1) \;\widehat{=}\; x = 0 \rightarrow \mathsf{skip} \qquad\qquad T(Op_2) \;\widehat{=}\; x = 0 \rightarrow x := 1$$

Here, T represents a plausible implementation strategy. If Op_1 is called when $x = 0$, then the subsequent value of x should be 0 or 1: the implementation $T(Op_1)$ might quite sensibly leave the value of a variable unchanged when the current value would satisfy the postcondition. In the new specification, Op_2, this nondeterminism in Op_1 has been resolved, and $T(Op_2)$ must change the value of x when x is initially zero. The data type corresponding to the second implementation is not a refinement of the one corresponding to the first.

Observation 3 *A model refinement in which postconditions are strengthened may lead to the generation of a system that is different to, and not a refinement of, the current implementation.*

There is however a circumstance in which refinement at the model level can be guaranteed to produce refinement in the implementation: when the domain or precondition of an operation is extended, but the applicable postconditions are left unchanged. Such a circumstance is quite likely to arise in the course of iterative development. Having specified an operation, a designer may find that the specification is less applicable than they had expected: that there are cases that have not been considered. If they then extend the specification to cover these cases, then they might reasonably expect that the behaviour of the implementation would remain the same for those cases already covered: that is, those within the domain of the existing specification.

For this to be the case, the following property must hold of transformation T: for any set of states S,

$$S \lhd R_A \llbracket\, Op \,\rrbracket = S \lhd R_A \llbracket\, Op' \,\rrbracket \Rightarrow S \lhd R_C \llbracket\, T(Op) \,\rrbracket = S \lhd R_C \llbracket\, T(Op') \,\rrbracket$$

That is, if Op and Op' have the same relational semantics for that region of the state space, then so do their respective implementations. For any pair of operations for which the specifications agree within the domain of the first:

$$\begin{aligned}
&\operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_A \llbracket\, Op_2 \,\rrbracket = R_A \llbracket\, Op_1 \,\rrbracket \\
&\hfill \text{[specifications agree where } Op_1 \text{ defined]} \\
&\quad \Rightarrow \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_A \llbracket\, Op_2 \,\rrbracket = \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_A \llbracket\, Op_1 \,\rrbracket \\
&\hfill \text{[domain restriction]} \\
&\quad \Rightarrow \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket = \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[property above]} \\
&\quad \Rightarrow \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket \subseteq R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[domain restriction]} \\
&\quad \Rightarrow \operatorname{dom} R_C \llbracket\, T(Op_1) \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket \subseteq R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[partial correctness of } T\text{, domain restriction]}
\end{aligned}$$

The second condition for refinement, that the domain of the operation is preserved, follows from the same condition. The correctness of transformation T guarantees that $\operatorname{dom} R_C \llbracket\, T(Op_1) \,\rrbracket \subseteq \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket$ and hence

$$\begin{aligned}
&\operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket = \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[third line of argument above]} \\
&\quad \Rightarrow \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket = R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[partial correctness of } T\text{, domain restriction]} \\
&\quad \Rightarrow \operatorname{dom}(\operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \lhd R_C \llbracket\, T(Op_2) \,\rrbracket) = \operatorname{dom} R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[property of domain operator dom]} \\
&\quad \Rightarrow \operatorname{dom} R_A \llbracket\, Op_1 \,\rrbracket \cap \operatorname{dom}(R_C \llbracket\, T(Op_2) \,\rrbracket) = \operatorname{dom} R_C \llbracket\, T(Op_1) \,\rrbracket \\
&\hfill \text{[domain restriction]} \\
&\quad \Rightarrow \operatorname{dom} R_C \llbracket\, T(Op_1) \,\rrbracket \subseteq \operatorname{dom} R_C \llbracket\, T(Op_2) \,\rrbracket
\end{aligned}$$

Observation 4 *A model refinement in which preconditions are weakened, but already-applicable postconditions are left unchanged, will produce a corresponding refinement of the implementation.*

The condition that T should produce the same implementation from Op_1 and Op_2, when restricted to the domain of Op_1, can be translated into a constraint upon the interaction of T and the grammar of our modelling language. As we suggested above, the constraints

$$x = 4 \land y = 5 \qquad \text{and} \qquad x = y - 1 \land y = 2x - 3$$

may produce the same relational semantics, but it may be that only the first of them is successfully translated into an implementation. To guarantee a refinement at the implementation level, we need to know that the constraints of Op_1 and Op_2 will be treated in the same way by transformation T.

In practice, the easiest way to ensure this is to have an operator "or" within the modelling language that corresponds to disjoint union in the relational semantics, and extend the precondition of the specification so that

$$Op_2 \mathrel{\widehat{=}} Op_1 \ \mathsf{or} \ Op_e$$

where Op_e describes the intended behaviour of the operation in circumstances left uncovered by Op_1, so that $\operatorname{dom} R_A [\![\, Op_1 \,]\!] \cap \operatorname{dom} R_A [\![\, Op_e \,]\!] = \emptyset$.

The suitability of a data refinement will depend upon our interpretation of preconditions at the implementation level. In Section 3, we argued that—where data is important—we should interpret preconditions as guards. Weakening a precondition may make an operation available in some circumstance where it is currently blocked, perhaps with good reason. For example, if the precondition for an edit() operation includes the constraint that the current value of user matches the value of owner, then weakening this condition might not constitute an improvement in the design.

The difficulty here is that we are not distinguishing between a constraint that has been included deliberately—and is intended as a restriction upon availability—and one that appears as a consequence of "underspecification". A simple solution is to include a description of the intended availability of an operation, or of a sequence of operations, as part of the model. If we treat this as part of the precondition for the purposes of code generation, then we can guarantee that it will be respected in the implementation. However, as a separate, distinguished part of the specification, it can be excluded from consideration in any subsequent, manual refinements.

The same approach allows us to address the issue of *liveness* in the implementation. Since the correctness of the code generation process is characterised as trace refinement, we have no guarantee that the implementation will do anything at all. If we have an indication A of the intended availability of an operation, or sequence of operations, then we may compare this with the precondition P of the generated implementation: if $A \neq P$, then the operation is less available than A would suggest. As the comparison involves determining the semantic equivalence of two different predicates, we would not in general be able to rely upon fully-automatic verification. However, restrictions upon the form of the specifications, coupled with the expected regularity of preconditions for implementations, should mean that automation is a perfectly feasible proposition.

Observation 5 *In the model-driven engineering of information systems, we can establish "safety properties", or partial correctness, automatically. Liveness or availability properties may require manual intervention, through semi-supervised testing or proof.*

5 Generalisation and Inheritance

In object-oriented design, a distinction is often drawn between *generalisation* and *inheritance*. For example, the UML reference manual [12] states that:

> Generalisation is a taxonomic relationship among elements. It describes what an element is. Inheritance is a mechanism for combining shared incremental descriptions to form a full description of an element. They are not the same thing, although they are closely related.

In this view, *generalisation* is a modelling concept and *inheritance* a programming concept. This begs a question: in the context of model-driven engineering, where we are using a modelling language as a programming language, which of these concepts is applicable?

If we define one class B as a specialisation of another class A—being the inverse of generalisation—then we expect everything that we know about A to remain true of B. Any class invariant should be strengthened, and so too should any operation specifications. This means that an operation specification declared in the context of A may be less applicable when considered in the context of B. As an example, consider the classes shown in Figure 4, where the operation setWidth(w:Number) should have the effect of setting the width of the current figure to w.

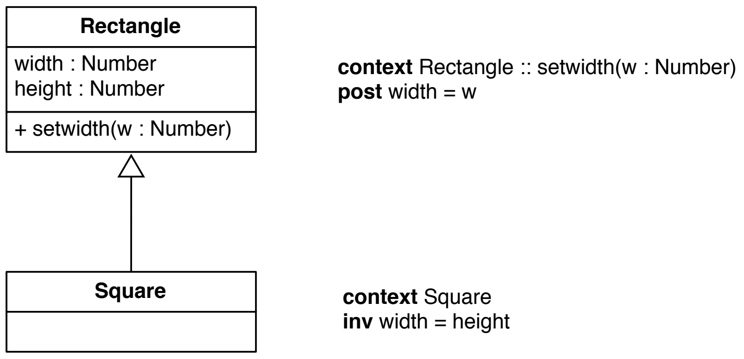


Fig. 4. Square and Rectangle

We would argue that the most appropriate strategy for code generation, in the face of such a specification, is to produce an assignment to the attribute width and—in the context of Rectangle—to leave the value of all other attributes unchanged. Any other approach might come as something of a surprise to the user, and thus reduce the utility of the notation as a programming language. However, when we consider setWidth in the context of Square, we find that it has the implicit precondition w = height.

It should be immediately apparent that we cannot simply use an instance of Square whenever we might have used an instance of Rectangle: a setWidth operation in which the width is set to anything other than the current height would fail for an instance of Square, when it would have succeeded for an instance of Rectangle. Whether this amounts to a violation of the substitivity condition of Liskov and Wing [13]—

> Let $P(x)$ be a property provable about objects x of type T. Then $P(y)$ should be true for objects y of type S where S is a subtype of T.

—depends upon the notion of properties involved. Certainly, if setWidth is performed on an instance of Square, then we know at least as much about the states of the object before and after the operation as we would if it had been performed on an instance of Rectangle. However, if we were to consider the availability of the operation as a property of interest, then the condition would indeed be violated.

As was the case with code-generating model transformations, the fact that preconditions are strengthened rather than weakened means that specialisation may not characterised simply as data refinement. Instead, a more appropriate characterisation may be that of trace refinement, under the assumption that no new operations are introduced—or, at least, that no new operations are introduced that may update the state in such a way as to affect the availability or effect of one or more of the existing operations.

Observation 6 *In the context of model-driven development, specialisation need not correspond to subtyping in the programming sense; for transformational specifications, trace refinement may be a more appropriate notion.*

Of course, in the design of an object-oriented program, the relationship of Figure 4 may well have been reversed. The class Rectangle might have been introduced as an extension of Square, with the addition of a distinct *height* property. Consider the two fragments of Java code shown below

```
class Square{
  float width;
  float getArea(){ ... }
}
```

```
class Rectangle extends Square{
  float height;
  float getArea(){ ... }
}
```

Here, `Rectangle` will inherit the attributes of class `Square`, and will redefine the method `getArea()`—to use both `width` and `height`. This form of inheritance represents code re-use, and it is certainly possible to exhibit specifications in which Rectangle is a refinement of Square, in terms of corresponding abstract data types. For example, we might imagine a specification for `Square.getArea()` that stated

post (self.oclIsKindOf(Rectangle) and result = width $*$ height)
or
(self.oclIsKindOf(Square) and result = width $*$ width)

If `Rectangle.getArea()` were assigned the same specification, or merely the first disjunct above, then we could exhibit a forward simulation between the two corresponding data types.

Such specifications might be produced in the course of a post-hoc activity in which existing Java code is annotated with specifications that take account of any inheritance hierarchy. However, in a model-driven context, our purpose in supplying specifications is to enable the generation of an implementation: a suitable specification for `Square.getArea()` would be **post** $\mathsf{result} = \mathsf{width} * \mathsf{width}$, and one for `Rectangle.getArea()` would be **post** $\mathsf{result} = \mathsf{width} * \mathsf{height}$. These specifications would not produce a forward simulation, or a data refinement, in the corresponding abstract data types.

Furthermore, in model-driven development the emphasis is upon specification re-use, rather than the specification of code re-use. In this example, the potential for re-use is in the other direction: Square should be seen as a specialisation of Rectangle, inheriting the constraints as additional conjuncts alongside any new specification provided. This will also support the expected refinement relationship—trace refinement—between classes in an inheritance hierarchy.

Observation 7 *In model-driven development, re-use is afforded by specialisation rather than inheritance.*

6 Discussion

In this paper, we have argued that classes are not a suitable basis for behavioural composition in the context of model-driven engineering. For the purposes of code generation, a component is a closed collection of associated classes: *closed* in the sense that every constraint refers only to attributes declared in classes within the collection. This applies whether the component is implemented as a separate system, communicating by means of a messaging protocol, or whether it is used to generate part of the applications programming interface for a larger system.

We have argued also that the notion of correctness associated with code generation, and with specification re-use, should be that of trace refinement. This reflects our understanding that the model transformations used to generate the code may not be able to resolve all of the nondeterminism within the specifications supplied: either because the specification 'problem' cannot be solved, or because it is unclear which of the possible solutions corresponds to the intentions of the designer. Our strategy for the verification of liveness properties is to provide a separate constraint specifying the intended availability of a given operation, or a given sequence of operations. This can be compared with the generated guard or availability constraint in the implementation: if it is stronger, then we may wish to modify the model and repeat the generation process.

Finally, we have argued that refinements to the model need not correspond to refinements of the implementation. We identified a necessary and sufficient condition for this to be the case, but argued that this would be an unrealistic objective in practice. We then identified a necessary condition that would be eminently achievable in the iterative development of operation specifications.

If the existing description is correct, but fails to address all of the relevant combinations of state and input values, then we may extend the specification with an appropriate alternation operator, and repeat the generation process, without the need to repeat any testing or development based on the previous, generated implementation.

The question of whether classes can be treated as components has been addressed before, although not in this context. Most relevant is the work of Szyperski [14], who characterises a *component* as a unit of independent deployment and of composition, with no externally observable state, and an *object* as a unit of instantiation that may have an externally observable state. If we interpret 'observability' as the ability to refer to an attribute in an externally-declared constraint, then his argument that components are collections of classes, rather than individual classes, is in line with that presented in this paper. In the area of formal techniques, Barnett and Naumann [15] come to the same conclusion about association constraints in 'real-life situations', and provide a mechanism for working with collections of 'cooperating classes'.

The formal technique *Object-Z* [16] allows the definition of object references, and constraints can mention attributes of other classes. Such *object coupling* induces additional conditions upon the constraint information in a model in order to achieve individual class refinement [5]: in object-oriented design, as we have argued, these conditions may not be fulfilled. Some authors [17, 18] rule out such constraints, insisting that read access to attributes is through accessor methods only, or aligning preconditions of component methods by introducing derived input attributes. A similar approach is taken in *OhCircus* [19]. The result is a semantics that aligns closely with that of CSP: each class is a separate process, with no externally observable state.

CSP-OZ [20] and TCOZ [21], building upon earlier work on action systems [22], allow the definition of a separate guard, as well as a precondition, for each operation. In these methods, the guard and the precondition together define the operation. Our approach is different in that the user-supplied precondition is treated as an upper bound on availability: if it does not hold, then the operation should be blocked. In the code generation process, our preconditions are treated as (partial) guards. There is value in providing a second, separate piece of information, analogous to a guard, that corresponds to a *lower bound* upon the intended availability: a liveness constraint. This could be used as the basis for the generation of a suite of tests, or as a property to be checked using a theorem prover.

Our objective is to add useful, formal support for model-driven, object-oriented development. In doing so, we have identified a need for two different notions of composition: one in which classes are combined to produce a complete description of a sequential component, and one in which sequential components are combined to produce a complete working system. In this paper, we have focussed our attention on the first of these, where the notion of composition is purely static: the behaviours of the sequential component cannot be derived from the behaviours of the individual classes; instead they emerge as the result of the combination of classes and associations.

References

1. Kleppe, A., Warmer, J., Bast, W.: MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
2. Frankel, D.: Model Driven Architecture: applying MDA to enterprise computing. OMG Series. Wiley (2003)
3. Matena, V., Stearns, B., Demichiel, L.: Applying Enterprise JavaBeans: Component-Based Development for J2EE. Pearson (2003)
4. Woodcock, J., Davies, J.: Using Z. Prentice Hall (1996) www.usingz.com.
5. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: foundations and advanced applications. Springer (2001)
6. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley (2003)
7. Faitelson, D., Welch, J., Davies, J.: From predicates to programs. In: Proceedings of SBMF 2005. Volume 184. (2007)
8. Davies, J., Faitelson, D., Welch, J.: Domain-specific semantics and data refinement of object models. ENTCS **195** (2008)
9. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (2000)
10. Beck, K., Cunningham, W.: A laboratory for teaching object oriented thinking. SIGPLAN Not. **24**(10) (September 1989)
11. Wirfs-Brock, R.: Responsibility-driven design. *The Smalltalk Report* (1991)
12. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley Professional (2004)
13. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16**(6) (1994)
14. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley (2002)
15. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC, volume 3125 of LNCS, Springer (2004) 54–84
16. Smith, G.: The Object-Z Specification Language. Kluwer (2000)
17. McComb, T., Smith, G.: Compositional class refinement in Object-Z. In: Proceedings of FM 2006. LNCS, Springer (2006)
18. Smith, G.: A fully abstract semantics of classes for Object-Z. Formal Aspects of Computing **7** (1995)
19. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software and Systems Modeling **4** (2005)
20. Fischer, C.: How to combine Z with a process algebra. In: Proceedings of ZUM 98. Volume 1493 of LNCS. Springer (1998)
21. Mahony, B., Dong, J.S.: Blending Object-Z and Timed CSP: An introduction to TCOZ. In: Proceedings of ICSE98, IEEE Press (1998)
22. Back, R.J.R., von Wright, J.: Trace refinement of action systems. In: Structured Programming, Springer-Verlag (1994)