Piecewise Modelling with State Subtypes

Friedrich Steimann and Thomas Kühne

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Fachgebiet Metamodellierung
Technische Universität Darmstadt
D-64289 Darmstadt
kuehne@informatik.tu-darmstadt.de

Abstract. Models addressing both structure and behaviour of a system are usually quite complex. Much of the complexity is caused by the necessity to distinguish between different cases, such as legal vs. illegal constellations of objects, typical vs. rare scenarios, and normal vs. exceptional flows of control. The result is an explosion of cases causing large and deeply nested case analyses. While those based on the kinds of objects involved can be tackled with standard dynamic dispatch, possibilities for differentiations based on the state of objects have not yet been considered for modelling. We show how the handling of class and state-induced distinctions can be unified under a common subtyping scheme, and how this scheme allows the simplification of models by splitting them into piecewise definitions. Using a running example, we demonstrate the potential of our approach and explain how it serves the consistent integration of static and dynamic specifications.

1 Introduction

In the age of model-driven development, models are required to specify both structure and behaviour of a system. Although there will always be the case for coarse abstractions omitting many particulars, a large class of models must address the specification of detail. Such models have to deal with considerable complexity introduced by the necessity to distinguish between many cases of alternative collaboration and control flow. Particular problems present cases of undefinedness, i.e., illegal constellations of objects or method invocations with which no reasonable behaviour can be associated. Excluding such cases through many explicit conditions disrupts the primary concern of a model, the depiction of what is right and what should be done. As a result, models are more difficult to write, to read, and to maintain than they should be.

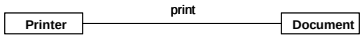
In this paper, we suggest to use subtyping and overloading of relations (where relations include associations, attributes, and operations) as a means for structuring a domain into defined and undefined cases, and to provide modular, piecewise definitions for the defined cases. Our approach shares some similarity with multi-dispatching known from programming languages, but significantly extends it with the possibility to take the dynamic state of objects into account. By doing so, our approach not only allows simpler models, it also improves the integration of static and dynamic specifications that were previously thought to be rather isolated.

In the remainder of this paper we first we show how the declaration of relations is used as an essential ingredient of type-level specifications, and how systematic over-

loading can be used to refine the information conveyed (Section 2). Following the concept of multi-dispatch, we show how attaching definitions to different (overloaded) branches of a declaration can eliminate the need for certain types of case distinctions (Section 3). We extend this idea to cover the state of objects, by introducing state subtypes and overloading relation declarations on these (Section 4). Furthermore, we show how state subtypes can be automatically derived from the statecharts associated with classes, and how in general statecharts can be integrated with structure and sequence diagrams. In Section 5, we elaborate process issues associated with our approach. We then present a discussion of the limitations of our approach and a comparison with related work (Section 6), and conclude with Section 7.

2 Modelling with Declarations

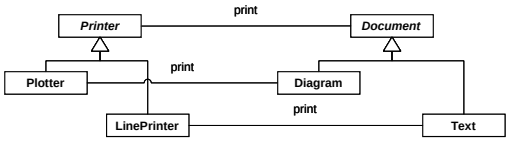
In object-oriented modelling, declarations are an accepted form of specification on the type level. For instance, an excerpt from a static structure (class) diagram such as



expresses that documents and printers can engage in a relation[1] named *print*. Its textual equivalent is the declaration of the signature of a relation:

$$print: Printer \times Document \quad (1)$$

Following the general understanding in programming (which suggests that the types in a declaration must be substitutable by all of their subtypes), such a declaration is usually (mis)interpreted as a statement of total definedness, i.e., it is assumed that printing is defined for all combinations of printers and documents. Looking at the problem domain more closely, however, one notices that there are such different entities as diagrams and texts, as well as line printers and plotters, and that texts are printed only on line printers, while diagrams are printed only on plotters. This refinement can be modelled by *overloading* the declaration of *print*, either graphically by



or textually by

$$print: Plotter \times Diagram \quad (2)$$
$$print: LinePrinter \times Text$$

However, there is no way to explicitly declare that

$$\neg print: LinePrinter \times Diagram \quad (3)$$

Instead, declaration (1) seems to warrant the printability of diagrams on line printers, a misunderstanding that results from the erroneous interpretation of *declarations* as *definitions*. In fact, in mathematics, a declaration such as (1) expresses that

[1] We deliberately speak of relations here and not of associations, since following [20] we take relations to cover associations, attributes, and methods.

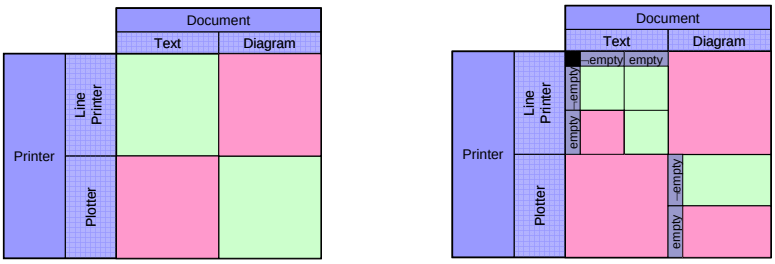


Fig. 1. (a) Definition holes (dark – or red – squares) induced by undefinedness of certain class combinations. (b) Additional holes imposed by dynamic conditions. The relation's domain is irregular and can only be declared piecewisely; besides, it cannot be declared using static classification (classes) alone.

$$print \subseteq Printer \times Document \tag{4}$$

i.e., that the extension of *print* is a subset of the Cartesian product of the domains of its places (which subset exactly typically being subject to further definition). The Cartesian product $Printer \times Document$ provides merely an upper bound, which is too high in our example since as mentioned above there are combinations of documents and printers that will *never* engage in the *print* relation. In fact, least upper bounds are usually non Cartesian, as illustrated by Fig. 1(a).

Moreover, in modelling and programming we have an additional temporal dimension, which means that the extension of *print* grows and shrinks with time. So how are the declarations (1) and (2) to be interpreted? Generally, we assume that more specific declarations (i.e., declarations of the same relation, but involving subtypes) are intended to overrule the more general ones. In fact, we go as far as requiring that only *minimal overloadings* of a declaration (i.e., overloadings such that there is no other overloading that involves only subtypes of the first) may have tuples. This allows us to define relations whose domain is not Cartesian, but rather a hypercube with (hypercubic) holes (*cf.* Fig. 1), with each tuple of the relation binding to a minimal overloading (not necessarily precisely one; *cf.* the discussion in Section 6.4). In Sections 3 and 4, we will attach specifications[2] to minimal overloadings and show how they represent *branches of a piecewise definition of the relation* within the model. One might be tempted to conclude that any tuple for which a minimal declaration exists is defined – however, as argued in Section 6.1 this is not necessarily the case.

Returning to the meaning of the declarations (1) and (2): they now bound the extension of *print* by the union of the Cartesian products of its minimal declarations:

$$print \subseteq (Plotter \times Drawing) \cup (LinePrinter \times Text) \tag{5}$$

Since relations include attributes (see Footnote 1) we can furthermore write

[2] A more appropriate term would be "implementation", but this word is problematic in the context of modelling. "Definition" is another alternative; however, this would make our definition of definedness appear circular.

$$myPrinter: Document \rightarrow Printer \quad (6)$$

in order to declare an attribute *myPrinter* with value type *Printer* for *Document* as a special relation where the arrow separates the last argument of the relation (the value) from the rest. Again, this declaration must not be interpreted as statement of total definedness, since (6) can be overloaded as in (2), e.g., with the branches

$$\begin{aligned} myPrinter: Diagram &\rightarrow Plotter \\ myPrinter: Text &\rightarrow LinePrinter \end{aligned} \quad (7)$$

defining the range of the attribute as being dependent on its domain (*cf.* the discussion of dependent types in Section 6.4).

3 Attaching Definitions to Declarations

To illustrate how piecewise definitions based on overloaded declarations can simplify modelling, we extend our printing example by first adding a print manager that prints documents on printers. Second, we let texts consist of pages and let them be printed page by page, in contrast to diagrams, which are printed on a single page each. Last but not least, a printer can run out of paper in which case all printing attempts fail (i.e., are undefined), the only exception being the printing of an empty text (i.e., a text that has no pages). Later, we will be confronted with a niggling user who is dissatisfied with the undefinedness of the out-of-paper situation; fortunately, thanks to our piecewise definition approach pleasing him will turn out to be easy.

The static structure of our domain is modelled by the class diagram in Fig. 2. *PrintManager* is a singleton that acts as a façade to the printing module. Its sole purpose is to accept printing requests and forward them to the printer. *Printer* and *Document* are linked by an association *print*, which represents the same *print* relation as expressed by the operation *print(Document)* in class *Printer*, but shows additionally how it is overloaded. Since *Printer* and *Document* are both abstract, the *print* relation must recruit its elements (tuples) from its concrete subtypes. Following our binding rules from Section 2, missing combinations (e.g., *Text* × *Plotter*) are undefined.

The sequence of actions required to process a printing request is shown in the sequence diagram of Fig. 3. Although the overall behaviour is rather simple, the many case distinctions make the diagram appear complex. Note that the branches depend on the type of the arguments of the print relation (as expressed by the *instanceof* operator) and on the state of the individual involved objects (whether or not the printer is

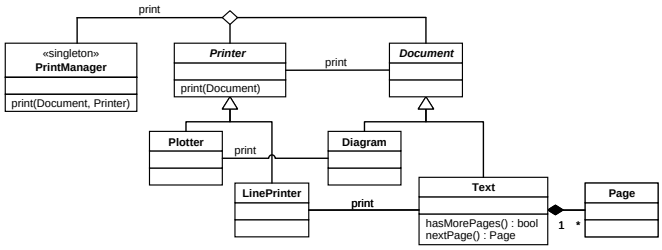


Fig. 2. Static structure of the printing example.

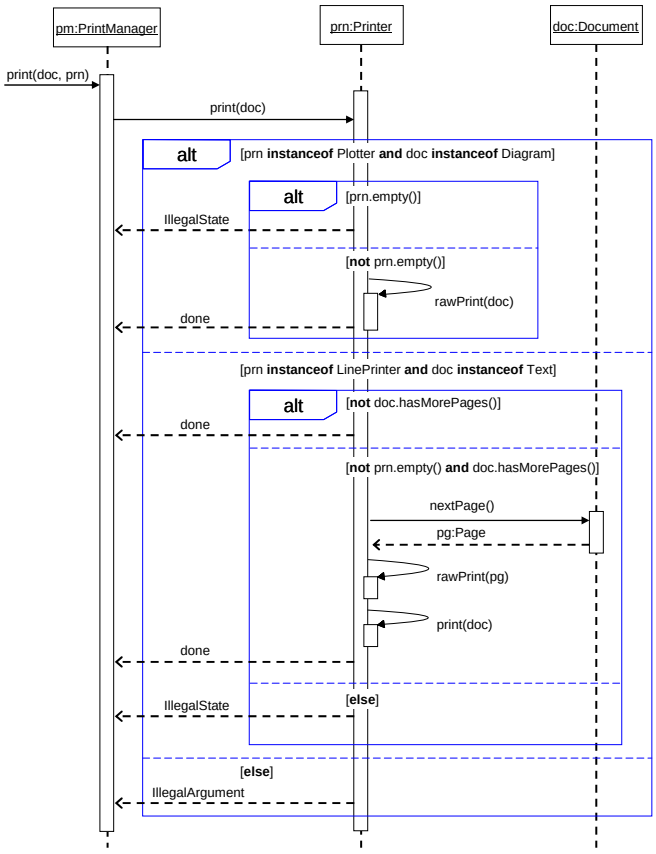


Fig. 3. UML sequence diagram handling printing requests for arbitrary printers and documents.

empty, whether or not the text has more pages). With growing detail in the modelled scenarios the number of special cases needed to be considered steadily increases, quickly leading to a combinatorial explosion. Since nested branches are known to be extremely error-prone (both in programming and in modelling), significant improvements can be expected from any modelling construct that does away with them.

This is where our overloaded declarations come into play. They allow us to split the sequence diagram of Fig. 3 into the two pieces shown in Fig. 4, one for each admissible combination of documents and printers. A message *print*(*doc*, *prn*) sent to a print manager *pm* is then bound to one of the two diagrams, or rejected as undefined. We thus have separated *defined* from *undefined* cases and provided *alternative definitions* depending on the *types* of arguments.

4 Introduction of State Subtypes

The sequence diagrams of Fig. 4 still contains undesirable case analyses, but this time, the distinctions are induced by the *states* of the involved objects. If we could capture the states of the objects with corresponding (sub)types, we could use the same

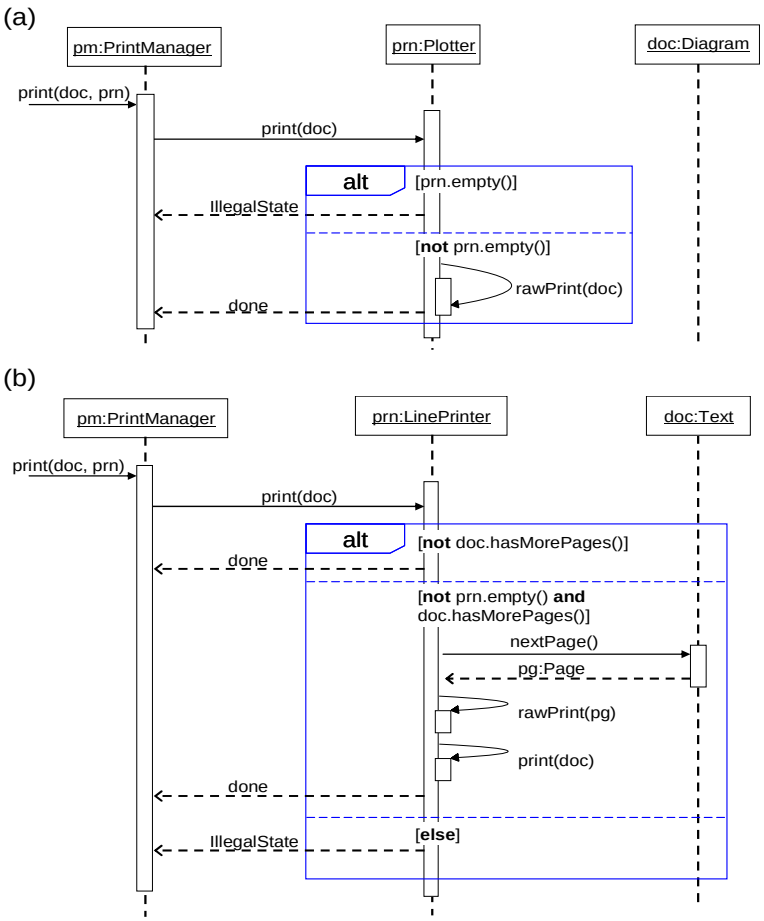


Fig. 4. Same specification as that of Fig. 2, with type-based branching replaced by binding.

technique as before, namely overloading and piecewise definition, to further reduce the complexity of each sequence diagram. Assuming that we declare the state subtypes and the new minimal branches of *print* as shown in Fig. 5, we can extend Fig. 1(a) to 1(b) and replace the sequence diagrams of Fig. 4 with those of Fig. 6. Note that each object may only be in one state at a time; hence we have marked the classes that have state subtypes (*Plotter*, *LinePrinter*, and *Text*) as abstract.

If a state subtype does not engage in an overloading then this subtype contributes no elements to the relation. For instance, the absence of an overloading involving *Plotter|empty* and *Diagram* expresses that diagrams cannot be printed on empty plotters. Note that we use a vertical bar to create a state subtype's name from the name of the class it subtypes and the name of the state.

The diagram in Fig. 6 (c) raises an important issue. Because the state types of *doc* and *prn* may change after each printed page, the types specified in the lifelines heads

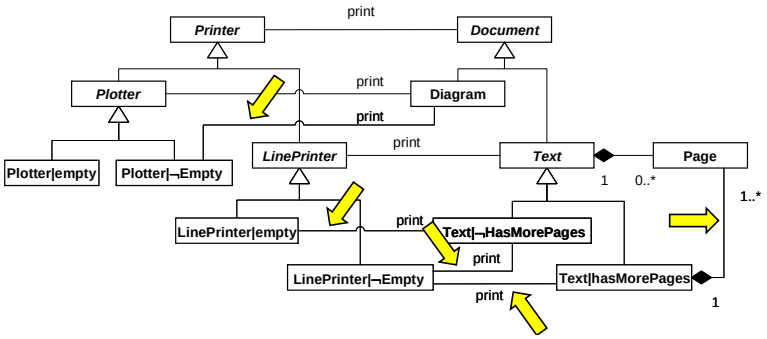


Fig. 5. Addition of state subtypes and corresponding overloadings (marked by arrows). Note the missing link from *Plotter|empty* to *Diagram*. Also, the multiplicity of the aggregation between *Text* and *Page* has been restricted for *Text|hasMorePages*, while *Text|¬hasMorePages* does not relate to *Page*.

may no longer be valid. However, subsequent relation tuples (i.e., method sends, including the recursive *print*(*prn*, *doc*)) will then *bind to branches based on the new state subtypes*. This allows us to elegantly model the printing of pages until either the printer is empty or all pages of the text have been printed, where all the control logic is implicit in the binding of message sends. Not inserting a new binding after possible state changes would require explicit tests and branching (in our example, a loop with explicit loop conditions), which are still possible but cumbersome.

Note that the change in state could be reflected by using a state invariant on the lifeline of the object, a feature of UML 2.0 [15]. As we will see below, possible state changes can be automatically derived from statecharts modelling the state transitions of objects.

4.1 Definition of State Subtypes

By definition, subtypes add to the intension of their supertypes: they add properties to and pose additional constraints on their elements. Hence, the extension of a subtype is always a subset of that of its supertypes, implying that relations that are only partially defined on supertypes may be totally defined on subtypes.

We define a *state subtype* as a subtype of a class that collects all objects of that class that are in a certain state. A state subtype adds to the intension of a class by restricting the range of attribute values, and by restricting the associations[3] and methods its objects can engage in. Although state subtypes need not generally be mutually exclusive, those that are (because they are generated from the same statechart; see below) provide a complete partition of the class's extension (*cf*. also Fig. 1(b)).

State subtypes allow piecewise total definitions of otherwise partial relations using subtyping and overloading, even if definedness depends on dynamic conditions. For instance, the relation *print* from above is totally defined for all objects of state subtype

[3] Indeed, the fact that an object plays a certain role (sits in the place of a relationship) can be considered (part of) its state.

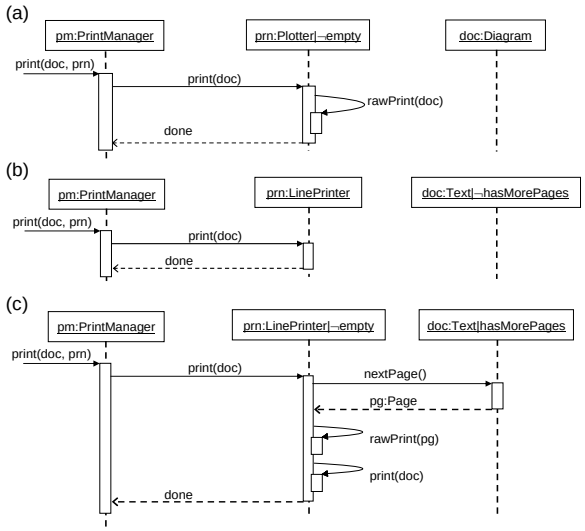


Fig. 6. Further simplification of behaviour specification made possible by the introduction of state subtypes. Note that Figs. 5 and 6 are equivalent to Figs. 2 and 3.

Plotter|¬*empty* (but not for all objects of class *Plotter*[4]). Like the subclasses from Fig. 2, state subtypes may serve to express definition holes by overloading relations.

4.2 Integration with Statecharts

One might argue that the static structure diagram of Fig. 5 has taken over much of the complexity removed from the sequence diagram of Fig. 3. Particularly larger examples may cause such structure diagrams to quickly become unwieldy. Fortunately, the latter can be automatically generated from much simpler diagrams.

The key to only specify a regular class diagram and obtain the additional information contained in Fig. 5 for free is to exploit supplementary statecharts. Fig. 7 shows three simple statecharts, describing the behaviour of the objects of classes *Plotter*, *LinePrinter*, and *Text* respectively. Note that the states from each statechart partition the dynamic extension of the class it is associated with since each object of a class must be in exactly one state at a time. Furthermore, the events of Fig. 7 code the definedness of relations in each state: they correspond to the minimal overloaded declarations of Fig. 5 as well as to the method signatures found in Fig. 6. This allows us to *automatically* derive the state subtypes and overloadings of *print* in Fig. 5 from Fig. 7. In addition, the transitions of Fig. 7 specify which possible state changes need to be

[4] At first glance, our notion of state subtypes seems to contradict the idea of subtyping, because objects of a (state-) subtype can no longer substitute for those of its supertype. However, we remind the reader that the (assumed) all-quantification of a declaration (cf. Section 2) is untenable anyway: a stack cannot be popped if it is empty, no matter what the declaration of the class promises. Since at any point in time every instance of a class is also an instance of one of its state subtypes, the notion of substitutability can only involve those properties that are independent of state.

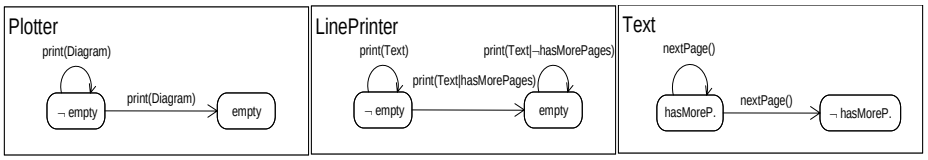


Fig. 7. Statecharts for classes *Plotter*, *LinePrinter*, and *Text*. States correspond to the state subtypes in Fig. 4, events correspond to the overloaded declarations in Fig. 4 and to the methods in Fig. 5; their absence determines which method is undefined for which state. Note that UML's semantics for statecharts is different: events for which no transitions are shown are ignored.

considered in a sequence diagram, e.g., that the state subtype of both a text and a line printer may change after printing each page (*cf.* section 4). Last but not least, the sequence diagram specifies the interaction between objects, i.e., it binds the different statecharts together by showing which events in one statechart lead to which events in another (*print* of *LinePrinter* leads to *nextPage* of *Text*). It follows that state subtypes are natural pivotal points for the integration of static and dynamic specifications.

5 Process Issues

Model Evolution. Modelling is an iterative, incremental process, and the usability of modelling languages that do not support this development style is severely limited. Modelling with piecewise definitions based on subtypes and overloading supports incremental development rather well, as the following considerations suggest.

When modelling, we must distinguish two different kinds of undefinedness: *natural undefinedness* and *as-yet undefinedness* (*as-yet* with respect to the progress of the modelling process). An example of the former is the printing of a diagram on a line printer, and example of the latter is the printing of a document on an empty printer. While natural undefinedness will persist right to the final version of the model, as-yet undefinedness is usually removed while the model progresses: it is a form of "temporary omission". In terms of supporting model evolution, the question is how much rearrangement of the model is required to remove such temporary omissions.

In the case of printing on an empty printer, the corresponding refactoring is trivial: all that needs to be added is a branch that lets the printer wait until paper is refilled; no interaction with other objects is required. Note how this added behaviour also fixes

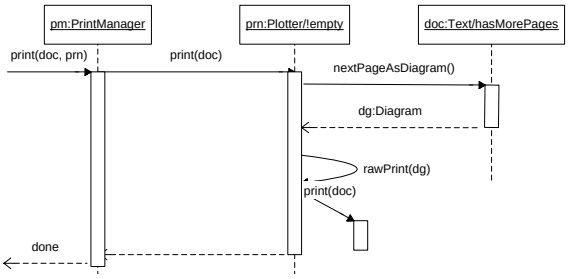


Fig. 8. Added behavior specification for printing texts on plotters. No interference with other specifications must be considered.

the undefinedness problem of a text having more pages than the printer has paper.

Also the decision that texts cannot be printed on plotters can be easily revoked. Fig. 8 shows the corresponding new collaboration which can be added without interfering with the rest of the model. These "low impact changes" are possible because we extend the definition of relations rather than that of single classes, thereby allowing *a form of refinement that is automatically coordinated among classes*. Our approach thus fulfils the old promise of object-orientation, namely to be able to *refine models locally through subtyping*.

Tool Support. Just as programming today is unthinkable without programming environments, the quality of models is an increasing function of the quality of modelling tools used. However, this is only true if a modelling language allows tools to contribute to the quality of models. In particular, consistency and completeness checks should be supported; otherwise tools are reduced to mere drawing aids.

Fortunately, our framework offers a wealth of opportunities for tool support. Computing the coverage of minimal declarations of the domain spanned by the most general declaration of a relation shows potential definition holes, which can then be marked by the modeller as either natural or as as-yet undefined. Overlapping of minimal declarations can also be flagged: for instance, if default behaviour for all empty printers is added as suggested above, the resulting overlap with the specification for printing empty texts (*cf*. Fig. 1(b)) can be discovered. In fact, we envision a relationship browser that presents the definition of relations in a form similar to that of Fig. 1, collecting all piecewise definitions and allowing their quick access and editing. Also, consistency of statecharts with associated structure and sequence diagrams can be automatically checked, and changes in one diagram may propagate directly to changes in the others. Last but no least, the state transitions of statecharts can be used to check whether possible state changes are adequately accounted for in the sequence diagrams. This may include the computation of sets of operations that are admissible in all possible post-states of a previous operation.

6 Discussion

6.1 Total and Partial Relations, and Error Propagation

At first glance it appears that our piecewise definition approach is capable of turning all partially defined relations into totally defined ones, by separating out the undefined cases. However, this need not always be the case: even though the minimal branch *print*: *LinePrinter*|¬*empty* × *Text*|*hasMorePages* appears to be OK, printing text on a non-empty line printer is in fact undefined if the text has more pages than the printer has sheets of paper (see Fig. 6). Unfortunately we cannot solve this problem by introducing a finite number of new state subtypes, since the number of pages is generally not limited. In fact, total definedness of a branch is granted if and only if

1. its specification does not rely on other branches, or
2. it involves only branches that are themselves totally defined.

In other words, the partiality of a branch automatically propagates to all specifications that depend on it. While this is a feature of our approach (because it frees the modeller

from dealing with partiality explicitly), it is also a problem, because undefinedness can creep into seemingly innocuous parts of a model.

We address this problem by making definition holes explicit, for instance by adding exceptions to the signature of relations. These exceptions could be interpreted as the names of sets of tuples that are to be subtracted from the domain of a relation. For instance, we could write

$$print: LinePrinter|\neg empty \times Text|hasMorePages \setminus TooManyPages \quad \textbf{(8)}$$

in order to denote that the set labelled *TooManyPages* must be subtracted from the domain of *print*. This set would contain all pairs of texts and printers of which the text has more pages than the printer has paper denotes; its specification could either be left implicit, or tied to the fact that the printing results in an empty printer with pages left to print. Relations that depend on *print* will inherit the exception, as for instance the ternary version of *print* that is associated with the print manager (see Fig. 3). We can unify this explicit form of error propagation with the implicit one (the inability to bind a tuple to a branch) since the latter can be automatically translated to explicit exceptions.

6.2 Open Problems

Although we could demonstrate some appealing properties, our approach certainly deserves further elaboration. For instance, in order to eliminate all explicit branches of a model, large numbers of small specifications will have to be provided. While this is considered good practice in object-oriented programming, it may lead to models that are difficult to trace (the negative impact dynamic binding may have on program understanding applies accordingly). Also, many of the opportunities for model integration suggested here depend on simple diagram languages. Given the complexity of UML's current statechart specification, it would be naïve to believe that integration with sequence diagrams can remain as simple as shown here. Another problem we did not touch on is that a single class can have several statecharts attached, and that the resulting state subtypes need not be unrelated, so that possible interdependencies between statecharts must be considered (for instance through attributes whose range is constrained by states of more than one statechart).

6.3 Possible Improvements

Many basic states such as *empty* occur over and over, as do the transitions linking them. Rather than specifying one statechart for each class separately, one could envision the definition of "abstract" statecharts that are "implemented" by various classes, each possibly adding variations. For instance, the statecharts for *LinePrinter* and *Plotter* in Fig. 7 are sufficiently similar to think about deriving them from a common generalization. However, a corresponding investigation is beyond the scope of this paper.

The notion of state subtypes and piecewise definition could also be extended to other diagram types, for instance activity diagrams. As the latter – together with sequence diagrams – are a popular means to formalize use cases, this immediately points out a path to piecewise definitions of use cases as well. Note that the "extends" relationship between use cases may already be considered as an existing way to

piecewisely specify exceptional or alternative behaviour, clearly demonstrating the need for modularity in use case specifications.

Last but not least, an automatic translator of models into declarations of a statically typed language with (checked) exceptions (such as Java) could be devised. This translator may automatically add *illegal argument* and *illegal state* exceptions as shown in Fig. 3 to method signatures, and translate subtype information of overloaded declarations to preconditions firing these exceptions. For more sophistically typed languages, the generation of dependent types should also prove to be a fairly simple exercise.

6.4 Related Work

Interestingly, the UML standard does not even attempt to specify binding rules: "The dispatching method by which a particular behaviour is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point)." [15] It appears that specification of binding is deliberately left to the target programming language selected for the project. The consequences of such a policy have been discussed in [1]; we add that UML still lacks an explicit notion of overloading (cf. the discussion in [19]).

Avoiding case analyses caused by state-induced behaviour differentiation is also the goal of the *State* pattern [8]. Its application might be a viable alternative in the context of programming (e.g., with Java) but models should not contain explicit realization structures that are designed to address lacking language support. Applications of the *State* (addressing state-based dispatch) or the *Visitor* pattern (addressing multi-dispatch) would introduce realization structures to models which are not induced by an original to be described or a system to be specified.

Our approach has some similarity to order-sorted algebraic specifications [11], in particular the universal order-sorted algebras independently developed by Gogolla and others [10, 13]. This form of algebraic specification assumes that each operation symbol represents a single operation defined on a universe of individuals, with sorts corresponding to subsets of that universe. In this framework, overloaded operations whose domains overlap must be identical on the overlap (because they denote the same operation). This feature has particular appeal to us, since it saves the modeller from having to deal with such things as overriding and calls to *super*, programming constructs that are believed to increase the compactness of specifications, but really introduce a lot of problems. In fact, our requirement that specifications can only be attached to minimal declarations should be seen as a first step towards congruent relation specifications (as it can flag possible inconsistencies, *cf.* Section 5). Conditions required so that all tuples bind to one minimal declaration unambiguously (corresponding to the regularity of signatures simplifying the implementation of overloaded order-sorted algebras [10]) have not been formulated explicitly here, because they would have required a more formal exposition which we sacrificed in favour of a more readable description of our work.

Also in the context of algebraic specifications, Gogolla et al. have introduced the distinction between *unsafe* and *OK* functions, the former of which may lead to errors [8]. Terms containing unsafe functions are themselves unsafe; in analogy to our relations that are specified in terms of other, partial relations. Interestingly, the partiality of operations and the propagation of errors remain hard problems for algebraic speci-

fications [13]; given the analogies pointed out above, universal order-sorted algebras with exceptions can provide a nice formal semantics of our approach.

Shang shares our view that is inadequate to interpret declarations as statements generally all-quantified over their parameters [18]. For instance, a declaration *Animal.eat*(*Food*) (where *Animal* and *Food* are supertypes) should not be read as "all animals eat all food". He notes that what he calls *component types* (types of fields or of parameters to methods) are sometimes dependent on the type of the enclosing object (e.g., the specific kind of food is dependent on the specific kind of animal), and that declarations on (abstract) supertypes should explicitly express this dependency. Piecewise definitions of relations as suggested above cover the idea of dependent types, but are more general: in particular, they do not require a statement of which parameter type depends on which (which would be somewhat arbitrary in our printing example). Also, dependent types do not account for state-induced case analyses.

Castagna has argued the case for overloading by way of covariant redefinition and multiple dispatching in object-oriented programming [2]. In his λ&-calculus, he defines a message as a set of methods, where each method defines a branch of a function distinguished by the types of its parameters. Castagna's branches roughly correspond to our piecewise definitions, and his multi-dispatch to our binding rules. Castagna allows contravariance of method parameters to ensure substitutability for parameters on which no dispatching is desired; by contrast, we prefer to treat all parameters equally. Like Shang, Castagna does not consider dynamic typing.

Based on work by Ernst et al. [6], Millstein has extended Java with "predicate dispatch", i.e., a generalization of method dispatch that not only includes the types of parameters (as in multi-dispatch), but also potentially their values [12]. The emphasis in this work is on creating a modular static type system and the automatic detection of ambiguous method definitions. For this purpose, Millstein considers structural types, integers, and Boolean values; although dispatch on state subtypes could be emulated, such is not explicitly addressed.

Chambers introduced the concept of predicate classes (corresponding to our state subtypes) to the programming language CECIL and demonstrates their utility with a number of examples [3], thereby validating the concept in a programming language context. Our work differs from and goes beyond the work by Chambers by suggesting the applicability of state subtypes in the context of modelling and by using piecewise definitions for a number of diagram types ranging from sequence diagrams to use case diagrams. Working in a modelling context we can draw on the existence of statecharts associated to classes and *automatically* derive state subtypes from them, thus advancing the integration of static and dynamic diagram types. We furthermore show how to unify the treatment of class and state-based behaviour with undefined behaviour and hint at potential support by modelling tools.

Fickle$_{II}$ is a proposed extension to the static type system of object-oriented programming languages like Java that allows the type-safe, dynamic reclassification of aliased objects stored in temporary variables (including *this*) [5]. It uses state subtypes that extend so-called root types (classes) by adding and overriding members. The state type of an object can be changed at runtime, so that subsequent dispatches of same members on same variables may yield different results. Since different state subtypes of the same root can have different members, the contents of attributes that are not

common to all state subtypes are dropped upon state change. In contrast, our state subtypes can only restrict the ranges of attributes and the applicability of operations.

Salzman and Aldrich have also suggested removing explicit branching by multi-dispatch based on the state of objects [17]. However, they use prototypes rather than state subtypes for this, and do not cater for undefined constellations as we do, but instead require that methods are provided for all possible parameter constellations, thus requiring explicit handling of undefinedness. Regarding natural "definition holes", their approach is not as expressive as dependent types, or our work.

Nierstrasz defines "regular types" by specifying (or at least approximating) their protocol through non-deterministic finite state machines [14]. Thanks to the existence of an equivalence test for this category of automata, he can check whether one type is a behavioural subtype of another type, with respect to their specified protocols. Although Niersrasz's work would be helpful for dealing with the inheritance of statecharts (*cf.* Section 6.3), unfortunately we cannot directly draw on it since we must assume more powerful automata types in general.

Similar to Nierstrasz, Paech and Rumpe specify rules for behavioural subtyping relationships but see their relevance predominantly in refining existing types in an incremental model development process [16]. Similar to our discussion of covariantly redefined attribute values in Section 2, Paech and Rumpe also constrain the attribute values of objects, depending on which state (of an automaton) the object is in.

DeLine and Fähndrich introduce type states to object-oriented programming in order to make statements about object states through the help of a modular, static type system [4]. They use type states in pre- and postconditions, for instance, to guarantee the error free execution of methods. As a result, state transitions are scattered over methods and – in contrast to our explicit state diagrams – the corresponding state transition graph is only implicitly defined. DeLine and Fähndrich thoroughly investigate state subtypes in the context of inheritance between classes, but do not consider the extension of dynamic binding to include the state (-types) of objects.

The type system of ALLOY [7] computes for every model two kinds of types, called bounding types and relevance types. Bounding types restrict the possible set of values of expressions from above, whereas relevance types approximate the sets of elements that make a difference in the evaluation of an expression in a given context; contrary to our approach, they are a derived, rather than a declared, property. Empty relevance types flag modelling errors (since no object can make a difference). Non-empty relevance types on the other hand do not indicate total definedness of an expression; in fact, ALLOY has no explicit relation declarations, and undefined expressions evaluate to the empty set (ALLOY has no exceptions). Interestingly, ALLOY also allows overloading and interprets it as union of relations; in addition, it resolves all non-abstract supertypes to abstract supertypes with one additional, (implicit) subtype containing the remainder. Issues of openness (and modularity) of a model are not addressed.

7 Conclusion

While much work remains to be done, we believe to have shown that state subtypes can significantly reduce the complexity of models. In particular, in combination with overloading they allow the definition of models in a piecewise fashion, thus avoiding

explicit case analyses. Hence state subtypes support incremental model development, allowing modellers to easily address temporary omissions by simply adding new local definitions, leaving the rest of the model unchanged. Our presented framework furthermore enables a unified treatment of special type-based, state-based, and undefined behaviour. Finally, state subtypes turn out to be natural pivotal points for the integration of static and dynamic specifications.

References

1. A Beugnard “Is MDA achievable without a proper definition of late binding?” *UML Workshop in Software Model Engineering (WiSME)* (2002).
2. G Castagna “Covariance and contravariance: conflict without a cause” *ACM TOPLAS* 17:3 (1995) 431–447.
3. C Chambers “Predicate Classes” in: *Proceedings of the European Conference on Object-oriented Programming* (ECOOP '93) LNCS 707 (1993) 268–296.
4. R DeLine, M Fähnrich “Typestates for Objects” in: *Proceedings of the European Conference on Object-oriented Programming* (ECOOP '04) LNCS 3086 (2004) 465–490.
5. S Drossopoulou, F Damiani, M Dezani-Ciancaglini, P Giannini “More dynamic object reclassification: Fickle$_{II}$” *ACM Trans. Program. Lang. Syst.* 24:2 (2002) 153–191.
6. M Ernst, C Kaplan, C Chambers “Predicate Dispatching: A Unified Theory of Dispatch” in: *Proceedings of the European Conference on Object-oriented Programming* (ECOOP '98) LNCS 1445 (1998) 186–211
7. J Edwards, D Jackson, E Torlak “A type system for object models” in: *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering* (2004) 189–199.
8. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1995)
9. M Gogolla, K Drosten, UW Lipeck, HD Ehrich “Algebraic and Operational Semantics of Specifications Allowing Exceptions and Errors” *Theor. Comput. Sci.* 34 (1984) 289–313.
10. JA Goguen, R Diaconescu “An Oxford Survey of Order Sorted Algebra” *Mathematical Structures in Computer Science* 4:3 (1994) 363–392.
11. J Meseguer, JA Goguen “Order-sorted algebra solves the constructor-selector, multiple representation, and coercion problems” *Information and Computation* 103:1 (1993) 114–158.
12. T Milstein “Practical Predicate Dispatch” in: *Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA) (2004) 345–364.
13. PD Mosses “The Use of Sorts in Algebraic Specifications” in: *COMPASS/ADT* (1991) 66–92.
14. O Nierstrasz “Regular types for active objects” in: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1993) 1–15.
15. OMG *Unified Modeling Language: Superstructure* Version 2.0 (2005).
16. B Paech, B Rumpe “A new concept of refinement used for behaviour modelling with automata” in: *2nd Int. Symp. of Formal Methods Europe* Springer LNCS 873 (1994) 154–174.
17. L Salzman, J Aldrich “Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model” in: *ECOOP* (2005) 312–336.
18. DL Shang “Covariant deep subtyping reconsidered” *SIGPLAN Notices* 30:5 (1995) 21–28.
19. F Steimann “A radical revision of UML's role concept”, in: A Evans, S Kent, B Selic (eds) *UML 2000: Proceedings of the 3rd International Conference* (Springer, 2000) 194–209.
20. F Steimann, T Kühne “A radical reduction of UML’s core semantics” in: *UML 2002: Proceedings of the 5th International Conference* (Springer, 2002) 34–48.