A new Concept of Refinement used for Behaviour Modelling with Automata

Barbara Paech, Bernhard Rumpe

Fakultät für Informatik,
Technische Universität München
D-80290 München
email: paech,rumpe@informatik.tu-muenchen.de

Abstract. This paper introduces a new approach of using automata to model behaviour of objects. Automata allow to design a software model on the abstract level of states and transitions. We make precise the meaning of states and transitions in the context of objects and types. This formal semantics serves as a link between informal and formal software development methods. Second, we give a formal, but nevertheless intuitive definition of automata specialization which not only provides a way for reuse of type definitions in subtypes, but also shows how to incrementally design types through refinement. Third we define the notion of a role an object takes when viewed from the environment. Then we can give a formal, but short proof that our definitions satisfy the subtype requirement, that means that an object of the subtype plays the roles of its supertypes.

1 Motivation

Object oriented analysis and design (OOA/OOD) results in software models capturing static and dynamic aspects of a system. Types, consisting of attributes and operations, and relationships between types, in particular inheritance, constitute the static part. Dynamic aspects are captured in the behaviour of types. On the level of object oriented programming languages (OOP) the behaviour of a type is determined through the code of the operations and possibly some synchronization conditions [Fro92, Mes93]. The latter determine the circumstances under which the operations are enabled. During the analysis phase the code is not available, so another behaviour description is neccessary. Many authors [RBP+91, MO93] use some kind of automata as a very intuitive means of describing operation sequences. However, the relationship between the (labels of the) automata and the types is not made clear, and similarly the relationship between the automaton of a type and the automata of its subtypes is not made precise. On the other hand several formal approaches to the *specification* of object systems exist, e.g. [JSHS91, DDP93, LW93]. Specifications capture behaviour separated from its implementation. They can be used to give an abstract, but *complete* account of the behaviour formulated as the *outcome* of the analysis phase. During the analysis *process*, however, one needs to be able to work with *incomplete* behaviour descriptions which are easy to extend.

In this paper we will show that *incomplete* behaviour descriptions need not be *imprecise*. Instead, incompleteness imposes some kind of abstraction, describing only certain aspects of an information system. Therefore we will enhance automata by a few formal details leading to the concept of *behaviour automata*. Behaviour automata can serve as a link between informal and formal software design methods. On the one hand they are easy to read and to use informally. Therefore they are an adequate base for the discussions between software designers and customers in the process of designing the system model. On the other hand the formal semantics leaves no room for ambiguities. Thus, we can define a formal notion of *refinement* which shows how to *safely* enhance the system model. This notion of refinement coincides with the notion of type specialization (typesafe inheritance). This is not surprising, since specialization can be viewed as a preservation of a refinement step for reuse.

This paper is structured as follows: First we define behaviour automata as a means of behaviour description of types. The main contribution of this definition is to make explicit the *state space* of a type. The state space of a type is the set of all states an object of that type may occupy in its lifecycle. As usual an object state is given by the values of its attributes. The labelling of the states of the behaviour automata induces a partition on the state space of the type. The transitions of the behaviour automata are labelled by operations (and possibly conditions). Behaviour automata allow to grasp quickly the sequence of operations objects of a given type may perform, and also the important states these objects may pass.

In the second part we define specialization of behaviour automata. This notion of specialization allows for reduction of the state space, refined partition of the state space and reduction of nondeterminism. First we motivate this definition by discussing several examples of type specialization (inheritance). We also show that this specialization relation is transitive. Then we discuss refinement steps based on the specialization relation.

In the third part we show that our definition of type specialization satisfies the *subtype requirement* as formulated in [LW93], namely that "the subtype's objects must behave 'the same' as the supertype's objects as far as anyone using the supertype's objects can tell". The expectations of the using objects are made explicit in the concept of *role* which are a special kind of type. The behaviour automaton of a type has to specialize all the automata of its roles. Because of transitivity of automata specialization it is trivial to show the above subtype requirement.

We conclude the paper with a discussion of related work.

2 Behaviour Automata

Our work is independent of a specific notation for the static aspects of types. In Figure 1 an example is given for the essential parts of such a notation.

Type `Figure` describes geometric figures typically used in graphical editors. The term 'type' is mostly used in the context of specifications, while 'class' is

Type	Figure
Attributes	center position(x,y-coordinate), color, outline, visibility(true,false), selection(true,false)
Operations	move(new center), show, hide, select, deselect

Fig. 1. Type Figure

used in OOA and OOP. We use the term 'type' to indicate that in our framework inheritance is behaviour preserving. The type definition consists of the name of the type and a set of attributes and operations. Attributes are defined by a name and an optional value space (another type or an informal description). Operations are given by names and a possibly empty list of parameters. Relationships to other types - other than specialization - are captured by attributes. The attributes in the type definition do not have to be implemented as attributes. E.g. attributes depending on other attributes could be implemented as *derived functions* [Mey92]. However, during the analysis phase it is natural to think in terms of attributes. We distinguish between *visible* and *hidden* attributes and operations. The client (or user) of an object can see only the visible attributes, via *selection operations* that do not affect the object's state. Visible operations constitute the *interface* of the type together with the selection operations. Hidden attributes and operations describe internal details.

A type defines a set of objects. Each object has a unique identity and a *state*. The state is defined by the values of the attributes. The application of the operations to the objects depends on the state. The *behaviour of an object* is its lifecycle, a unique sequence of states and operation calls. The *behaviour of a type* is the set of possible lifecycles of its instances. Note that in OOP the objects' lifecycles are completely determined through the code of the operations and the synchronization constraints. This is due to the fact that the state of an object can only be modified through operations and that the semantics of the operations is completely determined. The latter is not true for OOA. The behaviour of an object is incompletely specified. Behaviour automata allow to restrict lifecycles without having to define them completely. They only define the *minimal* behaviour of objects, namely some operations, the states in which objects are certain to respond to calls of these operations and a rough description of the effect of the operations.

2.1 Automata States

Automata states partition the set of allowed object states through constraints on their attributes. As an example consider the type `Automatic Teller Machine` of Figure 2.

Type Automatic Teller Machine
Attributes **Visible** output ("Insert Card!", "Password?", "What please?")
Hidden password(undefined or digits),
account number (undefined or digits), cash (natural number)
Operations **Visible** input card(account number), check password (password),
fill(amount), withdraw(amount), deposit(amount), transfer,
cancel
Behaviour

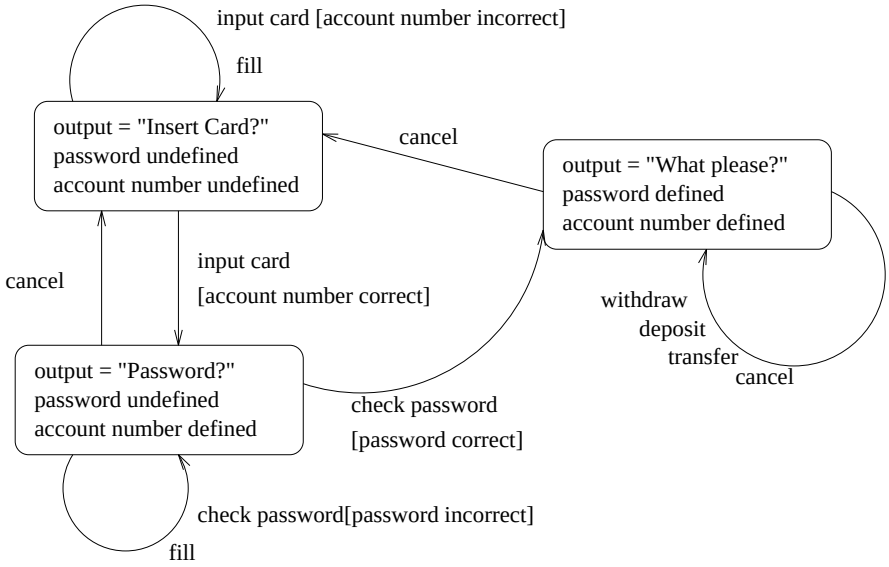

Fig. 2. Type Automatic Teller Machine

Its attributes are `password, account number, cash` and `output`, where only the latter is visible. We abstract from the money given to or from the teller machine. Possible constraints are *password undefined, account number defined.* Many methods [RBP+91] characterize automata states by names instead of constraints. Introducing a special attribute *status* ranging over all the possible names of the states one can model the name X as the constraint $status = X$. This way, however, the dependencies between the attributes are not made explicit. In the automatic teller machine example (Figure 2) the constraint *account number undefined, password defined* is not meaningful. This cannot be expressed through names.

2.2 Transitions

The state of an object can only be modified by operations defined in the type description (*encapsulation*). Thus only operations can cause transitions between automata states. Pre- and postconditions of the operations which concern only attributes are captured by the automata states. Additionally the automata states show how the operations interact. Preconditions which concern the parameters are not covered through automata states but through additional constraints on the transitions. A typical example is checking the password for access to the automatic teller machine (see Figure 2). The *transition constraint* is made explicit in brackets [,] following the operation name.

The transition relation is not required to be deterministic. Therefore it is also possible to leave the dependency of the successor state from the parameters unspecified (e.g. operation `get` of type `Queue` in Figure 3). A missing constraint is equivalent to the constraint `true`.

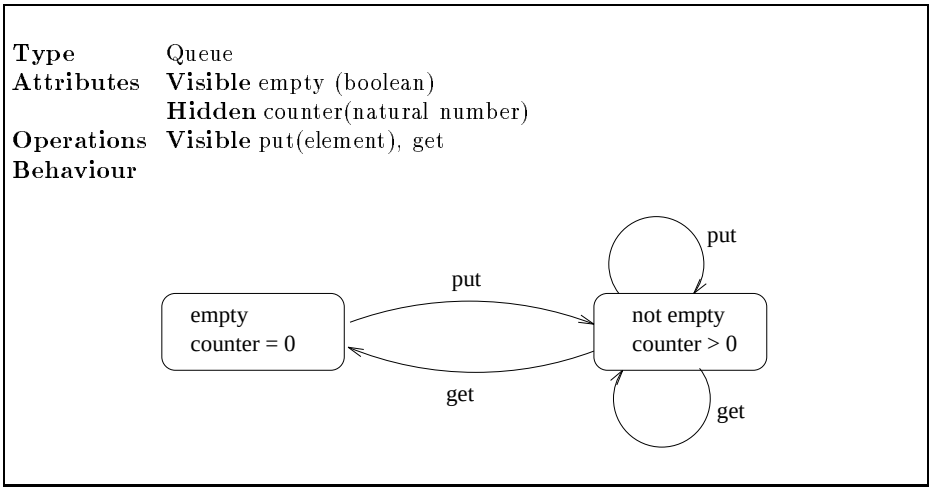

Fig. 3. Type Queue

Clearly, operations not only modify the internal state but also invoke operations on other objects (including operations of *self*). This interaction between types can be captured by listing the invoked operations together with the transitions (see e.g.[GM93]). We will not pursue this topic here.

We close this section with a list of the graphical elements representing behaviour automata in Figure 4.

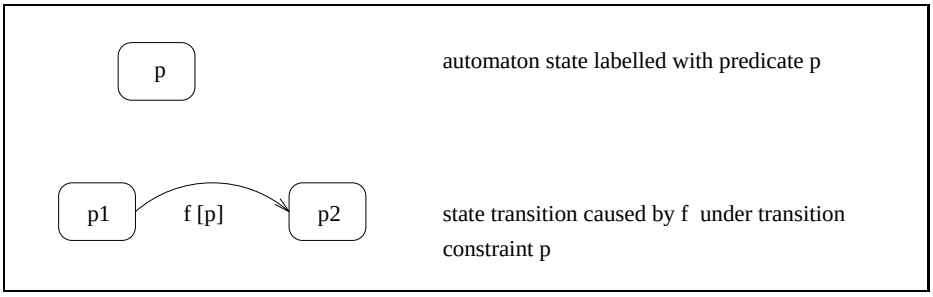

Fig. 4. Graphical Representation of Automata

2.3 Formal Definition of Behaviour Automata

There are two important restrictions on the labelling of the automata states and the transitions.

- As noted above automata states induce a partition. Therefore the predicates labelling the automata states have to be exclusive.
- The transition constraints capture the case analysis on the parameters. This analysis has to be exhaustive. Because of nondeterminism we allow multiple transitions labelled with the same operation and overlapping transition constraints.

For readability reasons a constraint common to all automata states can be factorized as an *invariant*. All states of all objects of the type have to satisfy this invariant. For the moment we only allow non-temporal invariants talking about single states. We expect it to be easy to extend our work to more powerful invariants as in [LW93, JSHS91, DDP93]. The invariant of the automata need not be maximal in the sense that every condition implied by the labels of all the automata states is implied by the invariant. The reason is that sometimes it might be difficult to check for this implication.

In figure 5 the concept of behaviour automaton is made precise.

Because the invariant is not required to be maximal the state space is characterized by $I \wedge \bigvee_{s \in S} L(s)$. Automata states whose label contradicts the invariant are possible, however, they do not correspond to any object state.

2.4 Formal Definition of Types

Behaviour automata are intended to define the behaviour of a type. This is only possible, if they satisfy certain syntactic restrictions. In the following definition of a type $Var[x]$ denotes the set of free variables of the term $x \in \mathcal{L}$.

Definition of Behaviour Automata:

Let $\mathcal{L}$ be a first-order language, , $\models$ denote validity and $\mathcal{OP}$ be a set of names.
Then $Lab = \{f[p] : f \in \mathcal{OP}, p \in \mathcal{L}\}$ is the set of *transition constraints*.
Let S be a nonempty, but finite set, $\Delta \subseteq S \times Lab \times S$ a finite relation, $I \in \mathcal{L}$ and $L : S \to \mathcal{L}$ a function. For $s \in S, f \in \mathcal{OP}$ the set

$$enabled(s, f) = \{p \in \mathcal{L} : \Delta(s, f[p], t) \text{ holds for some } t \in S\}$$

contains the transition constraints for f in state s. $\mathcal{A} = (S, \Delta, L, I)$ is called a *behaviour automaton* with *automata states* S, *transition relation* Δ, *state labelling* L and *invariant* I, if

A1. the state labels are exclusive:
for all $s_1 \neq s_2 \in S$ holds $\models (I \wedge L(s_1) \wedge L(s_2)) \Leftrightarrow$ `false`.
A2. transition constraints are exhaustive:
for all $s \in S, f \in \mathcal{OP}$ such that $enabled(s, f) \neq \emptyset$ holds
$\models (L(s) \wedge I) \Rightarrow \bigvee enabled(s, f)$.
(Note that a missing transition constraint is per default `true`.)

Fig. 5. Definition of Behaviour Automata

Definition of a Type:

Let $\mathcal{L}$ be a first-order language, $\mathcal{V}$ be the set of its free variables, $\mathcal{OP}$ be a set of names.
A *type* is given by $(name, Att, VAtt, Op, VOp, par, \mathcal{A}, s_0)$, where *name* is the name of the type, $Att \subseteq \mathcal{V}$ are the names of the attributes, $VAtt \subseteq Att$ denotes the visible attribute subset, $Op \subseteq \mathcal{OP}$ are the operation names, $VOp \subseteq Op$ denotes the visible operation subset, $par : Op \to Set(Par)$ is a function associating parameter names $Par \subseteq \mathcal{V}$ to operations with $Att \cap Par = \emptyset$, $\mathcal{A} = (S, \Delta, L, I)$ is a behaviour automaton over $\mathcal{L}$ and $\mathcal{OP}$ and $s_0 \in S$ is the initial state such that

T1. state labels and invariants are restrictions on attributes only:
$Var[I] \subseteq Att$ and for all $s \in S$ holds $Var[L(s)] \subseteq Att$.
T2. only operations label the transitions:
for all $s, t \in S, f[p] \in Lab$ with $\Delta(s, f[p], t)$ holds $f \in Op$
T3. the transition constraints are restrictions on parameters and attributes:
for all $s, t \in S, f[p] \in Lab$ with $\Delta(s, f[p], t)$ holds $Var[p] \subseteq Att \cup par(f)$

Note that not every operation of the type has to appear as label of the automaton. If not, nothing is required of its behaviour. The selection operations corresponding to visible attributes are not shown in the automaton, since they cannot modify the state. They are only relevant to the interface of the type.

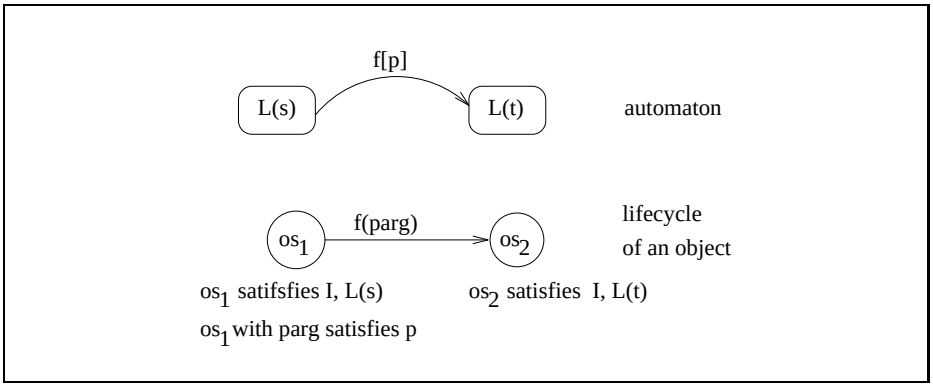

Fig. 6. Lifecycles and Automata

The state space of a type is formalized as follows:
Let U be the *universe* of all values and object identifiers, including the undefined value $\perp$. The *state space of a type* $T = (name, Att, VAtt, Op, VOp, par, \mathcal{A}, s_0)$ is characterized by

$$OS(T) = \{os \in [Att \rightarrow U] : \perp \notin os(Att) \wedge (os \models I \wedge \bigvee_{s \in S} L(s))\},$$

where $os \models p$ denotes the fact that the predicate $p \in \mathcal{L}$ is true under the valuation induced by os. A type definition $T = (name, Att, VAtt, Op, VOp, par, \mathcal{A}, s_0)$ induces the following requirement on the implementation of an operation $f \in Op$ (see also Figure 6):

Let $os_1 \in OS(T)$ be an object state, $s \in S$ with $os_1 \models I \wedge L(s)$. Let $parg \in [par(f) \rightarrow U]$ be a parameter valuation. The possible destination states of f are given by

$$D_f(parg, os_1, s) = \{t \in S : \text{there exists } p \in \mathcal{L} \text{ such that } parg \cup os_1 \models p \text{ and } \Delta(s, f[p], t)\}.$$

If $D_f(parg, os, s)$ is not empty, the application of f with parameter valuation $parg$ in state os_1 results in state $os_2 \in OS(T)$ such that $os_2 \models I \wedge L(t)$ for some $t \in D_f(parg, os_1, s)$.

From this semantics it is easy to see that the behaviour description by automata is very close to the *rely/guarantee*-style of behaviour specification [Jon83]. Assuming that the environment calls $f(parg)$ in an object state os_1 satisfying $I \wedge L(s)$ for an automata state s, the object guarantees the successor state os_2 to satisfy $I \wedge \bigvee_{t \in D_f(parg, os_1, s)} L(t)$.

More formally, for every operation f define the rely/guarantee pair (R_f, G_f) (written as $R_f \Rightarrow G_f$) as follows:

$$R_f(os_1) \equiv \quad (os_1 \models def_f), \text{ and}$$
$$G_f(os_1, os_2, parg) \equiv \quad (os_2 = apply(f, parg, os_1) \Rightarrow$$
$$\exists s, t \in S, p \in \mathcal{L}.\ (parg \cup os_1 \models p) \wedge (os_1 \models L(s)) \wedge$$
$$(os_2 \models L(t)) \wedge \Delta(s, f[p], t)),$$

where $def_f = I \wedge \bigvee \{L(s) : enabled(s, f) \neq \emptyset\}$.

Then for all $f \in \mathcal{OP}, os_1, os_2 \in OS(T), parg \in [par(f) \rightarrow U]$ holds

$$R_f(os_1) \Rightarrow G_f(os_1, os_2, parg).$$

Note that, since we allow state labels to be contradictory (to the invariant), the above guarantee condition might not be satisfiable for an operation f. Clearly, this has to be checked for at some point during the development process. However, incorporating this test into the definition above, in our view, constitutes too strong a restriction to be useful in the analysis process.

Altogether, the semantics of behaviour automata can be summarized as follows: the automata states together with the invariant characterize the *maximal* set of states an object of the type can assume. For every transition a minimal set of enabling states and a maximal set of successor states to these enabling states is given.

3 Specialization

The specialization relation between types is an important means of structuring the static aspects of the system. Coming from OOP at first specialization allowed arbitrary reuse of code. In the meantime several specialization relations have been proposed preserving some kind of behaviour, e.g. [LW93, Mes93, Fro92]. In the following we introduce the concept of *behaviour automata specialization*. In section 5 we show how it relates to other definitions of the specialization relation between types.

Subtypes inherit all attributes and operations of the supertype as well as the invariant. Visible attributes and operations remain visible. To motivate further conditions on the specialization relation we discuss typical instances of specialization. Consider the type `Figure` and its subtypes depicted in Figure 7 (the definitions of the subtypes exhibit only the additional attributes, operations and invariants, all attributes and operations are visible).

Addition of an Invariant

This corresponds to a restriction of the state space. See for example the specialization of type `Ellipse` to type `Circle` through the addition of the invariant `xradius=yradius`. The operations of the subtype can take advantage of the additional invariant, e.g. for efficiency reasons. For example, the operation `rotate` will be trivial for type `Circle`, since by the invariant $xradius = yradius$ a circle is preserved by rotation.

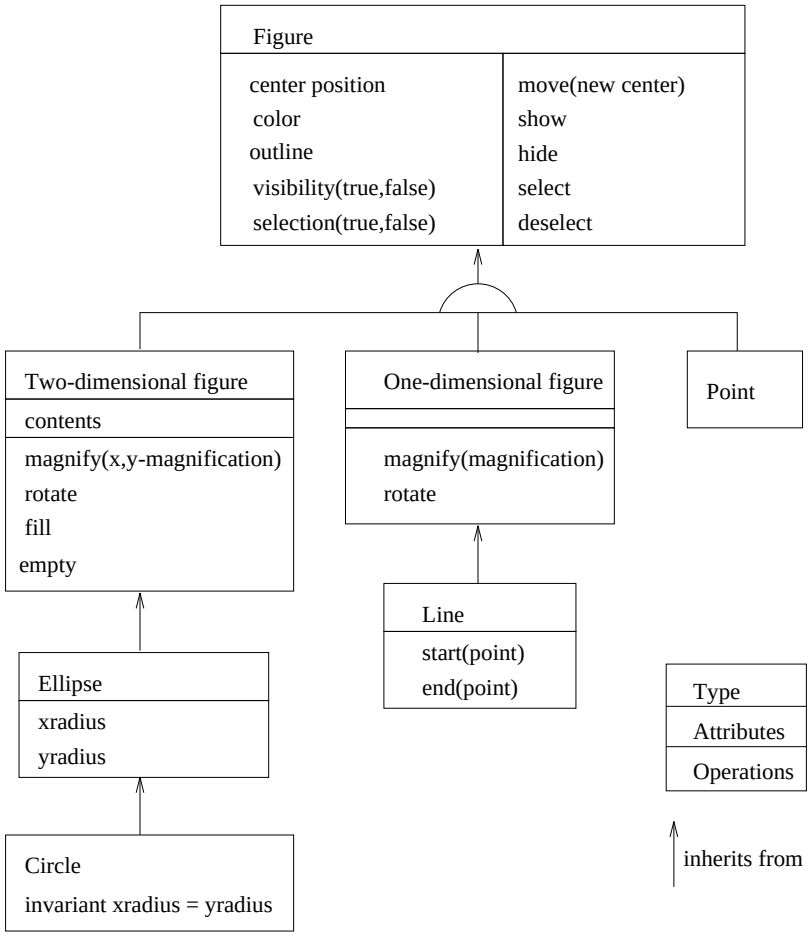

Fig. 7. Typehierarchy for Figure

Addition of one or more Attributes

The state space is *constrained* through the new attribute[1]. The addition of the attribute x can also be considered as the addition of an invariant "x exists".

In its most simple case the addition is orthogonal preserving the behaviour of inherited operations. For an example consider the specialization of

[1] Often addition of an attribute is considered an extension instead of a restriction. Note, however, that the state space of a type contains all objects with at least the attributes of the type. Therefore addition of an attribute restricts the state space.

`Two-dimensional Figure` to `Ellipse` by addition of the attributes `xradius`, `yradius`.

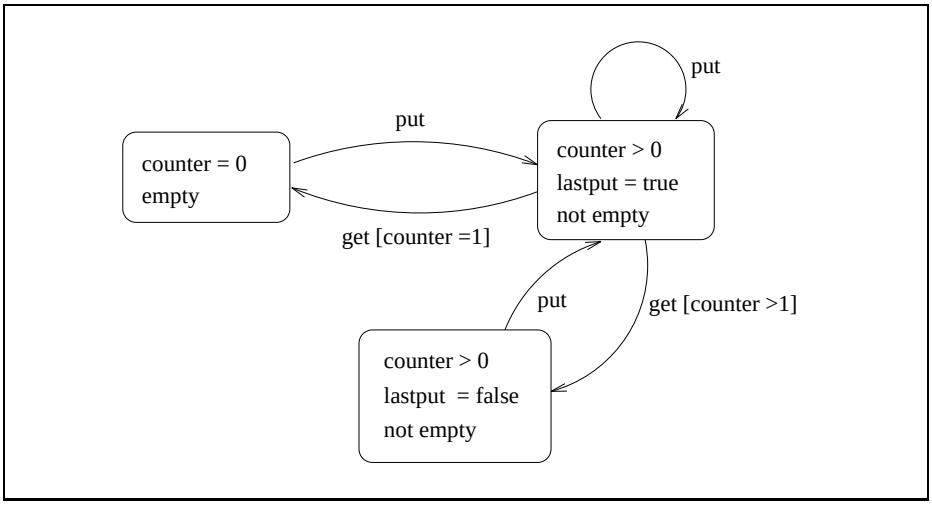

Fig. 8. Behaviour Automaton of Type Bad Queue

We do not allow arbitrary modification of inherited operations by a treatment of the new attribute. Consider the type `Bad Queue` adding attribute `lastput` to type `Queue`. Figure 8 shows the behaviour automaton of type `Bad Queue`. The operation `get` is only activated in `Bad Queue`, if `lastput` is true (that means the last operation was `put`). Contrary to type `Queue` operation `get` is not activated in all states satisfying $counter > 0$. This is considered too strong a modification to be summarized under specialization.

Addition of Operations

The preconditions of the new operations give rise to a refined partition of the state space. An example is type `Two-dimensional Figure`, where operations `fill` und `empty` have been added. `Fill` is only activated, if $content = false$, `Empty` is only activated, if $content = true$. The corresponding behaviour automaton results from splitting the state $visibility = true, selection = true$. Compare Figures 9 and 10, where we use a Statechart-like notation [Har87][2].

Addition of operations induces the restriction that the implementation of the new operations may not invalidate the inherited invariant.

[2] Here we use this notation only as a shorthand to cope with the state explosion. We are not concerned with the real-time semantics underlying the framework of Statecharts.

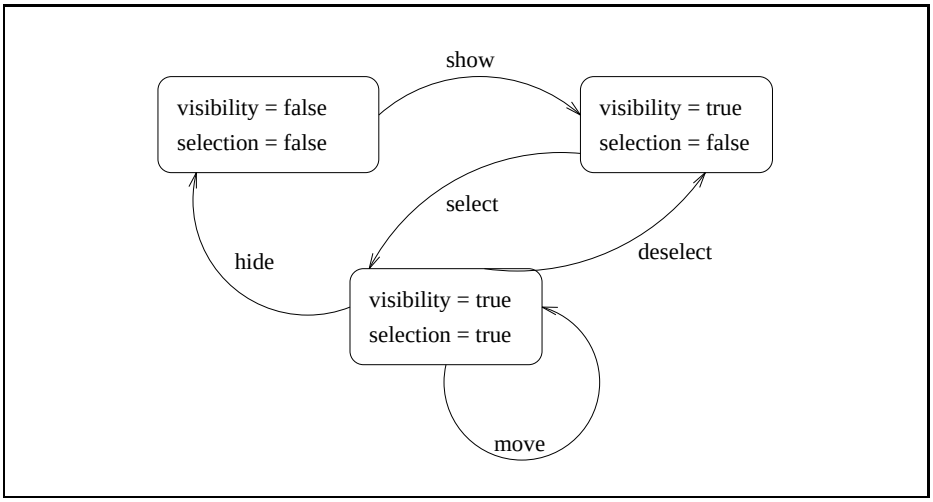

Fig. 9. Behaviour Automaton of Type Figure

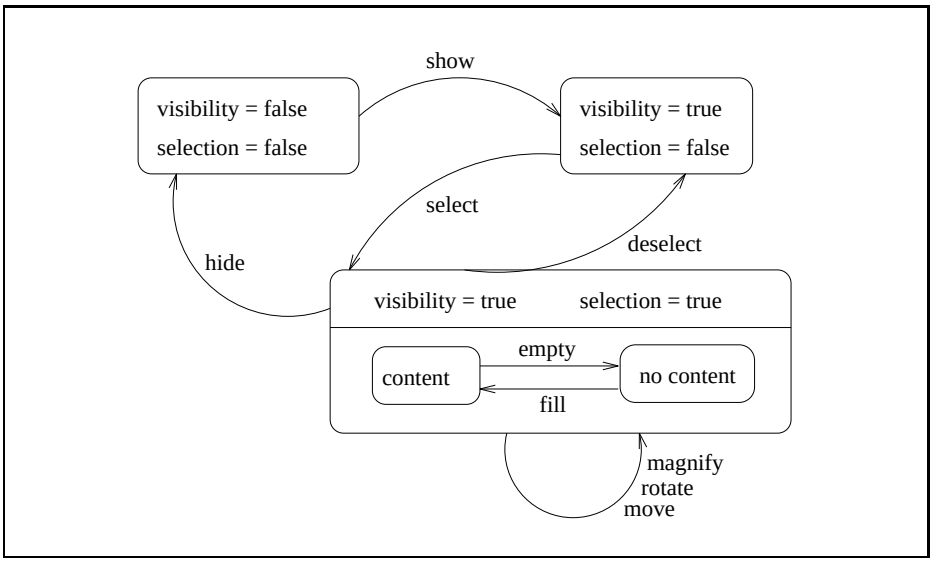

Fig. 10. Behaviour Automaton of Type Two-dimensional Figure

Altogether we have the following characteristics of automata specialization:

- Specialization may restrict the set of possible object states. Every state of the subautomaton has to refine a state of the superautomaton. However, it is possible that states of the superautomaton have no correspondent in the subautomaton. It is also possible that the correspondent state in the subautomaton is not satisfiable by an object state (e.g. because of strengthening the invariant).
- Transitions of the superautomaton are preserved, if their enabling states are preserved. However, reduction of non-determinism is allowed (and also desirable). Since transition constraints correspond to an exhaustive case analysis, they have to be preserved in any case. Enabledness of an operation can be restricted and extended. The latter only, if no additional non-determinism is introduced. The former only, if it is induced by reduction of the state space in the specialized automaton.

Note the way this spezialization relation reflects the automata semantics. Since automata characterize the maximal set of states of a type, this set can only be reduced in a subtype. Relative to the restricted state space the set of enabling states of an operation can be extended (the set of enabling states being minimal), while the set of successor states of these enabling states can only be reduced (by maximality of sucessor states).

Now let us give a formal definition of behaviour automata specialization:

Definition of Behaviour Automata Specialization:

Let $\mathcal{A}_i = (S_i, \Delta_i, L_i, I_i), i = 1, 2$ be two behaviour automata. $\mathcal{A}_2$ *specializes* $\mathcal{A}_1$ (denoted as $\mathcal{A}_2 \leq \mathcal{A}_1$), if

Sp1. the invariant is preserved, that means $\models I_2 \Rightarrow I_1$.

Sp2. the set of possible object states is not extendend and the partition is preserved, that means for all $s_2 \in S_2$ there exists $s_1 \in S_1$ such that $\models (I_2 \wedge L_2(s_2)) \Rightarrow L_1(s_1)$.

Sp3. transitions are preserved, if their enabling states are preserved, that means for all $s_i \in S_i, i = 1, 2$ such that $\models (I_2 \wedge L_2(s_2)) \Rightarrow L_1(s_1)$ and $\not\models (I_2 \wedge L_2(s_2)) \Leftrightarrow false$ and for all f such that $enabled_1(s_1, f) \neq \emptyset$ holds:

Sp3a. $enabled_2(s_2, f) \neq \emptyset$

Sp3b. for all $t_2 \in S_2, p_2 \in \mathcal{L}$ with $\Delta_2(s_2, f[p_2], t_2)$ there exists $t_1 \in S_1, p_1 \in \mathcal{L}$ with $\Delta_1(s_1, f[p_1], t_1)$ and $\models (I_2 \wedge L_2(s_2)) \Rightarrow (p_2 \Rightarrow p_1)$ and $\models (I_2 \wedge L_2(t_2)) \Rightarrow L_1(t_1)$.

Note that spezialization permits the elimination of invalid states (whose labels contradicts the invariant) with no incoming transitions. Invalid states with incoming transitions can only be eliminated through reduction of non-determinism. Otherwise they indicate a meaningless and therefore not implementable behaviour model.

Based on automata specialization one can define *type specialization*. The only additional requirement is that initial states have to be preserved and visible attributes and operations remain visible[3].

Definition of Type Specialization:

Let $T_i = (name_i, Att_i, VAtt_i, Op_i, VOp_i, par_i, \mathcal{A}_i, s_{0i}), i = 1, 2$ be two types. T_2 is a subtype of T_1 (denoted as $T_2 \leq T_1$), if $Att_1 \subseteq Att_2$, $VAtt_1 \subseteq VAtt_2$, $Op_1 \subseteq Op_2$, $VOp_1 \subseteq VOp_2$, par_2 restricted to Op_1 equals par_1, $\mathcal{A}_2 \leq \mathcal{A}_1$ and $\models (I_2 \wedge L_2(s_{02})) \Rightarrow L_1(s_{01})$.

We close this section with the proof of transitivity of automata specialization. The transitivity of type specialization follows trivially.

Theorem 1: Transitivity of specialization

Let $\mathcal{A}_i = (S_i, \Delta_i, L_i, I_i), i = 1, 2, 3$ be three behaviour automata. Then $\mathcal{A}_3 \leq \mathcal{A}_2$ and $\mathcal{A}_2 \leq \mathcal{A}_1$ imply $\mathcal{A}_3 \leq \mathcal{A}_1$.

Proof:

Sp1: $\models I_3 \Rightarrow I_1$ follows trivially from transitivity of implication.

Sp2: Similarly, for $s_3 \in S_3$ follows $\models (I_3 \wedge L_3(s_3)) \Rightarrow L_1(s_1)$ for some $s_1 \in S_1$.

Sp3a: Now let $s_3 \in S_3, s_1 \in S_1, f \in \mathcal{OP}$ such that $I_3 \wedge L_3(s_3)$ is satisfiable and $\models (I_3 \wedge L_3(s_3)) \Rightarrow L_1(s_1)$ and $enabled_1(s_1, f) \neq \emptyset$.
Since $\mathcal{A}_3 \leq \mathcal{A}_2$ there exists $s_2 \in S_2$ with $\models (I_3 \wedge L_3(s_3)) \Rightarrow L_2(s_2)$. Therefore $L_2(s_2) \wedge L_1(s_1)$ is satisfiable. By exclusiveness of state labels of $\mathcal{A}_1$ and $\mathcal{A}_2 \leq \mathcal{A}_1$ follows $\models (I_2 \wedge L_2(s_2)) \Rightarrow L_1(s_1)$. Since $\mathcal{A}_2 \leq \mathcal{A}_1$ follows $enabled_2(s_2, f) \neq \emptyset$ and by $\mathcal{A}_3 \leq \mathcal{A}_2$ follows $enabled_3(s_3, f) \neq \emptyset$.

Sp3b: Now let $\Delta_3(s_3, f[p_3], t_3)$ for some $t_3 \in S_3, p_3 \in \mathcal{L}$. By $\mathcal{A}_3 \leq \mathcal{A}_2$ follows $\Delta_2(s_2, f[p_2], t_2)$ for some $t_2 \in S_2, p_2 \in \mathcal{L}$ and $\models (I_3 \wedge L_3(s_3)) \Rightarrow (p_3 \Rightarrow p_2)$ and $\models (I_3 \wedge L_3(t_3)) \Rightarrow L_2(t_2)$. By $\mathcal{A}_2 \leq \mathcal{A}_1$ follows $\Delta_1(s_1, f[p_1], t_1)$ for some $t_1 \in S_1, p_1 \in \mathcal{L}$ and $\models (I_2 \wedge L_2(s_2)) \Rightarrow (p_2 \Rightarrow p_1)$ and $\models (I_2 \wedge L_2(t_2)) \Rightarrow L_1(t_1)$. By transitivity of implication follows $\models (I_3 \wedge L_3(s_3)) \Rightarrow (p_3 \Rightarrow p_1)$ and $\models (I_3 \wedge L_3(t_3)) \Rightarrow L_1(t_1)$. ♠

4 Refinement

Refinement is used to incrementally develop the system model. In our view type refinement coincides with type specialization. Typical refinement steps are reduction of non-determinism and change of data representation. The former has already been discussed in the context of type specialization and is reflected in

[3] We do not allow addition of parameters to operations through type specialization, since this violates the subtype requirement.

the definition of behaviour automata specialization through condition (Sp3). The latter can be achieved through addition of a new attribute and a new invariant relating the new and the old attribute. For example, the type `Queue` can be refined through addition of an attribute `elements` of type `list` to hold the elements of the queue (see Figure 11).

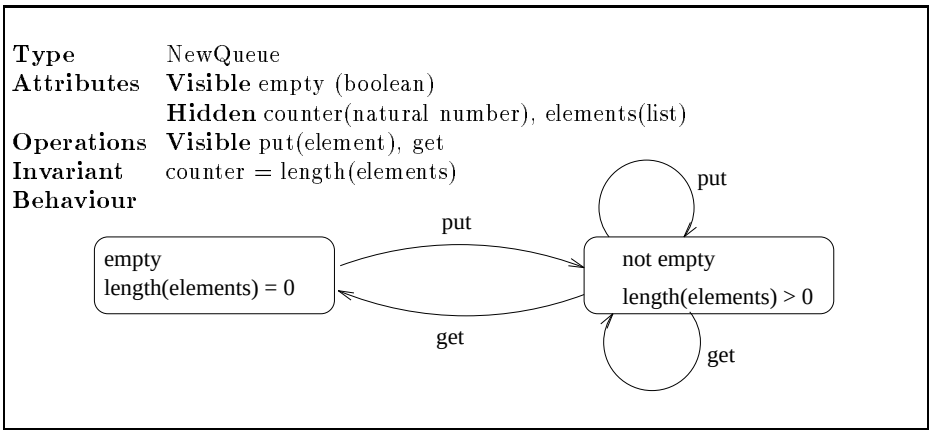

Fig. 11. Type NewQueue

The relation to `counter` is given by the invariant `counter = length(elements)` . Often instead of an invariant an abstraction function is used (e.g. [LW93]) to relate the different representations. Our definition of type specialization can easily be extended such that only the visible attributes and operations need to be preserved, while hidden attributes and operations can also be related through abstraction functions.

Note that in our framework it is not possible to specify that `put` adds exactly one element to the queue, since we are only dealing with finite state automata. For fine grained specification a more expressive formalism has to be used. Therefore we suggest to use an algebraic specification language like SPECTRUM [BFG+93] together with our concept of behaviour automata.

We also allow reduction of the state space through refinement. As an example consider the type `Bounded Queue` (see Figure 12).

If we add attributes `full` and `bound` in Figure 3, type `Queue` can be seen as a refinement of `Bounded Queue` via the addition of the invariants

```
counter < bound, bound > 1, not full
```

to `Bounded Queue`. It is somewhat unusual to regard a `bound` as a dynamic changing value, but `bound` can be seen as the amount of dynamically allocated memory to store the elements of the queue.

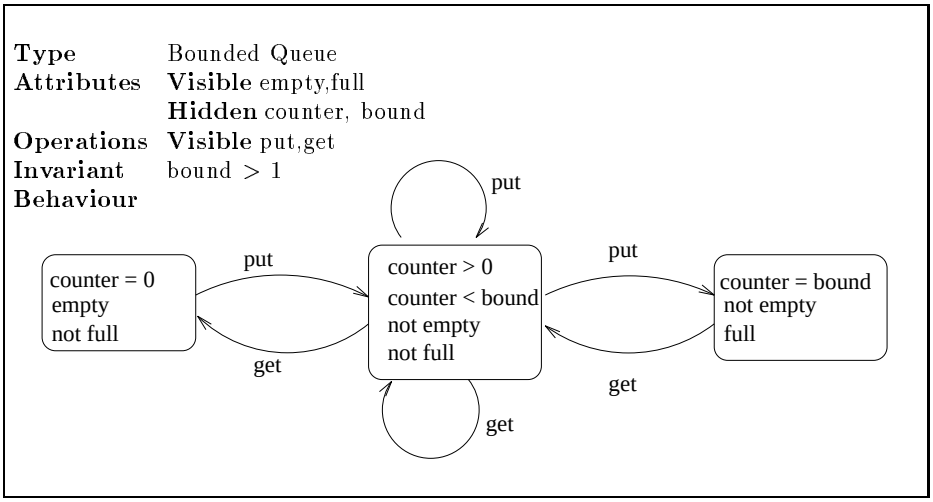

Fig. 12. Type Bounded Queue

It is interesting to compare our notion of refinement with the refinement used for specifications in rely/guarantee style (e.g. [SDW93]). A rely/guarantee pair $(R_2 \Rightarrow G_2)$ is a refinement of $(R_1 \Rightarrow G_1)$, if $R_1 \Rightarrow R_2$ and $(R_1 \wedge G_2) \Rightarrow G_1$.

Taking R_f and G_f as defined in section 2 it is straightforward to show that $\mathcal{T}_2 \leq \mathcal{T}_1$ implies for all operations $f \in \mathcal{OP}$, object states $os_1, os_2 \in OS(T_2)$ and arguments $parg \in [par(f) \rightarrow U]$ that

$$R_f^1(os_1) \Rightarrow R_f^2(os_1) \text{ and}$$
$$(R_f^1(os_1) \wedge G_f^2(os_1, os_2, parg)) \Rightarrow G_f^1(os_1, os_2, parg).$$

Thus type specialization implies operation refinement. The opposite is not true, since type specialization also induces structural refinement of the automaton which cannot be expressed by operation refinement.

5 Roles

In the preceeding sections we have discussed how to model the behaviour of a *single* type and its subtypes. To understand the interaction between several types it is important to make explicit the expectations the environment has about the behaviour of a type. These expectations are often called *roles* of the type [Ree92]. A typical example is the `Bounded Queue` whose environment consists of a `Producer` and a `Consumer` (for the type description see Figure 12).

The `Producer` expects to be able to invoke some number of `put` operations (depending on the `bound` and the `counter`). The `Consumer` expects to be able to invoke some number of `get` operations. The expectations can be made explicit

in the `Enqueue` and `Dequeue` role, respectively. These roles are captured by the behaviour automata depicted in Figure 13. Note that in roles all attributes and operations are visible.

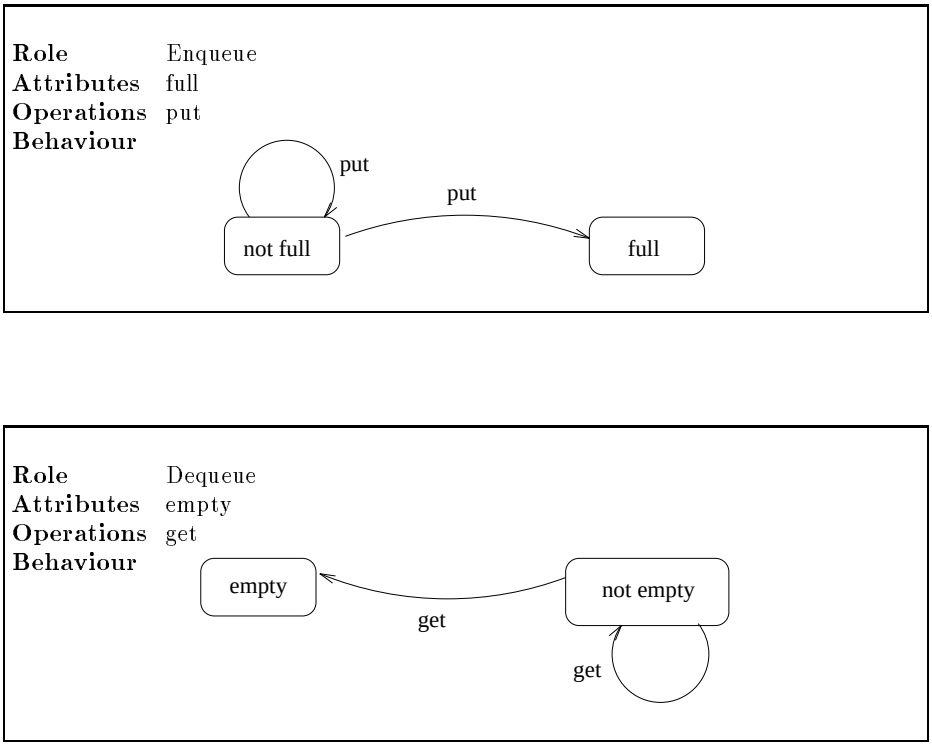

Fig. 13. Roles Enqueue and Dequeue

The `Producer` is only concerned with the attribute `full` and doesn't care about `empty`. Similarly, the `Consumer` doesn't care about `full`, but only is concerned with `empty`.

Having made the expectations explicit as roles it is easy to check whether a type fulfills its roles. The behaviour automaton of the type has to specialize the behaviour automata of its roles. In the example above this is straightforward to show. The only interesting point is the split up of one state of every role automaton in two states of the type automaton. In the case of `Enqueue` state s labelled with `not full` is split up into s_1, s_2 labelled `empty, not full` and `not empty, not full` respectively. The implications $(I \wedge L_{Queue}(s_i)) \Rightarrow L_{Enqueue}(s)$ are straightforward to show. Also condition (Sp3) is straightforward to show.

In general it is important to note that, although the expectations are captured

by automata, nothing is said about *sequences* of operations the environment of an object may invoke. The semantic of behaviour automata is only concerned with states, the enabledness of operations and to some extend their result. Since we allow a concurrent environment, a user object o cannot expect the used object to be modified solely according to the sequence of operations invoked by o. Therefore single operations invocations are the units of concern.

In the following we give a formal definition of roles.

Definition of Roles:

Let $T = (t, Att_t, VAtt_t, Op_t, VOp_t, par_t, s_0, \mathcal{A}_t)$ be a type description.
$R = (r, VAtt_r, VOp_r, par_r, \mathcal{A}_r)$ is called a *role* of T, if r is a name, $VAtt_r \subseteq VAtt_t$ are the attributes, $VOp_r \subseteq VOp_t$ are the operations, par_t is a function from operations to parameters which restricted to VOp_r equals par_r and $\mathcal{A}_t \leq \mathcal{A}_r$

Note that the only difference to the subtype relation is that for roles no initial states are specified[4] and that roles do not have hidden, but only visible attributes and operations.

Now it is easy to show the subtype requirement.

Theorem 2: Subtype Requirement for Roles holds

Let T_1, T_2 be two type descriptions and R be a role of T_1.
If $T_2 \leq T_1$, then R is also a role of T_2.

The proof is trivial by transitivity of automata specialization.

6 Related Work

In this section we discuss related work. In the framework of [LW93] the behaviour of a type is captured by pre- and postconditions. As mentioned before our description by predecessor and successor states is more coarse. So for example we cannot describe different behaviour for every individual element added to a `Queue` by `put`. More generally, we cannot describe enabledness of operations which depends on parameters[5].

In our framework def_f defines some sort of precondition. As mentioned in section 4 for all $os_1 \in OS(T_2)$ holds $R^1_f(os_1) \Rightarrow R^2_f(os_1)$. This implies that for all $s_2 \in S_2$ holds $\models (I_2 \wedge L_2(s_2)) \Rightarrow (def^1_f \Rightarrow def^2_f)$. Liskov and Wing require

[4] The expectations of an object o on another object which was created by o might also include initial states. This can easily be incorporated in our framework.

[5] However we can model this situation by totalizing the operation and introducing an explicit error state.

that $pre_f^1 \Rightarrow pre_f^2$. Thus our specialization is more general, since we only require the implication $pre_f^1 \Rightarrow pre_f^2$ for the states the refined objects can assume. This weaker requirement can only be formulated because we have made the state space of a type explicit. Another difference is the treatment of invariants. In [LW93] every property of the supertype is preserved. In OOA this is too strong a requirement, since the specification of the types is not complete. Consider the example of `Fat Sets` given in Figure 14.

Type	Fat Set
Attributes	**Hidden** elements
Operations	**Visible** insert, select, size

Type	Set
Attributes	**Hidden** elements
Operations	**Visible** insert, delete, select, size

Fig. 14. Types Set and Fat Set

`Fat Sets` cannot shrink because the `delete` operation is missing. `Sets` can be seen as a refinement of `Fat Sets` by addition of `delete`. Clearly, the property not to shrink is lost. Therefore in [LW93] `Sets` are not allowed as specialization of `Fat Sets`. In our framework the analyst can decide. If the property not to shrink is important, it can be formulated as an invariant of the behaviour automaton[6]. Then no `delete` operation can be added. If the property is not deemed important and the invariant not specified, it is not required of subtypes and therefore `Set` can be a subtype of `Fat Set`.

In some sense our definition of subtyping can be seen as an integration of the work of [LW93] and [Fro92]. Frølund requires the synchronization constraints of the supertype to be an "upper limit" on the subtypes. In his framework therefore the subtype requirement can be violated. In our framework the state space of the supertype is an upper limit for the state space of the subtypes and the synchronization constraints depend on the state space. Relative to the reduced state space of the subtype, however, we require the synchronziation constraints of the supertype to be stronger than the ones of the subtypes (see the discussion about def_f above).

[6] Note that we have not considered such temporal invariants so far.

Recently, yet another definition of type specialization has been given in [SHJ+94], which is also based on automata. In that work automata states are not labelled, therefore only the trace set of the automaton is relevant for the type semantics. Contrary to our work the trace set, and therefore also the set of enabling states, is considered to be maximal. Thus that definition of automata specialization allows for reduction of operation enabledness.

7 Conclusion

In this paper we have shown how to model the behaviour of types by automata in a precise way. By labelling the automata states with predicates we make the state space of a type explicit. These labels partition the state space. The transitions are labelled by operations. Thus operations are only classified in so far as they relate different equivalence classes of the state space. In our view this level of granularity is adequate during the process of OOA, while specification with pre- and postconditions is too fine-grained. Behaviour automata allow to grasp quickly important classes of object states and the effect of operations on them. We have given a precise definition of automata specialization. This not only provides for a new definition of the subtype relationship, but also shows how to *safely* enhance the system model during the analysis and design process. Thus we have sketched the use of our specialization mechanism for type refinement. Our definition of type specialization guarantees the subtype requirement, namely that as far as the environment is concerned the object of the supertype and the subtype cannot be distinguished. Using behaviour automata we have defined precisely the expectations of the environment.

In our future work we intend to extend this work to the aggregation hierarchy and general interaction between types. This requires to make explicit in the operations of a type invocation of operations on other types. Similar to [SHJ+94] we plan to give a set of automata transformations which is correct and complete wrt. our refinement definition.

Since automata are a very intuitive means to model behaviour, it is easy to use behaviour automata in an informal way. The formal definitions guide the user to the important points where to look at while designing and modifying the behaviour of types. Thus behaviour automata are an adequate link between informal and formal software development methods.

Acknowledgments

We would like to thank Klaus Bergner, Manfred Broy, Thomas Fees, Stefan Merz, Stefan Schiffer, Lothar Schmitz and Alois Stritzinger for critical discussions and two anonymous referees for helpful comments during the preparation of this paper.

References

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0. Technical Report TUM-I9312, Technische Universität München, 1993.

[DDP93] E. Dubois, P. DuBois, and M. Petit. Oo requirement analysis: an agent perspective. In *ECOOP,LNCS 707*, pages 458–481. Springer Verlag, 1993.

[Fro92] S. Frolund. Inheritance of synchronization constraints in concurrent oo programming languages. In *ECOOP,LNCS 615*, pages 187–196. Springer, 1992.

[GM93] D. Gangopadhyay and S. Mitra. Objchart: Tangible specification of reactive object behaviour. In *ECOOP,LNCS 707*, pages 432–457. Springer Verlag, 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM ToPLaS*, 5(4):596–619, 1983.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The troll language. report 91-04, TU Braunschweig, 1991.

[LW93] B. Liskov and J.M. Wing. A new definition of the subtype relation. In *ECOOP,LNCS 707*, pages 118–141. Springer Verlag, 1993.

[Mes93] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *ECOOP,LNCS 707*, pages 220–246. Springer Verlag, 1993.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MO93] J. Martin and J. Odell. *Object-oriented Analysis and Design*. 1993.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, 1991.

[Ree92] Trygve Reenskaug et al. OORASS: seamless support for the creation and maintenance of object oriented systems. *JOOP*, 5(6):27–41, October 1992.

[SDW93] K. Stolen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. report TUM-I9303, TU München, 1993.

[SHJ+94] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra. Inheritance conditions for object life cycle diagrams. In *GI-Workshop Formale Grundlagen für den Entwurf von Informationssystemen*. Universität Hannover, Informatik Bericht 03/94, 1994.

This article was processed using the LaTeX macro package with LLNCS style