Sharing Actions and Attributes in Modal Action Logic[*]

Mark Ryan, José Fiadeiro and Tom Maibaum

Department of Computing, Imperial College, London SW7 2BZ, Great Britain

mdr@doc.ic.ac.uk

Abstract

Distributed systems may be specified in Structured Modal Action Logic by decomposing them into *agents* which interact by sharing *attributes* (memory) as well as *actions*.

In the formalism we describe, specification texts denote theories, and theories denote the set of semantic structures which satisfy them. The semantic structures are Kripke models, as is usual for modal logic. The "possible worlds" in a Kripke model are the states of the agent, and there is a separate relation on the set of states for each action term.

Agents potentially share actions as well as attributes in a way controlled by locality annotations in the specification texts. These become locality axioms in the logical theories the texts denote. These locality axioms provide a refined way of circumscribing the effects of actions.

Safety and liveness conditions are expressed (implicitly) by deontic axioms, which impose obligations and deny permissions on actions. We show that "deontic defaults" exist so that the specifier need not explicitly grant permissions or avoid obligations in situations where normative behaviour is not an issue.

1 Introduction

The idea of using Modal Action Logic for specifying distributed systems is well-established [3, 4, 5]. Additionally the frame problem can be overcome by specifying *structure* on specifications—the system is split into *agents* (or objects, or components) which interact by sharing actions. This is the approach taken by Fiadeiro & Maibaum [1], and fits well with object-oriented specification: an object has a private memory and public procedures for its manipulation.

But often, as we discuss below, it is more natural to share *attributes* between agents than to share *actions*. In this paper we give a logical semantics to a pseudo-language in which both attribute and action sharing are allowed. The logic is called Structured Modal Action Logic (Structured MAL).

The agent is the unit of structure. An agent is any component of the system being described which has an independent existence; it may be passive or active. Agents are composable. A collection of agents can be viewed as a single agent in a precise way to be described later. The single agent incorporates the behaviours of the individual agents. As already stated, agents can interact by sharing attributes or by sharing actions. An *attribute*

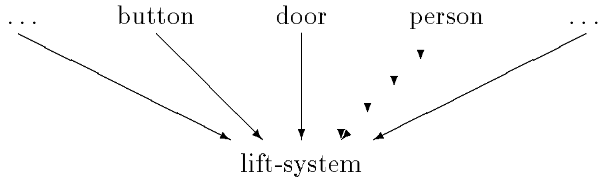
[*] Appeared in Proc. Theoretical Aspects of Computer Software (TACS'91), Sendai, Japan, eds. T. Ito and A. Meyer. Springer Verlag Lecture Notes in Computer Science 526, pages 569–593, 1991.

is part of the state of the system—a predicate or function which varies not only with its arguments but also with the state of the agent to which it belongs. In general agents should be as self-contained as possible; the actions of an agent should only update attributes of that agent, and vice versa. But clearly some interaction between agents is necessary. The interaction is precisely controlled by means of *locality axioms.* For each agent, the specifier must declare which of the actions and attributes are local and which are sharable with other agents. A sharable action is one which can update the attributes of a superagent (one which incorporates it) — these attributes may come from other agents which the superagent incorporates. Similarly, a sharable attribute is one which can be updated by the action of a superagent.

Throughout this paper we will make use of the **lift-system** example. The atomic agents which might make up the specification include:

- **button**, with the action 'press' and the attribute 'lit' (buttons have lights on them)
- **door**, with the actions 'open' and 'close' and the attribute 'posn' (their position).
- **person**, with, among others, the action 'press-button'.

Of course there are other agents, but we will concentrate on these three for the purpose of motivating the two types of interaction which Structured MAL adopts. These agents make up the lift specification:



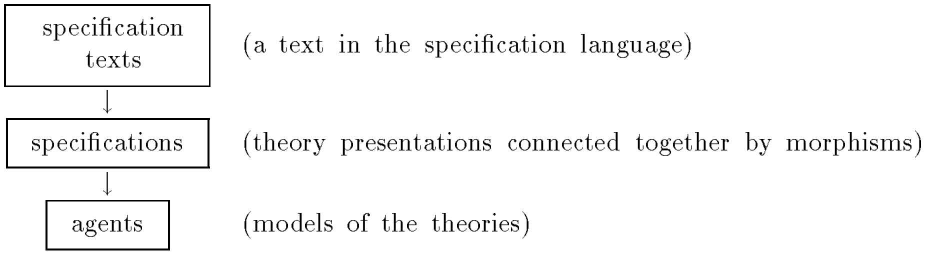
When the person selects a button and presses it, this is really an in action which both the person and the button participate. True, the person initiates the action; the button participates in it in a purely passive way. But since the button moves in and out, and could do so independently of the person if incorporated in a specification in a different way, the natural choice is to specify this interaction by joint participation of actions, that is, action sharing. We do this by having a press action in both the agents **person** and **button** say person.press and door.press, and then including the axiom person.press = door.press in the **lift-system**.

Now consider what happens when the lift arrives at a floor and the door opens. The light of the appropriate button is extinguished. We could specify that both the door and the button jointly participate in an action which opens the door and extinguishes the light. But it makes more sense to say that the door's opening extinguishes the light — the button has nothing to do with it. The action 'open' of the agent **door** directly updates the attribute 'lit' of **button**. This is attribute sharing.

Both these examples are fairly marginal. One could make a case for specifying the first one as an example of attribute sharing and the second one as an example of action sharing. The specifier is free to do this if he or she wishes. In Structured MAL, both types of sharing can be used freely.

Each agent specification has a collection of actions and attributes — the collection is known as its *signature.* In addition, the specification language also says which actions and attributes are *local* and which are *sharable.* If an action is declared as local to an agent, an axiom is generated which states that the action can only update the attributes of that agent. Similarly, a local attribute comes with an axiom which says that it can only be changed by an action belonging to the agent in which it is defined. These *locality axioms* are examined in Section 5.

Locality axioms are not an explicit part of the specification in the way that other axioms are. They are part of the theory presentations which the specifications denote. What we have is a three tiered system which looks like this:


specification texts
(a text in the specification language)
specifications
(theory presentations connected together by morphisms)
agents
(models of the theories)

A specification text is a text in the specification language. It specifies the behaviour of a collection of interacting agents. This text denotes a family of theory presentations, one presentation for each agent specified, connected together by *morphisms.* A model of a theory presentation is an agent which satisfies the specification.

The best way to think of this is to bear in mind that the level of primary interest is the middle level. That level consists of theories connected by morphisms. The level above, the specification language level, says how this is to work by using high level constructs like inheritance, clientship and parameterisation. At the middle level there may be more agent specifications (theories) than in the specification text, because many of them will be by-products of the high-level constructs used. At the bottom level, models of the theories correspond to agents which meet the specifications.

The term *morphism* comes from category theory. Indeed, theory presentations and their morphisms form a category in which, following general categorical principles [2], colimits explain how to build a complex system from a diagram that expresses how its components are interconnected. See [1] for the application of this principle to the specification of object-oriented systems. As far as we are concerned here, a morphism is simply a map with certain properties between theory presentations with which we can specify agent interaction. A precise definition comes later. The agent diagram above is an example of a morphism diagram.

2 Agents and morphisms

Agents are the units of structure. Each agent is an encapsulation of behaviour — it consists of a state (the values of its attributes) which is changed as it performs actions. An action or an attribute may be entirely local to an agent, or it may be shared with other agents. If an action is local to an agent **X**, it can only change the values of the attributes of **X**. Of course it need not change them all, but it cannot affect any others. If an attribute is local to **X**, it can only be changed by the actions of **X**. We will call these two types of locality *action locality* and *attribute locality*, respectively.

The lift example starts with the 'atomic' agents **button** and **door**. Buttons have lights and illuminate when pressed. They are extinguished by actions which are external to them (the opening of the door). So **button** looks like this:

```
agent button
attributes
    s   lit : bool;
actions
    ℓ   press;
axioms
    [press]lit;
end
```

This is the text which the specifier writes in the specification language. The ℓ and s annotations mean local and sharable, respectively. The signature of the theory presentation which this denotes consists of the attribute lit and the action press. The theory presentation has two axioms; one comes from the specification text, and says simply that lit is true after press has taken place[1]. The other is the locality axiom for the local action press, and says that press can only affect the attributes of the agent **button**. How this is done will be revealed in section 5. The attribute 'lit' is sharable, because it will have to be updated by actions taking place in other agents (namely, the opening of the door).

The attribute lit has values of sort boolean. Atomic formulas are equalities of values in a sort, often of the form 'attribute = value'. If the sort concerned is boolean, we glibly let attribute values stand as formulas in the obvious way. Thus '[press]lit' is an abbreviation for '[press](lit = true)'.

Doors also have one attribute, their position, and two actions, open and close.

```
agent door
attributes
    ℓ   posn : (op, cl);
actions
    s   open;
    ℓ   close;
axioms
    [open](posn = op)
    [close](posn = cl)
end
```

The attribute 'posn' has the enumerated sort (op, cl). Everything is local except the action open, which has to be sharable to be able to extinguish the lights.

Inside each lift there is a panel of lift buttons. For an n-floor system, the agent **lift-buttons** is made of n copies of **button**. At the specification language level, its specification looks like this:

```
agent lift-buttons
includes button via 1;
includes button via 2;
⋮
includes button via n;
end
```

[1] In fact, the light only comes on if the lift is at a floor *other than* the one being requested. An elegant way of handling this fact by means of *defaults* is described in [6].

The clause '**includes** button **via** 1' means that at the theory presentation level there is a morphism, named 1, from **button** to **lift-buttons**. The morphism maps the action symbol press in **button** to the action symbol 1.press in **lift-buttons**. The clause '**includes** subagent **via** morphism-name' implicitly declares all signature symbols and axioms of subagent in the agent being specified, but renaming the signature symbols by prefixing them by the morphism name. At the theory presentation level, the theory which **lift-buttons** denotes has n sharable attributes 1.lit, 2.lit, …, n.lit and n local actions 1.press, …, n.press, one 'lit' attribute and one 'press' action for each button.

We need to describe one more agent before we can describe a **lift**: it is the 'agent' **lift-position**.

```
agent lift-position
attributes
    $\ell$  floor : $1..n$
actions
    $\ell$  up;
    $\ell$  down;
axioms
    $(\text{floor} = f) \wedge (f < n) \rightarrow [\text{up}](\text{floor} = f + 1)$
    $(\text{floor} = f) \wedge (f > 1) \rightarrow [\text{down}](\text{floor} = f - 1)$
    $\text{per(up)} \rightarrow \text{floor} < n$
    $\text{per(down)} \rightarrow \text{floor} > 1$
end
```

The axioms containing the formulas per(up) and per(down) are deontic axioms; they express the fact that the lift is only *permitted* to move up or down when the floor variable is within the right bounds. There are also deontic axioms which express *obligations*. These deontic axioms are described in section 3.

The reader may be surprised that **lift-position** deserves the status of an agent, but there is an advantage of having it as a separate agent rather than just including its attributes and actions in the specification of **lift**, which is that the local actions up and down are then constrained to being able to update the value of floor only. Remember that an action local to an agent gets a locality axiom in the theory presentation which says that it can only affect the attributes of that agent. The smaller the agent, the more powerful the locality axiom. Indeed, a principle of this approach, the "structuring principle", is that all structuring should be done by judicious choice of agents, and hence of locality constraints. The specifier need never get involved in including explicit locality axioms in specifications; they should all be implied by making actions and attributes local or sharable.

The lift itself consists of the agents **lift-buttons**, **door** and **lift-position**:

```
agent lift
includes door via door;
includes lift-buttons
    localising 1.lit,...,$n$.lit ;
includes lift-position;
axioms
    $(\text{floor} = 1) \rightarrow [\text{open}]\neg 1.\text{lit};$
    ⋮
    $(\text{floor} = n) \rightarrow [\text{open}]\neg n.\text{lit};$
    $\text{per(up)} \vee \text{per(down)} \rightarrow \text{door.posn=cl};$
end
```

The **localising** clause makes the attributes i.lit local in the agent lift. (Without this clause, they would retain the sharable status with which they were defined in the agent **button**.) This has the effect of adding a locality axiom in **lift**, in the way described later.

The theory presentation denoted by this specification text has all the signature elements and the axioms of its constituents. Since no renaming of the actions and attributes is necessary, the morphisms are labelled 'id' (for identity). **lift** also has an additional axiom, which we have referred to previously: $(\text{floor} = f) \rightarrow [\text{open}]\neg f.\text{lit}$. It says that when the door opens, the light corresponding to the floor at which the door has opened switches off. Here is the morphism diagram at the theory presentation level:

$$\begin{tikzcd}
 & \text{button} \arrow[dr, "1"'] & \text{button} \arrow[d, "2"] & \dots & \text{button} \arrow[dll, dotted, "n"] & \\
\text{door} \arrow[drr, "\text{id}"'] & & \text{lift-buttons} \arrow[d, "\text{id}"] & & & \text{lift-position} \arrow[dlll, "\text{id}"] \\
 & & \text{lift} & & &
\end{tikzcd}$$

3 Language and logic

In this section we describe the language and the logic used in the theory presentations, which are denoted by agent descriptions in the specification language.

We have mentioned that we have two types of locality, action locality and attribute locality. To be able to give the locality axioms that go with these two kinds, our language must be sufficiently rich to compare actions and to compare attributes. Comparing actions does not represent a problem because they will just be terms of the sort **action**. But by comparing attributes we mean comparing the actual attributes, not their values. Therefore we must distinguish between references and values — a distinction well known in programming languages. When we write $A = B$ we mean that attributes A and B have the same value. But $\&A = \&B$ means A and B are actually the same attribute — they refer to the same "cell". & is the reference or 'address-of' operator. To de-reference attribute names we use the $*$ operator. Thus $*\&A$ is the value of $\&A$, otherwise written A.

As we said, the specifier does not have to use these operators; they will only be needed for locality axioms, which, by the structuring principle mentioned above, the specifier will never have to provide explicitly. The locality axioms will be implicit in the specification texts. Nevertheless, the locality axioms are explicit in the theory presentations which the specifications denote, so we need to have the logical language to describe them.

3.1 Signatures

We said that a signature is the extralogical language in the theory presentation, *i.e.* the attribute and action symbols. It also consists of the sort symbols used and the usual functions. For example, in the agent **button** we used the sort boolean, and might have used the usual functions 'and', 'or' *etc.* which come with it.

A signature is:

- A family of sort symbols S with function symbols
- The special sorts **action** and, for each sort symbol $s \in S$, $\mathbf{ref}_s$. Terms of sort $\mathbf{ref}_s$ are names of (pointers to) values of sort s, and will be de-referenced by $*$.
- An S^*-indexed family of action symbols (action terms are of sort **action**)
- An $S^* \times S$-indexed family of attribute symbols (attribute terms formed from the attribute symbol $A : s_1 \times \ldots \times s_n \times s$ are of sort s).

Notice that actions and attributes can be parameterised by terms of sorts of the signature. (In the lift example, there happened not to be any parameterised actions or attributes.)

At the theory presentation level we do not distinguish between local and sharable actions and attributes. At this level that information is carried by the locality axioms. The annotations l and s appear at the specification language level, and the 'specification compiler' generates the locality axioms from them.

3.2 Language of theory presentations

This section describes the language of the theory presentations. As we have said, the language used by the specifier to write axioms is only a subset of this: the specifier does not need to make explicit use of the operators & and *, in view of what we have called the structuring principle.

We will describe the language for a signature with sorts S. In addition to the symbols of the signature, we have for each sort s in S enumerably many variables x, y,... of the sort s and α,... of the sort $\mathbf{ref}_s$, and enumerably many variables $x, y \ldots : \mathbf{action}$.

- Terms are formed using
  - variables of the sorts
  - function symbols of the sorts S with appropriately sorted term parameters; constant symbols of the sorts are simply zero-arity function symbols.
  - attribute symbols, again with appropriately sorted term parameters. Zero-arity attribute symbols vary only with the state of the agent.

    The values of functions are invariant across states, whereas attributes change from state to state.
  - action symbols, again with zero or more appropriately sorted term parameters.
  - for each attribute term $\mathbf{A} : s$, there is a term $\&\mathbf{A} : \mathbf{ref}_s$ (its "reference"). Notice that $\mathbf{A}$ is an attribute term, not an attribute symbol; in other words its parameters are instantiated by appropriately sorted terms. This means that an attribute symbol has a different reference for each set of (semantically distinct) actual parameters.
  - for each term $\alpha : \mathbf{ref}_s$ there is a term $*\alpha : s$ (its "contents").

- atomic formulas are equalities of terms ($t_1 = t_2$), and deontic formulas (per($\mathbf{a}$) and obl($\mathbf{a}$) for action terms $\mathbf{a}$). As we have said, if t is a term of sort boolean, we will sometimes abbreviate the atomic formula $t =$ true by t.
- formulas are made from smaller formulas with the usual connectives and the modal-action operators; if ϕ and ψ are formulas and $\mathbf{a}$ is an action term, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\neg\phi$, and $[\mathbf{a}]\phi$ are formulas.

3.3 Interpretation

Having described the language in which agents are described at a logical level, it is now time to see what models there are of such descriptions. Remember that an agent description is a pair consisting of a signature and a theory presentation (a finite set of axioms). Such a pair comes from an agent described by a specification text. At the logical level, all agent descriptions are simply ⟨signature,presentation⟩ pairs.

The semantic structures are Kripke models, as usual for modal logic. See [3] for details of Kripke frames for multi-modal logics. In the semantics that follows, for each agent there is a Kripke frame. Agent composition corresponds to taking the product of Kripke frames, modulo shared actions and attributes. The "possible worlds" in a frame are the states of the agent, and there is a separate relation on the set of states for each action term. The states interpret non-modal formulas locally (*i.e.* without reference to other states) by evaluating the terms in the sets which interpret the sorts, for atomic propositions, and by the usual truth-table definitions for the non-modal connectives.

An interpretation structure for a signature with sorts S consists of:

- for each sort s in S, a non-empty domain of individuals D_s; and for each function symbol $f : s_1 \times \ldots s_n \rightarrow s$, a function $[\![f]\!] : D_{s_1} \times \ldots \times D_{s_n} \rightarrow D_s$
- a set of states W, and a designated initial state $w_0 \in W$
- a set $\textsc{act} \subseteq W \rightarrow W$, which will be used to interpret action terms (terms of sort **action**).
- for each sort s, a set $\textsc{ref}_s \subseteq W \rightarrow D_s$. These will be used to interpret reference terms (terms of sort $\mathbf{ref}_s$).
- for each action symbol $a : s_1 \times \ldots \times s_n \rightarrow \mathbf{action}$, a function $[\![a]\!] : D_{s_1} \times \ldots \times D_{s_n} \rightarrow \textsc{act}$, as before
- for each attribute symbol $A : s_1 \times \ldots \times s_n \rightarrow s$, a function $[\![A]\!] : D_{s_1} \times \ldots \times D_{s_n} \rightarrow \textsc{ref}_s$
- a pair of sets P and O. Both are subsets of $\textsc{act} \times W$. They are used to interpret the deontic formulas.

An interpretation structure is therefore a set of states and a means of interpreting action terms as functions between states and a way of evaluating state-dependent formulas within states.

To interpret the quantifiers we start by assigning each variable of the language to an individual of the appropriate sort. This is an awkward technicality, and the reader may like to ignore all references to assignments in the passage below the following definition.

An assignment $\mathcal{A}$ is a map from variables to individuals in the carrier set of the variable's sort.

- $x : s$ is assigned to $\mathcal{A}(x) \in D_s$
- $\alpha : \mathbf{ref}_s$ is assigned to $\mathcal{A}(\alpha) \in \textsc{ref}_s$
- $x : \mathbf{action}$ is assigned to $\mathcal{A}(x) \in \textsc{act}$

We now show how to evaluate formulas in states, relative to an assignment $\mathcal{A}$. First we have to evaluate terms, as follows.

1. Variables: if x is a variable of sort s then its interpretation in the state w, written $[\![x]\!]_w^{\mathcal{A}}$, is $\mathcal{A}(x)$ in D_s. Similarly, if $\alpha : \mathbf{ref}_s$ then $[\![\alpha]\!]_w^{\mathcal{A}} = \mathcal{A}(\alpha)$ and if $x : \mathbf{action}$ then $[\![x]\!]_w^{\mathcal{A}} = \mathcal{A}(x)$.
2. Attributes: if $A(t_1, \ldots, t_n)$ is an attribute term of sort s, then its interpretation in state w, written $[\![A(t_1, \ldots, t_n)]\!]_w^{\mathcal{A}}$, is $[\![A]\!]([\![t_1]\!]_w^{\mathcal{A}}, \ldots, [\![t_n]\!]_w^{\mathcal{A}})(w)$. That is to say, to interpret $A(t_1, \ldots, t_n) : s$ in w, apply $[\![A]\!]$ to the interpretations of $t_1, \ldots, t_n$ in w, yielding a function from $W \to D_s$. This function is the reference (address) of the attribute concerned, which is then applied to w to yield the required result.
3. Actions: if $a(t_1, \ldots, t_n) : \mathbf{action}$ then $[\![a(t_1, \ldots, t_n)]\!]_w^{\mathcal{A}} = [\![a]\!]([\![t_1]\!]_w^{\mathcal{A}}, \ldots, [\![t_n]\!]_w^{\mathcal{A}}) \in \textsc{act}$
4. References: $[\![\&A(t_1, \ldots, t_n)]\!]_w^{\mathcal{A}} = [\![A]\!]([\![t_1]\!]_w^{\mathcal{A}}, \ldots, [\![t_n]\!]_w^{\mathcal{A}})$
5. Values: $[\![*\alpha]\!]_w^{\mathcal{A}} = [\![\alpha]\!]_w^{\mathcal{A}}(w)$. The interpretation of reference terms is covered by case 4 and this case. There are no constants or functions of sort $\mathbf{ref}_s$ — the only terms of sort $\mathbf{ref}_s$ are terms like $\&A(t_1, \ldots, t_n)$ and variables.

Formulas are true or false in a state. We will write $[\![\phi]\!]^{\mathcal{A}}$ for the set of states in which ϕ is true. First we deal with the atomic formulas, which are equalities between terms and permissions and obligations of actions.

- The atomic formula $t_1 = t_2$ is true in a state w if t_1 and t_2 have the same interpretations in w. Formally, $w \in [\![t_1 = t_2]\!]^{\mathcal{A}}$ iff $[\![t_1]\!]_w^{\mathcal{A}} = [\![t_2]\!]_w^{\mathcal{A}}$.
- The atomic formula per($\mathbf{a}$) is true in w if $\langle [\![\mathbf{a}]\!]_w^{\mathcal{A}}, w \rangle$ is in P. That is to say, P is just a set specifying what actions are permitted in what states.
- Similarly, $w \in \mathrm{obl}(\mathbf{a})$ if $\langle [\![\mathbf{a}]\!]_w^{\mathcal{A}}, w \rangle$ is in O. O is a set specifying what actions are obliged in what states.

In the discussion above, $\mathbf{a}$ is an action term (a term of type **action**), that is, an action symbol (some a) together with term parameters. Notice that permission or obligation to perform an action $\mathbf{a}$ depends on its parameters (so an agent may have permission to perform the action with some parameters and not with others).

Now for the interpretation of the connectives.

- $w \in [\![\phi \wedge \psi]\!]^{\mathcal{A}}$ iff $w \in [\![\phi]\!]^{\mathcal{A}}$ and $w \in [\![\psi]\!]^{\mathcal{A}}$.
- $w \in [\![\phi \vee \psi]\!]^{\mathcal{A}}$ iff $w \in [\![\phi]\!]^{\mathcal{A}}$ or $w \in [\![\psi]\!]^{\mathcal{A}}$.
- $w \in [\![\phi \to \psi]\!]^{\mathcal{A}}$ iff $w \notin [\![\phi]\!]^{\mathcal{A}}$ or $w \in [\![\psi]\!]^{\mathcal{A}}$.
- $w \in [\![\neg\phi]\!]^{\mathcal{A}}$ iff $w \notin [\![\phi]\!]^{\mathcal{A}}$.

- $w \in [\![[\mathbf{a}]\phi]\!]^{\mathcal{A}}$ iff $[\![\mathbf{a}]\!]^{\mathcal{A}}_{w}(w) \in [\![\phi]\!]^{\mathcal{A}}$. That is to say, to evaluate whether $[\mathbf{a}]\phi$ is true in w, first evaluate the action $\mathbf{a}$ in w. The result is a function in ACT. Then apply this function to w, yielding another state $[\![\mathbf{a}]\!]^{\mathcal{A}}_{w}(w)$. The answer is then given by whether ϕ is true in this new state.
- $w \in [\![\forall x.\, \phi]\!]^{\mathcal{A}}$ if $w \in [\![\phi]\!]^{\mathcal{A}'}$ for all assignments $\mathcal{A}'$ which differ from $\mathcal{A}$ at most by the assignment of x.
- $w \in [\![\exists x.\, \phi]\!]^{\mathcal{A}}$ if $w \in [\![\phi]\!]^{\mathcal{A}'}$ for at least one assignment $\mathcal{A}'$ which differs from $\mathcal{A}$ at most by the assignment of x.

The treatment of quantifiers we have given in the last two clauses above is standard. Recall that for all formulas ϕ, $[\![\phi]\!]^{\mathcal{A}}$ is evaluated relative to an assignment $\mathcal{A}$. We will sometimes ignore this complication and just write $[\![\phi]\!]$, which we justify by the following: If ϕ has no free variables then $[\![\phi]\!]^{\mathcal{A}}$ is independent of the assignment $\mathcal{A}$.

3.4 Logic

We are now in a position to give the consequence relation which defines our logic. Let Γ be the theory presentation corresponding to an agent specification $\mathbf{X}$. To compute the consequences of the specification, we first look at the agents which satisfy it. These are models of Γ. Formally, an interpretation structure $[\![\cdot]\!]$ together with an assignment $\mathcal{A}$ is a model of a sentence ϕ if for every state w of the interpretation structure, $w \in [\![\phi]\!]^{\mathcal{A}}$. A model of a *set* of sentences Γ is a $\langle$structure,assignment$\rangle$ pair which is a model of each of the sentences of Γ.

A model of Γ thus corresponds to an agent which satisfies the specification $\mathbf{X}$. It has a collection of states and undergoes actions which transform it from state to state. But this does not fully reflect the information contained in Γ, because only some actions are permitted in a state, and others may be obliged. We need to look at sequences of actions, *life-cycles*, to reflect this.

Given a model $[\![\cdot]\!]$, a life-cycle is a sequence of state-transitions $\langle e_1, e_2, \ldots\rangle$ where each e_i is in ACT. This is any sequence of actions that the agent can go through. Such a sequence is *normative* if it respects the deontic part of the specification, *i.e.* if each action which takes place is permitted, and if every obligation incurred is discharged. For permissions, this just means that for each n the pair $\langle e_n, w_n\rangle$ must be in P, where w_n is the state arrived at just before the transition e_n, that is, $w_n = e_{n-1}(e_{n-2}(\ldots e_1(w_0)\ldots))$.

One could treat obligations in MAL the other way around: whenever $\langle e, w_n\rangle$ is in O and w_n is $e_{n-1}(e_{n-2}(\ldots e_1(w_0)\ldots))$ for some n, then e must be e_n. However, in Structured MAL it was decided that obligations should be weaker than the 'immediate' obligations mentioned above. Instead of requiring immediate fulfillment, they require eventual fulfillment. That is, when an agent incurs an obligation it must carry out the action at some point in the future, not necessarily immediately. This is characterised in terms of life-cycles as follows: $\langle e_1, e_2, \ldots\rangle$ is normative with respect to obligations if for all e_i in the cycle, if $w_i = e_i(e_{i-1}(\ldots e_1(w_0)\ldots))$ and $\langle e, w_i\rangle \in O$ for some e, then $e = e_j$ for some $j > i$.

Now to define consequences of the specification of $\mathbf{X}$. A is a consequence of $\mathbf{X}$ if it is true in all states that an agent satisfying $\mathbf{X}$ could get into. Let $[\![\cdot]\!]$ be a model of the theory presentation Γ denoted by $\mathbf{X}$. Let $\langle e_1, e_2, \ldots\rangle$ be a normative life-cycle of $[\![\cdot]\!]$. A is a consequence if it is true in all states $e_i(e_{i-1}(\ldots e_1(w_0)\ldots))$.

A life cycle in which an agent incurs an obligation to perform the action **a**, and though it never does perform **a** it performs another action which, in the particular state in which it is performed, is equivalent to **a** in the sense of achieving the same transformation of that state, is not normative. One might think that because our definition of normativity looks only at denotations and not at terms, this would count as normative, but brief reflection should be enough to convince the reader that this is in general not so.

3.5 Deontic axioms

Since reasoning from a specification Γ involves consideration only of normative life-cycles (ones in which actions performed are permitted and obligations incurred are discharged), permissions default to 'on' and obligations to 'off'. This means that the specifier never has to assert permissions – the logic only takes into consideration models in which actions which occur are permitted. Similarly, the specifier never needs to deny obligations. The effect of this is that permissions should be *denied* in a specification, and obligations should be *asserted*. Typically, deontic axioms will look like

$$\mathrm{per}(\mathrm{a}) \rightarrow \mathit{condition}$$
$$\mathit{condition} \rightarrow \mathrm{obl}(\mathrm{a})$$

More precisely: there should be positive occurrences of obligations and negative occurrences of permissions in deontic axioms.

We say 'should', but the specifier is not barred from writing per(a) ← condition or per(a) ↔ condition. There are however disadvantages in doing so. The problem is that agents which inherit the agent for which we are writing deontic axioms may need to put more constraints on permissions (*i.e.* deny permission in further circumstances) and constraining permission to match a single condition will contradict this. There are examples of this in section 6. If permissions are written in the recommended way, they are easy to 'add-up'. Adding up $\mathrm{per(a)} \rightarrow \mathrm{cond}_1$, $\mathrm{per(a)} \rightarrow \mathrm{cond}_2$, and $\mathrm{per(a)} \rightarrow \mathrm{cond}_3$ yields $\mathrm{per(a)} \rightarrow \mathrm{cond}_1 \wedge \mathrm{cond}_2 \wedge \mathrm{cond}_3$. It should be obvious that the same remarks apply to obligations, except that further axioms *add* obligations instead of removing permissions. Axioms add up similarly: $\mathrm{cond}_1 \rightarrow \mathrm{obl(a)}$ and $\mathrm{cond}_2 \rightarrow \mathrm{obl(a)}$ make $\mathrm{cond}_1 \vee \mathrm{cond}_2 \rightarrow \mathrm{obl(a)}$.

The reader may object that having only negative occurrences of permissions makes it impossible to prove permissions. As this stands this is true, but it *must* be true given the way permissions are defined. We cannot assert permissions because we can not know that they are not going to be denied in agents which inherit the agent in question. But once we have completed the specification we can logically 'close' the deontic axiom, asserting $\mathrm{per(a)} \leftrightarrow \mathrm{cond}_1 \wedge \mathrm{cond}_2 \wedge \mathrm{cond}_3 \ldots$. Then permissions can be proved.

Again, the same remarks apply to obligations with the obvious changes: here we should say: having only positive occurrences of obligations makes it impossible to *use* them in proofs, until we logically close the conjunction of the axioms in the way described.

4 Structure

In Sections 1 and 2 we motivated the idea that agents can be composed in a variety of ways. In this section we will look at this in more detail. The primitive relation between agents is that of *morphism*. If there is a morphism from **X** to **Y**, **X** is identical in behaviour with some sub-agent of **Y**. The morphism is a map between signatures which shows us how to rename the signature elements to demonstrate the sub-agent relation.

A morphism is a map between agent specifications **X** and **Y** taking

- each sort in **X** to a sort in **Y** (the special sorts **action** and **ref**$_s$ are preserved)
- each action symbol in **X** to an action symbol in **Y**
- each attribute symbol in **X** to an attribute symbol in **Y**

such that the translation under the mapping of each theorem of **X** is a theorem of **Y**.

Of course the map must respect the sorts of the action and attribute symbols in the language. A symbol in **X** is mapped on to a symbol in **Y** whose sort is the sort which is mapped on to by the sort of **X**.

The following diagrams show examples of morphisms in the lift specification. **floor** is constructed as an aggregation of two **buttons** (see page 4), an 'up' button and a 'down' button. Each **button** is mapped to **floor**, with morphisms called 'up' and 'down'. As already indicated, we use the dot notation in naming the attributes and actions of **floor**. For example, the attribute 'lit' is mapped by the morphism 'up' to the attribute 'up.lit'. The next morphism is an example of refinement. We have decided that there should be only one light on each floor, which will represent the state of both buttons. When the user presses either of the buttons, the light illuminates and the user is not able to see which of the buttons was pressed. The morphism collapses the distinction between 'up.lit' and 'down.lit'. First we show how the agents are defined, and then the morphisms.

```
agent floor
includes button via up;
includes button via down;
end
```

```
agent floor′

attributes
     s   lit : bool;
includes floor via id where
   up.lit is lit
   down.lit is lit;
end
```

This example also shows the flexibility of the specification language. Given that we have specified **button** as on page 4, these specifications are really just abbreviations for the following more verbose ones:

```
agent floor
attributes
     s   up.lit : bool;
     s   down.lit : bool;
actions
     ℓ   up.press;
     ℓ   down.press;
axioms
     [up.press]up.lit;
     [down.press]down.lit;
end
```

```
agent floor′
attributes
     s   lit : bool;
actions
     ℓ   up.press;
     ℓ   down.press;
axioms
     [up.press]lit;
     [down.press]lit;
end
```

At the theory presentation level, the connections are as follows.

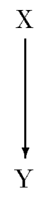
$$\begin{tikzcd}
\text{button} \arrow[dr, "\text{up}"'] & & \text{button} \arrow[dl, dashed, "\text{down}"] \\
& \text{floor} \arrow[d, "\text{id}"] & \\
& \text{floor}' &
\end{tikzcd}$$

4.1 Inheritance and clientship

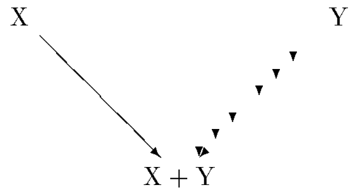
Agents can exist in a variety of relationships one to another. Two important relationships used in object-oriented design are *inheritance* and *clientship.* If **Y** inherits **X**, all the actions and attributes of **X** are also attributes of **Y**, but **Y** may have more besides. For example, consider the abstract agents **vehicle** and **car**. **Car** inherits all the actions and attributes of **vehicle**, but has more of its own besides. Now consider the agent **engine**. We might be tempted to say that **vehicle** inherits all its actions and attributes. This will certainly be true for actions like *start* or attributes like *running.* But we may want to say that some attributes or actions, like the attribute *flooded*, are attributes or actions which apply to the engine alone. It does not make sense to say that the vehicle is flooded — or at least, it is not the same as saying that the engine is flooded. **Vehicle** inherits only some of the actions and attributes of **engine**. This is clientship.

These object-oriented relationships, and many others, are captured by the sub-agent relation. **X** is a sub-agent of **Y** if there is a *morphism* from the specification of **X** to that of **Y**. Roughly speaking, this means that all the actions and attributes of **X** are (under appropriate renaming) actions and attributes of **Y**. We will now describe how three object-oriented constructs are captured by morphisms.

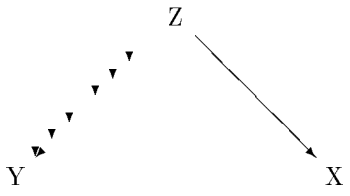
- Inheritance is captured by a single morphism. If **Y** inherits **X** then there is a morphism from the specification of **X** to that of **Y**.
- Aggregation is captured by two morphisms. If $\mathbf{X}+\mathbf{Y}$ is the aggregation of **X** and **Y**, then there are morphisms from **X** to $\mathbf{X}+\mathbf{Y}$ and from **Y** to $\mathbf{X}+\mathbf{Y}$. The aggregation may involve identifying some actions and attributes between **X** and **Y**, in which case we can form an abstraction **Z** of $\mathbf{X}+\mathbf{Y}$.
- Clientship is also captured by two morphisms. If **X** is a client of **Y**, that is to say it inherits selectively from **Y**, then we construct morphisms as follows. Let **Z** be the part of **Y** which is inherited by **X**. Then we have a morphism from **Z** to **X** and one from **Z** to **Y**.



inheritance



aggregation



clientship

5 Locality Axioms

We said in the introduction that the key benefit of structure was the ability to impose a notion of *locality* by which we can control the effects of actions. Without locality the structuring we have introduced would be wasted — all reasoning would have to take place in the biggest agent. But the 'biggest agent' that **button** is a part of is the **lift-system**, or perhaps the whole building or even the street that we are specifying. We would like to conclude properties of **button** or **door** without reference to these big systems.

Examples

Here are examples where attributes and actions should be local to the agents in which they are defined:

1. Only the actions in the agent **door** (*i.e.* 'open' and 'close') can affect the attribute 'posn'. This is locality for *attributes*:

   Only the actions of X can affect the attributes of X

2. The actions in agent **button** (*i.e.* 'press') can only affect the attribute 'lit'. This is locality for *actions*:

   The actions of X can *only* affect the attributes of X

Non-examples

There are instances where locality fails, showing why some actions and attributes *must* be sharable to allow the interaction between agents we want. For example, the axiom $(\text{floor} = 1) \rightarrow [\text{open}]\neg 1.\text{lit}$ in the agent **lift**, which says the light at the current floor is extinguished by the door opening, violates both types of locality. The action 'open' has to affect the attribute 'lit' which is declared outside the agent in which 'open' is declared; and the attribute 'lit' is changed by an action (open) declared outside the agent in which it is declared.

We need to have locality for some actions and attributes and not for others. In the specification language we have a mechanism for stating which action and attribute symbols are local and which are not. At the theory presentation level, this means that some attributes and some actions have a locality axiom. But as we are about to see, within an agent community there are degrees of sharability.

Locality axioms

Remember that the specifications of agents indicate whether the actions and attributes are local or sharable. For example, the attribute 'lit' is sharable in **button**, because as we have remarked it will be updated by the action 'open' in **door**. But the action 'press' in **button** is local. And 'lit' (or 'i.lit' as it is then known) in **lift** is local, because it will not be updated by any actions introduced in ever-bigger agents. Similarly, 'open' in **door** is sharable (it has to update i.lit), and it remains sharable in **lift** because it will have more work to do in **lift-system**. (It will have to extinguish the floor lights.)

Local attribute or action symbols have a locality axiom in the theory presentation denoted by the agent specification.

For example, 'posn' is a local attribute. The locality axiom is

$$\forall a : \textbf{action}, \forall v : (\text{op,cl})\ \ (a = \text{open} \vee a = \text{close} \vee (\text{posn} = v \rightarrow [a](\text{posn} = v))).$$

It means that no action other than 'open' and 'close' can affect the value of 'posn'; for either an action is 'open' or 'close', or the value of posn is unchanged by the action.

'Press' is a local action symbol in **button**: its locality axiom is

$$\forall \alpha : \textbf{ref}_{\text{bool}}(\alpha = \&\text{lit} \vee ((*\alpha = v) \rightarrow [\text{press}](*\alpha = v))$$

which means: no attribute other than the attribute 'lit' can be affected by the action 'press'.

General form

We will now show the general form of locality axioms. Unfortunately they look rather ugly in this form, but the purpose of this section is to show that for any action or attribute symbol in an agent we can construct a locality axiom.

Suppose A is an attribute symbol local to an agent with action symbols $a_1, \ldots, a_n$. The locality axiom should say that every action which occurs is either an instance of one of these actions or leaves A unchanged.

$$\forall \mathbf{a} : \textbf{action} \bigvee_{i=1}^{n} (\exists \underline{x_i}.\ \ (\mathbf{a} = a_i(\underline{x_i})))\ \ \vee\ \ \forall \underline{y}, v.\ \ (A(\underline{y}) = v) \rightarrow [\mathbf{a}](A(\underline{y}) = v))$$

The axiom says: either **a** is one of the actions of the agent with appropriate parameters, or it leaves the value of the attribute (with any parameters) unchanged. We have used $\underline{x_i}$ to mean a vector of variables. The number and sort of these variables is determined by the sort of the action symbol for each value of i. Similarly, $\underline{y}$ is a vector of variables whose sorts match the parameters of A.

Action locality: Now suppose the action symbol a is local to an agent with the following attribute symbols:

$$A_1^{s_1}, A_2^{s_1}, \ldots, A_{n_{s_1}}^{s_1}, A_1^{s_2}, A_2^{s_2}, \ldots, A_{n_{s_2}}^{s_2}, \ldots$$

For each sort s, we suppose that there are n_s attribute symbols with values of the sort s. Then for each sort s and each reference term α of sort $\textbf{ref}_s$, *either* α is the reference of one of the parameterised attribute symbols with values of sort s, *or* **a** (with any parameters) leaves the value of α unchanged.

The schematic form of the locality axiom is thus:

$$\bigwedge_{s\in S} (\forall\alpha : \mathbf{ref}_s \bigvee_{i=1}^{n_s} \exists \underline{x_i}.\, \alpha = \&A_i^s(\underline{x_i})) \quad \vee \quad \forall \underline{y}, v : s.\,((*\alpha = v) \to [a(\underline{y})](*\alpha = v))$$

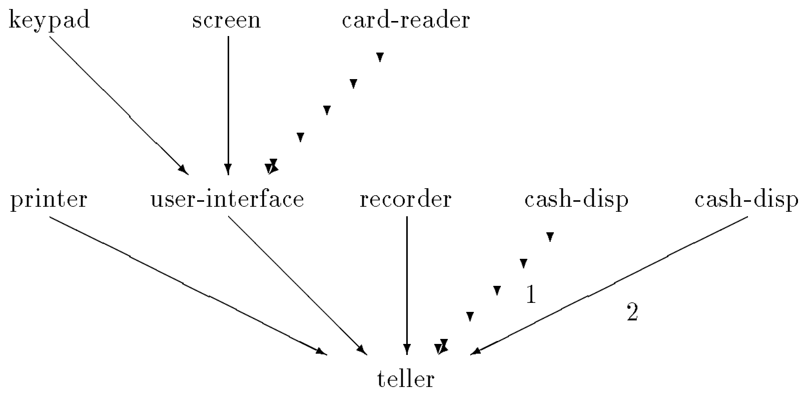
These axioms are complicated, but it is worth repeating that the specifier does not have to write them down or ever use the operators $*$ and &. Locality axioms are generated by the specification 'compiler' on the basis of the locality declarations the specifier makes.

To understand their effect, the crucial point about locality axioms is that (like all axioms) they are inherited by agents via morphisms. But the scope of the quantifiers $\forall\alpha : \mathbf{ref}$ and $\forall a : \mathbf{action}$ increases as we move to bigger and bigger agents. Notice that the natural requirement that morphisms cannot map *local* symbols to *sharable* ones is enforced by the fact that the locality axioms are inherited under morphisms.

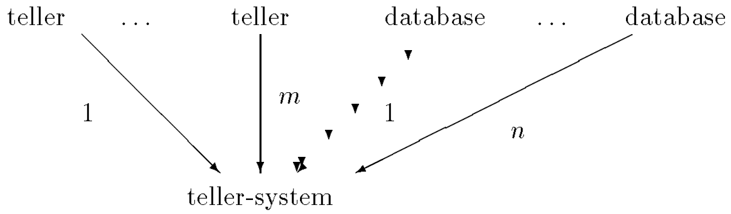
6 Auto-teller examples

The following example, of an autoteller system in a bank, shows some of the ideas in this paper in action. An autoteller (cash-dispenser) allows bank customers to withdraw money from their accounts by inserting a plastic card and typing a personal identification number at the keypad. The machine compares the identification number with the number magnetically coded on the card. It asks the customer the amount of cash required (by putting an appropriate message on its VDU screen) and, after checking one of the bank's databases, dispenses the cash. It records the transaction on an internal recorder, and prints the transaction on a chit which it gives to the customer. It has an internal hopper in which it can retain the card if the customer consistently types the wrong identification number.

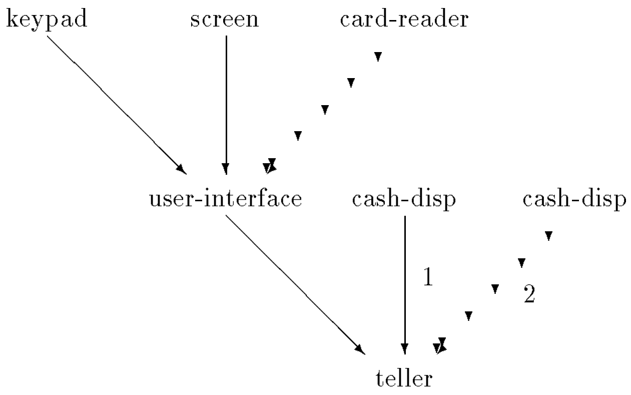
From the English-language specification we identify the following components:



The user-interface is made up from the keypad, the screen, and the card-reader. In turn the teller is made up from the printer, the user-interface, the recorder and (say) two cash-dispensers containing notes of different denominations. A teller-system consists of (say) m tellers and n databases.



But here we will deal with just the following fragment of these diagrams:



We are now in a position to axiomatise the agents.

agent keypad
attributes
s waiting : bool
ℓ last-no : nat
actions
s enter-no(n : nat)
s cancel
axioms
1. waiting $\rightarrow$ [enter-no(n)](last-no $= n \wedge \neg$waiting)
2. per(enter-no(n)) $\rightarrow$ waiting
end

A keypad has two attributes which define its state, waiting and last-no. Waiting specifies whether the agent is in a state of accepting a numerical input on its keypad. Last-no contains the last number typed. The agent keypad also has two actions, enter-no(n) (the user enters the number n), and cancel (the user elects to abort the transaction – we imagine that the cancel button is part of the keypad). Notice that the keypad does not, and should not, distinguish between the various types of numbers entered (identification numbers, amounts etc).

The keypad's first axiom states that last-no is updated by the action enter-no(n) if the agent is in the waiting state, and also that it then ceases to be in the waiting state. What happens if the agent is not in the waiting state is not defined, although the second axiom states that an occurrence of the action enter-no(n) would then be non-normative.

The keypad has a complex interface with the other components, as can be seen by the fact that both actions and one attribute are shared. The attribute waiting is shared because it will be updated by actions in the user-interface. Similarly, enter-no(n) and cancel are sharable because they have to update attributes there too.

Now for the card-reader:

```
agent card-reader
attributes
    $\ell$  has-card : bool
    $\ell$  card : (r, u)
    $\ell$  card-no : nat
actions
    $s$  accept-r($n$ : nat)
    $\ell$  accept-u
    $\ell$  return
    $\ell$  keep
axioms
    1.  has-card $\rightarrow$ obl(return) $\vee$ obl(keep);
    2.  [accept-r($n$)](card-no $=$ $n$ $\wedge$ has-card $\wedge$ card $=$ r);
    3.  [accept-u](has-card $\wedge$ card $=$ u);
    4.  [return]$\neg$has-card;
    5.  [keep]$\neg$has-card;
    6.  per(return) $\vee$ per(keep) $\rightarrow$ has-card ;
    7.  per(accept-r($n$)) $\vee$ per(accept-u) $\rightarrow$ $\neg$has-card;
end
```

The card-reader has three local attributes which respectively indicate: whether there is a card inside it; whether the card inside (if any) is readable or unreadable; and the personal identification number coded on the magnetic strip of the card. Notice that (r, u) is an enumerated sort. The card-reader can accept a readable card with number n (accept-r(n)), or accept an unreadable card (perhaps one which is damaged or wrongly inserted) (accept-u). It can then return or keep the card. Of these actions, only accept-r(n) need be sharable; it has to update attributes of the user-interface.

The axioms say the following:

1. If the reader has a card, it has an obligation to keep the card or to return it. That is, a life-cycle is non-normative if it has a state in which has-card is true but which is not eventually followed by a return action or a keep action.

2 and 3. After accepting a readable card, the appropriate attributes are updated.

4 and 5. After keeping (transferring to internal hopper) or returning a card there is no longer a card.

6. The agent may keep or return a card only if it has one...

7. ...and accept one only if it does not have one.

The last of the agents which make up the user-interface is the screen. It has just one sharable attribute, the message which it displays. Clearly it has to be sharable; if it was local, it would never get updated as there are no actions in that agent.

```
    agent screen
    attributes
        $s$   message : string
    end
```

Now we are in a position to tie these components together in the user-interface.

```
agent user-interface
attributes
    $\ell$   entering-id;
    $\ell$   entering-amt;
actions
    $\ell$   request-amt;
includes keypad
includes card-reader
includes screen
axioms
    1.  $\neg$(entering-id $\wedge$ entering-amt);
    2.  entering-id $\rightarrow$ (message=‘Enter id’ $\wedge$ $\exists n$ obl(enter-no($n$)) $\vee$ obl(cancel));
    3.  entering-amt $\rightarrow$ (message=‘Enter amount’ $\wedge$ $\exists n$ obl(enter-no($n$)) $\vee$ obl(cancel));
    4.  [accept-r($n$)](entering-id $\wedge$ waiting);
    5.  last-no=card-no $\wedge$ entering-id $\wedge$ $\neg$waiting $\rightarrow$ obl(request-amt);
    6.  [request-amt](entering-amt $\wedge$ waiting);
    7.  [cancel]obl(return)
end
```

The user-interface includes all the actions and attributes and axioms of its constituents. It also has the attributes entering-id and entering-amt and the action request-amt, all of which are local. The purpose of these is to guide it through the sequence of events to do with verifying the card and dispensing the cash. The axioms mean:

1. The machine is never at once processing the identifier and the amount.
2. While expecting the customer identifier, the screen displays the appropriate message and a number must be entered on the keypad.

   The clause $\exists n$ obl(enter-no(n)) means that there is an obligation to enter *a* number, but not any specific number. In other words, the user fulfills this obligation by typing an arbitrary number. The reader may consider how one could specify an obligation to enter some specific number.
3. Similarly when expecting an amount.
4. After accepting a readable card the machine enters the expecting-identifier state and activates the keypad

5 and 6. If the machine is processing the identifier and the customer has typed a number which was found to be correct, the machine enters the processing-amount state.

7. If the cancel button is pressed the machine must return the card.

The cash dispenser dispenses cash if it has enough left:

```
agent cash-disp
attributes
    $\ell$  qty-left : nat;
actions
    $\ell$  dispense($q$ : nat);
axioms
    $x$ = qty-left $\wedge$ $x > q$ $\rightarrow$ [dispense($q$)](qty-left = $x - q$);
    per(disp($q$)) $\rightarrow$ $q >$ qty-left
end
```

The auto-teller itself is left with the rather boring task of deciding how much of each denomination is to be dispensed. Note that the dispensers are obliged to dispense if (and permitted only if) the right conditions are met. Recall the form of deontic axioms described in section 3.5.

```
agent teller
includes user-interface
includes cash-disp via 1
includes cash-disp via 2
axioms
    entering-amt $\wedge$ $\neg$waiting $\wedge$
      $n_1$ = last-no div denom$_1$ $\wedge$ $n_2$ = (last-no mod denom$_1$) div denom$_2$ $\rightarrow$
      obl(1.dispense($n_1$)) $\wedge$ obl(2.dispense($n_2$));
    per(1.dispense($n_1$)) $\vee$ per(2.dispense($n_2$)) $\rightarrow$
      entering-amt $\wedge$ $\neg$waiting $\wedge$
      $n_1$ = last-no div denom$_1$ $\wedge$ $n_2$ = (last-no mod denom$_1$) div denom$_2$;
end
```

One perhaps counter-intuitive feature of the logic which this example brings out is the relative weakness of the locality axioms compared with the frame rule. For example, we cannot deduce from the axioms of the card-reader that the return action does not affect the card-no attribute. That is because they are in the same agent and there is no locality axiom to that effect. The important points here are that (i) it does not matter – we do not need the assumption that return does not affect card-no; and (ii), that an implementation which chose to reset card-no after returning the card to some null value would be a perfectly good implementation, even a likely one.

The structuring principle says that whenever we need to make such constraints we must do so by structuring the agents in a way which gives rise to the right locality axioms.

Acknowledgments

This work is one of the results of the SERC FOREST Project at Imperial College, London. Mark Ryan is funded by that project. José Fiadeiro is on leave from Departamento de Matematica, Instituto Superior Tecnico, Lisboa, Portugal, as a grantee of the Commission of the European Communities.

References

[1] J. Fiadeiro and T. Maibaum. Describing, structuring and implementing objects. In *Proc. REX Workshop on Foundations of Object-Oriented Languages*. Springer-Verlag, in print.

[2] J. Goguen. A categorial manifesto. Technical Report PRG-72, Programming Research Group, University of Oxford, March 1989.

[3] R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes, 1987.

[4] S. Khosla and T. S. E. Maibaum. The perscription and description of state based systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification.* 1989. Lecture Notes in Computer Science 398.

[5] T. S. E. Maibaum. A logic for the formal requirements specification of real-time embedded systems. Technical report, Imperial College, London, 1987. Deliverable R3 for FOREST (Alvey).

[6] Mark Ryan. Defaults and revision in structured theories. In *Logic in Computer Science (LICS)*, July 1991.