On Dynamic Aspects of OOD Frameworks in Component-based Software Development in Computational Logic

Juliana Küster Filipe,[1*] Kung-Kiu Lau,[2] Mario Ornaghi,[3] and Hirokazu Yatsu[4]

[1] Abt. Informationssysteme, Informatik, Technische Universität Braunschweig,
Postfach 3329, D-38023 Braunschweig, Germany
J.Kuester-Filipe@tu-bs.de

[2] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

[3] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

[4] Information Technologies & Services Division
Technologies Research & Development
Nihon Unisys Ltd., 1-1-1 Toyosu, Koto-ku, Tokyo 135-8560, Japan
Hirokazu.Yatsu@unisys.co.jp

Abstract. In component-based software development, object-oriented design (OOD) frameworks are increasingly recognised as better units of reuse than objects. This is because OOD frameworks are groups of interacting objects, and as such they can better reflect practical systems in which objects tend to have more than one role in more than one context. In an earlier paper, we described a formal semantics of the static aspects of OOD frameworks in computational logic. In this paper, we make a preliminary attempt to extend this to the dynamic aspects.

1 Introduction

In [12, 13] we present a formalisation of Object-Oriented Design (OOD) frameworks in component-based software development (CBD) in computational logic. Such frameworks are groups of (interacting) objects. For example, in the CBD methodology *Catalysis* [6], a driver may be represented as the OOD framework shown in Figure 1.[1] A driver is a person who drives a car, or in OOD termi-

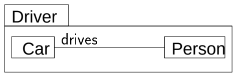

Fig. 1. The Driver OOD framework.

* The first author was supported by the DFG under Eh 75/11-2 and partially by the EU under ESPRIT-IV WG 22704 ASPIRE.

[1] *Catalysis* uses the UML notation, see e.g. [18].

nology, a driver is a framework composed of a car object and a person object, linked by a 'drives' association (or attribute).

OOD frameworks are increasingly recognised as better units of reuse in software development than objects (see e.g. [9, 15]). The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and OOD frameworks can capture this, whereas existing OOD methods (e.g. Fusion [4] and Syntropy [5]) cannot. The latter use classes or objects as the basic unit of design or reuse, and are based on the traditional view of an object, as shown in Figure 2, which regards an object as a closed entity with one fixed role. On the other hand, OOD frameworks allow objects that play different

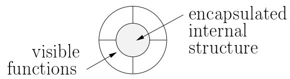

Fig. 2. Traditional view of an object.

roles in different frameworks to be composed by composing OOD frameworks. In *Catalysis*, for instance, this is depicted in Figure 3.

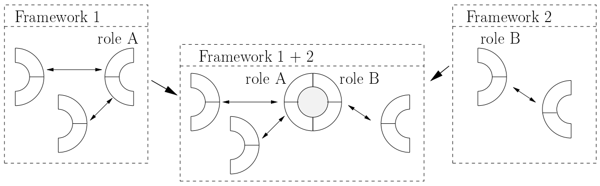

Fig. 3. Objects by composing OOD frameworks.

For example, a person can play the roles of a driver and of a guest at a motel simultaneously. These roles are shown separately in the PersonAsDriver and PersonAsGuest OOD frameworks in Figure 4. If we compose these two

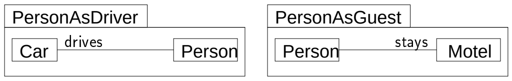

Fig. 4. PersonAsDriver and PersonAsGuest OOD frameworks.

frameworks, then we get the PersonAsDriverGuest OOD framework as shown in Figure 5. In this OOD framework, a person object plays two roles, and is a

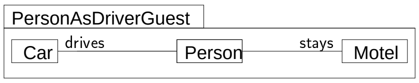

Fig. 5. PersonAsDriverGuest OOD framework.

composite object of the kind depicted in Figure 3.

OOD frameworks should play a crucial role in next-generation CDB methodologies. Our intention is to endow them with suitable (declarative) semantics for CBD in computational Logic.

2 Specification Frameworks

In [13, 12] we describe the static semantics of OOD frameworks, i.e. one without time or state transitions. This semantics is based on *specification frameworks* (or *frameworks* for short), which we briefly describe in this section. In the next section, we will introduce classes and OOD frameworks.

A (*specification*) *framework* $\mathcal{F} = \langle \Sigma, X \rangle$ is defined in the context of first-order logic with identity. It is composed of a signature Σ (containing sort symbols, function declarations and relation declarations), and a finite or recursive set X of Σ-axioms. The main purpose of a framework is to axiomatise a problem domain and to reason about it. More specifically, a framework contains the ADT's and the concepts needed to build a (possibly, though not necessarily, object-oriented) model of the application at hand.

A framework is thus a (first-order) theory, and we choose its intended model to be a *reachable isoinitial model*, defined as follows:[2]

Definition 1. Let X be a set of Σ-axioms. A Σ-structure i is an *isoinitial model* of X iff, for every other model m of X, there is one isomorphic embedding $i : \mathsf{i} \to \mathsf{m}$.

A model i is *reachable* if its elements can be represented by ground terms.

We distinguish between *closed* and *open* frameworks. A framework $\mathcal{F} = \langle \Sigma, X \rangle$ is *closed* iff there is a reachable isoinitial model i of X. An *open* framework $\mathcal{F}(\Omega) = \langle \Sigma, X \rangle$ does not have an isoinitial model, since its axioms leave open the meaning of some symbols Ω of the signature, that we call *open* symbols. Non-open symbols are called *defined* symbols.

An open framework $\mathcal{F}(\Omega) = \langle \Sigma, X \rangle$ can be *closed*, i.e made into closed frameworks, by *instantiating* the symbols of Ω by new axioms, that fix their meaning. The new axioms are called *closure axioms*, and a set of closure axioms is called a *closure*. To guarantee that a closure works, i.e., it is consistent and gives rise to a closed framework, we use suitable *constraints* (see [13]) and we assume that a constraint satisfaction relation is defined, at least for closures of a suitable form. In this way, for a constrained open framework, every closure that satisfies the constraints gives rise to a closed framework.

In this paper, we will consider closure by *internalisation*, as defined in [13]. As shown in [13], internalisation can be used to implement constrained parameter passing, as well as to introduce *objects* as the closures of suitable open frameworks that represent *classes*.

Let $\mathcal{F}(\Omega) = \langle \Sigma + \Delta, X \rangle$ be an open framework with a subsignature Δ that does not contain symbols of Ω. A Δ-*internalisation* is a closure operation that may change the signature or add new axioms, as follows:

- *Sort closure*. The closure: CLOSE S BY s
  renames the open *sort* S by a sort s of Δ. No axioms are added.

[2] See [13] for a justification of this choice and [2, 11] for a discussion of isoinitial theories.

– *Relation closure.* The operation: $\textsc{close}\ r\ \textsc{by}\ \forall x\,.\,r(x) \leftrightarrow R(x)$ introduces the explicit definition $\forall x\,.\;r(x) \leftrightarrow R(x)$ of the *relation* r as a new axiom. The declaration of r may contain only sorts of Δ, and $R(x)$ is a quantifier free Δ-formula.
– *Function closure.* The operation: $\textsc{close}\ f\ \textsc{by}\ \forall x\,.\,y = f(x) \leftrightarrow F(x,y)$ introduces the explicit definition $\forall x\,.\;y = f(x) \leftrightarrow F(x,y)$ of the *function* f as a new axiom. The declaration of f may contain only sorts of Δ, and $F(x,y)$ is a quantifier-free Δ-formula such that $X|\Delta \vdash \forall x \exists! y\,.\,F(x,y)$.[3]

For Δ-internalisation, constraints are a distinguished subset of the axioms. Thus we represent an open constrained framework as $\mathcal{F}(\Omega) = \langle \Sigma + \Delta, X \cup \mathit{Constr}\rangle$, where *Constr* is the (distinguished) set of constraints. A Δ-internalisation of Ω is an internalisation of all the symbols of Ω. In general, it closes the open symbols by a set Cl of explicit definitions. We say that the constraints are *satisfied* if

$$X|\Delta \cup Cl \vdash \mathit{Constr}.$$

Of course, if some open sort has been internalised, then we assume that the corresponding renaming has been applied to Cl and to the constraints.

Example 1. The following framework imports the closed framework $\mathcal{NAT}$ of natural numbers, and introduces two new symbols q and r.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{Framework}\ \mathcal{NAT}(q : [Nat, Nat]);} \\
\textsc{import:} & \mathcal{NAT}; \\
\textsc{decls:} & q, r : [Nat, Nat]; \\
\textsc{axioms:} & r(x,y) \leftrightarrow q(y,x); \\
 & r(x,y) \rightarrow x < y; \\
\textsc{constrs:} & q(x,y) \rightarrow x < y.
\end{array}
$$

In the first axiom, $q(x,y)$ is open (since it is declared as a parameter) and is used to define $r(x,y)$. Since it is open, we need to consider suitable closures for q. (Here we consider $\Sigma_{\mathcal{NAT}}$-closures, where $\Sigma_{\mathcal{NAT}}$ is the signature of $\mathcal{NAT}$.) To avoid closures that are inconsistent with respect to the second axiom, we have introduced the constraint $q(x,y) \rightarrow x < y$. The closure $q(x,y) \leftrightarrow y = s(s(x))$, for example, of q satisfies the constraint $q(x,y) \rightarrow x < y$, which can be proved using the closure itself and the axioms of $\mathcal{NAT}$.

3 Static Aspects of OOD Frameworks

As we have seen in Section 1, OOD frameworks are composite objects/classes. Here, we define OOD frameworks and objects/classes in terms of specification frameworks. Although it is based on [12, 13], the material here is mostly new in order that we can deal with dynamic aspects of OOD frameworks later on.

When we build a class or an OOD framework, we start from a specification framework $\mathcal{P} = \langle \Sigma_P, X_P \rangle$ that axiomatises the problem domain. For convenience, we will call $\mathcal{P}$ a *problem framework*.

[3] $X|\Delta$ is the restriction of X to the Δ-formulas.

Example 2. The Car class in Figures 4 and 5 can be defined in the context of a problem framework $\mathcal{CAR}$ that contains the data types for strings and integers, enumerated types to indicate (for example) the possible uses of cars, and so on. Thus it will contain data-type sorts (like Int for integers, Str for strings), data-type operations and predicates (like integer sum $+$, list concatenation $conc$, (overloaded) ordering $<$ on lists and strings), as well as sorts and operations from the problem domain, like the sort $Opts$ of possible optional features.

3.1 Classes

For simplicity, we will assume that our classes are based on a fixed closed problem framework $\mathcal{P} = \langle \Sigma_P, X_P \rangle$. As usual, a class is interpreted as the collection of objects that are instances of the class. A class has *attributes*, that represent dynamic properties of the objects of the class, and *actions*, that represent methods used for updating the values of the attributes of the objects of the class.[4]

Therefore, a problem signature Σ_P will be enriched by the special sort Act of possible *action states*, the special sort Obj of *object identities* (as well as a denumerable set of constants of sort Obj), and the special predicates $\rhd : [Act]$ and $\odot : [Act]$. Intuitively, for an action a, $\rhd a$ means that a is enabled, and $\odot a$ that it has just occurred. The intended meaning of the special symbols is fixed by the following special axioms:

- The axioms X_{Act}: $\exists! x : Act\,.\,(\rhd x \wedge \odot x)$
  $\exists! x : Act\,.\,(\rhd x \wedge \neg \odot x)$
  $\exists! x : Act\,.\,(\neg \rhd x \wedge \odot x)$
  $\exists! x : Act\,.\,(\neg \rhd x \wedge \neg \odot x)$
  This small theory has a (non-reachable) isoinitial model where the sort Act of action states contains exactly four elements, corresponding to the possible truth combinations of $\rhd$ and $\odot$. This means that, at each time, for an action a there are exactly four possibilities: a may be currently enabled or not, and it may have just occurred (in the transition from the previous state) or not. At the dynamic level, this meaning is enforced by the conditions to be satisfied by $\mathcal{F}$-labelled event structures, in Definition 3.
- The axioms X_{Obj} that contains the axioms $\neg o = o'$, for every pair of distinct constants o and o' of sort Obj. The isoinitial model of X_{Obj} has the set of its constants as domain. Constants of sort Obj are called *object identities* and will be used to (uniquely) name objects (and framework instances, see later).

We will always assume that the signature and axioms of X_{Act} and X_{Obj} are understood.

Besides these axioms, class axioms may also (explicitly) contain:

- A set ATB of attribute declarations. An attribute declaration introduces a new function or relation symbol using sorts from Σ_P.

[4] Actions can also effect inter-object communication since an object may invoke actions in another object.

- A set ACT of action declarations. An action declaration has the form $a : A \to Act$, where the (possibly empty) arity A contains sorts of Σ_P.
- A set *Constr* of attribute and action constraints, where an attribute constraint belongs to the language generated by $\Sigma_P{+}ATB$ (and contains at least one attribute), while an action constraint belongs to the language generated by $\Sigma_P{+}ATB{+}ACT$ (and contains at least one action).

Thus the signature of a class is $\Sigma_P{+}ATB{+}ACT$, where Σ_P contains the problem signature, the special sorts Act and Obj, the predicates $\rhd$, $\odot$, and the object identities. The symbols of Σ_P are closed by their axioms, while $ATB{+}ACT$ are open.

Example 3. In the context of the problem framework $\mathcal{CAR}$, the class Car in Figures 4 and 5 can be (statically) specified as follows:

Class $\mathcal{CAR}$;

ATTRIBUTES:	$.km : [\,] \to Int;$
	$.option : [Opts];$
ACTIONS:	$.go : [Int] \to Act;$
CONSTRS:	$.km \geq 0;$
	$\forall x \,.\, (.km + x < 100000 \to \rhd .go(x));$
	$\forall x, y \,.\, (\odot(.go(x)) \wedge \odot(.go(y)) \to x = y).$

$.km$ and $.option$ are attributes, representing the odometer reading and the car's optional features respectively. At the class level, they are open. In a car object, they will be closed to indicate the (current) odometer reading and the optional features of the car.

$.go$ is an (open) action symbol. In the current state of a car object, $go(x)$ may be enabled or have just occurred, for suitable values of x.

$.km$ must have positive values, as stipulated by the first constraint. The second constraint says that $go(x)$ is always enabled if the final odometer reading does not exceed 100000, while the third one says that $go(x)$ cannot have occurred with different x and y at the same time.

The dynamic meaning of attributes and actions will be explained in the next section. Here we are concerned only with their static meaning, namely with the way of fixing them in a state. To this end, we will use suitable closure axioms that we call *state axioms*.

The closure of *attributes* is obtained through internalisation in $\mathcal{P}$, as explained in the previous section. An attribute closure will be indicated by $Cl(atb)$.

The closure of an *action* $a : A \to Act$ has the following general form:

$$\forall((\rhd a(x) \leftrightarrow E(x)) \wedge (\odot a(x) \leftrightarrow H(x)))$$

where $E(x)$ and $H(x)$ are quantifier-free formulas that may contain symbols of Σ_P and attributes. A closure of all the actions will be denoted by $Cl(act)$.

We allow constraints on actions and attributes. Let *Constr* be such constraints. Then they must be proved by the closures of attributes and actions, i.e.:

$$X_P \cup X_{Act} \cup X_{Obj} \cup Cl(atb) \cup Cl(act) \vdash Constr. \quad (1)$$

The possible closures of a class by state axioms represents its possible objects. Moreover, each object has its own identity, represented by a unique name o of sort *Obj*, and its own language, obtained by prefixing attributes and actions by o. The closures $Cl(atb)$ and $Cl(act)$ for an object o, must be given in the language of o. Thus all the objects share the signature Σ_P, while different objects have disjoint attribute and action signatures. Constraint satisfaction (1) must be checked by each object, in its own language.

Example 4. An object *spider* of class $\mathcal{CAR}$ is given, for example, by the following state axioms:

$$
\begin{array}{lll}
\text{NEW } spider : \mathcal{CAR}; & & \\
\text{CLOSE: } \; spider.km & \text{BY} & spider.km = 25000; \\
\quad spider.option & \text{BY} & \forall(spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond); \\
\quad spider.go(x) & \text{BY} & \forall(\triangleright spider.go(x) \leftrightarrow x \leq 80000); \\
 & & \forall(\odot spider.go(x) \leftrightarrow x = 12500).
\end{array}
$$

Constraints are satisfied in the *spider* language because, in the context of the problem framework $\mathcal{CAR}$, we have:

$$
\begin{array}{l}
spider.km = 25000 \vdash spider.km \geq 0; \\
spider.km = 25000 \wedge \forall(\triangleright spider.go(x) \leftrightarrow x \leq 80000) \vdash \\
\qquad\qquad \forall(spider.km + x < 100000 \rightarrow \triangleright spider.go(x)); \\
\forall(\odot spider.go(x) \leftrightarrow x = 12500) \vdash \forall(\odot spider.go(x) \wedge \odot spider.go(y) \rightarrow x = y);
\end{array}
$$

State axioms can be updated, i.e. an object is a dynamic entity. As we will see in Section 4, updating is the result of actions.

3.2 OOD Frameworks

An OOD framework is a composition of many classes or simpler OOD frameworks, i.e., it is of the form $\mathcal{F} = comp(\mathcal{F}_1, \ldots, \mathcal{F}_k)$, where *comp* is a composition operation, and $\mathcal{F}_1, \ldots, \mathcal{F}_k$ are classes or sub-frameworks (of a lower nesting level, see below). $\mathcal{F}$ itself may have attributes and/or actions that are distinct from those in $\mathcal{F}_1, \ldots, \mathcal{F}_k$. We model $\mathcal{F}$ as a *container* which contains not only the classes in $\mathcal{F}_1, \ldots, \mathcal{F}_k$ but also a special class, that we call the *container class* of $\mathcal{F}$,[5] and (possibly) with its own attributes and actions.

Thus, whereas a class has objects as instances, an OOD framework $\mathcal{F}$ has *systems* of objects as instances, where each object is an instance of a class contained in $\mathcal{F}$, and objects can be created (and deleted) and updated dynamically.

[5] We will use the same name $\mathcal{F}$ to denote both the framework and its container class.

For simplicity, we will assume that all the classes contained in an OOD framework are based on the same problem framework $\mathcal{P}$. As stated in the previous section, $\mathcal{P}$ always contains the theory X_{act} of action states, and the theory X_{Obj} of object identities. Since $\mathcal{P}$ is always the same for a given OOD framework, we will assume that it is understood. For an OOD framework $\mathcal{F} = comp(\mathcal{F}_1, \ldots, \mathcal{F}_k)$, we need to distinguish between the (static) *framework language* and the associated (dynamic) *system languages*. The framework language is given by the languages of the contained classes. By contrast, the system language dynamically depends on the languages of the objects that belong to the system at any given time. We define the (static) framework language, (dynamic) system languages, and framework instances inductively, starting from frameworks of level 0.

An OOD framework of level 0 is a class $\mathcal{C}$, seen as a container of one class $\mathcal{C}$. Its static framework language contains Σ_P, the attributes and actions of $\mathcal{C}$, and the special *class predicate* $.C : [Obj]$, where $.C(x)$ is true if x is an object in the current system instance of class $\mathcal{C}$.

The system languages associated with a class $\mathcal{C}$ are defined through the instances of $\mathcal{C}$, as follows. For every identity $o : Obj$, we may have a system instance $o.\mathcal{C}$, with the following (system) language:

- For every attribute $.atb$ of $\mathcal{C}$, $o.atb$ is an attribute of $o.\mathcal{C}$.
- For every action $.act$ of $\mathcal{C}$, $o.act$ is an action of $o.\mathcal{C}$.
- $o.C : [Obj]$ is the special class predicate of the instance.

An instance $o.\mathcal{C}$ also has its *closure axioms*, which are a closure for $\mathcal{C}$ in the language renamed by prefixing the identity o to attributes and actions. $o.\mathcal{C}$ is an *object of class* $\mathcal{C}$ with *identity* o. If an instance $o.\mathcal{C}$ exists, then the special class predicate is closed by the axiom $o.C(x) \leftrightarrow x = o$.

As we can see, instances and their languages depend on the corresponding objects.

An OOD framework of level $(i + 1)$ is a framework $\mathcal{F} = comp(\mathcal{F}_1, \ldots, \mathcal{F}_n)$, where i is the maximum of the levels of $\mathcal{F}_1, \ldots, \mathcal{F}_n$. The framework language of $\mathcal{F}$ is the union of the languages of its classes, that are determined as follows:

- There is an indexed set of injective framework morphisms $h_j : \mathcal{F}_j \to \mathcal{F}$, for $1 \leq j \leq n$, where h_j is a signature morphism (possibly) renaming attributes and actions, and the h_j-translations of the constraints of $\mathcal{F}_j$ become constraints of $\mathcal{F}$. These morphisms depend on the kind of composition *comp*.
- The classes of $\mathcal{F}$ are the container class $\mathcal{F}$ and the (h_j images of the) classes of $\mathcal{F}_j$, for $1 \leq j \leq n$. Suitable renamings are applied by h_j (when necessary), to give a unique name to every class of $\mathcal{F}$.
- For every class $\mathcal{C}$ of $\mathcal{F}$, the predicate $.C : [Obj]$ is an attribute of $\mathcal{F}$ (the attribute $.F : [Obj]$ is always included).
- $\mathcal{F}$ may export only some of its classes; selective exporting, i.e., exporting of a selected subset of attributes and actions, is allowed. Non-exported symbols are said to be hidden. Thus we have a visible framework language that contains, at each level, only the part exported by the lower levels, and a full framework language that contains also the hidden parts.

– A framework may contain its own constraints, in the container class. Constraints can be given using only the visible language, while (part of) the full language will be used to define the system instances.

Example 5. Consider the frameworks of level 0 (i.e., classes) $\mathcal{CAR}$ (defined in Example 3), $\mathcal{PERSON}$ (defined below) and $\mathcal{SHOP}$ (sketched below).

Class $\mathcal{PERSON}$;

ATTRIBUTES: $.name : [\,] \to Str$;
$.pocket : [\,] \to Int$;

CONSTRS: $.pocket \geq 0$.

Class $\mathcal{SHOP}$;

ATTRIBUTES: ...

ACTIONS: ...

CONSTRS: ...

We can compose $\mathcal{PERSON}$ and $\mathcal{SHOP}$ into a framework $\mathcal{CONSUMER}$ of level 1.

Framework $\mathcal{CONSUMER}$;

IMPORT: $\mathcal{PERSON}$, $\mathcal{SHOP}$;

ATTRIBUTES: $.shops : [Obj]$;

ACTIONS: $.buy : [Int] \to Act$;

CONSTRS: $.CONSUMER(x) \to .PERSON(x)$;
$\triangleright .buy(x) \to .pocket \geq x$;
$.shops(x) \to .SHOP(x)$.

The classes of $\mathcal{CONSUMER}$ are $\mathcal{PERSON}$, $\mathcal{SHOP}$ and $\mathcal{CONSUMER}$. The class predicates $.PERSON : [Obj]$, $.SHOP : [Obj]$ and $.CONSUMER : [Obj]$ are understood. The first constraint says that a consumer must be a person, i.e., $\mathcal{CONSUMER}$ is a subclass of $\mathcal{PERSON}$. Therefore it inherits the attributes and constraints of $\mathcal{PERSON}$.

We can compose $\mathcal{PERSON}$ and $\mathcal{CAR}$ into a framework $\mathcal{DRIVER}$ of level 1.

Framework $\mathcal{DRIVER}$;

IMPORT: $\mathcal{PERSON}$, $\mathcal{CAR}$;

ATTRIBUTES: $.maydrive : [Obj]$;

ACTIONS: $.drives : [Obj, Int] \to Act$;

CONSTRS: $.DRIVER(x) \to .PERSON(x)$;
$.maydrive(c) \to .CAR(c)$;
$\triangleright .drives(c, m) \to .maydrive(c)$;
$\triangleright .drives(\hat{c}, m) \to \triangleright \hat{c}.go(m)$.

The classes of $\mathcal{DRIVER}$ are $\mathcal{PERSON}$, $\mathcal{CAR}$ and $\mathcal{DRIVER}$. A driver must be a person. In the fourth constraint $\hat{c}$ is a meta-variable standing for an object name, because it is used in $\hat{c}.go$. So it represents a family of axioms, one for each object identity.

Now we can compose $\mathcal{DRIVER}$ and $\mathcal{CONSUMER}$ into a framework $\mathcal{MALL}$ of level 2.

Framework $\mathcal{MALL}$;

IMPORT: $\mathcal{DRIVER}$, $\mathcal{CONSUMER}$;

CONSTRS: $\neg(.DRIVER(x) \wedge .CONSUMER(x))$.

The classes of $\mathcal{MALL}$ are $\mathcal{PERSON}$, $\mathcal{CAR}$, $\mathcal{SHOP}$, $\mathcal{DRIVER}$, $\mathcal{CONSUMER}$ and $\mathcal{MALL}$. By inheritance, drivers and consumers must be persons. By the framework constraint, a person cannot be a consumer and a driver at the same time.

Now we can define *framework instances*. Let $\mathcal{F}$ be a framework. A framework instance $f.\mathcal{F}$ is a closure of $\mathcal{F}$ that is built from a closure of the special class predicates of the framework. The special class predicate $f.F$ is closed by

$$f.F(x) \leftrightarrow x = f.$$

The intended meaning of a closure

$$f.C(x) \leftrightarrow x = o_1 \vee \ldots \vee x = o_k$$

is that the objects of class $\mathcal{C}$ contained in the instance $f.\mathcal{F}$ are $o_1, \ldots, o_k$.

More generally, a closure must satisfy the following meta-assumption:

> $f.C(o)$ holds in the current closure of $f.\mathcal{F}$ if and only if the current closure contains a class instance $o.\mathcal{C}$, and the language of $o.\mathcal{C}$, together with the closure axioms of $o.\mathcal{C}$, belong to the closure axioms of $f.\mathcal{F}$.

In this way, we obtain the system language and the corresponding closure axioms at a given state. Of course, closure axioms are assumed to satisfy all the constraints, those inherited by the constituent frameworks, and those stated in $\mathcal{F}$.

If a framework constraint H of a (possibly container) class $\mathcal{C}$ does not contain meta-variables, then the constraint instance $o.H$, to be satisfied by an instance $o.\mathcal{C}$, is obtained simply by prefixing o to the (possible) attribute and action symbols in H. If it is a schema, i.e. it contains meta-variables for objects, then it has the form

$$Guard(\hat{x_1}, \ldots \hat{x_n}) \rightarrow H$$

where the formula[6] $Guard(x_1, \ldots x_n)$ is true in an instance $o.\mathcal{C}$ for a finite set of objects, to be used to build the instances of the schema. For example, if in the current instance the formula $\triangleright.drives(c, m)$ is closed by

$$\triangleright Dustin.drives(c, m) \leftrightarrow (c = spider \wedge \ldots) \vee (c = spitfire \wedge \ldots)$$

then

$$\triangleright.drives(\hat{c}, m) \rightarrow \triangleright\hat{c}.go(m)$$

instantiates to

$$\triangleright Dustin.drives(spider, m) \rightarrow \triangleright spider.go(m)$$
$$\triangleright Dustin.drives(spitfire, m) \rightarrow \triangleright spitfire.go(m).$$

Thus, if a constraint of a class $\mathcal{C}$ is a (guarded) schema, then an instance of $o.\mathcal{C}$ determines a finite set of constraints in the system language, that must be satisfied by the instance.

[6] With variables $x_1, \ldots x_n$ in place of the corresponding meta-variables.

We assume that $\mathcal{F}$ is consistent, i.e., there is at least one consistent closure for it. We can see that a consistent closure closes all the classes, i.e., it is a theory with an isoinitial model in the current system language.

We will not deal with consistency problems in this paper. We assume that we can build well constrained frameworks, and that there is a suitable way to test constraints when we compose frameworks. Constraint satisfaction and propagation should guarantee that consistency is preserved. The goal of this paper is to give a semantics for the dynamic behaviours of systems specified by frameworks. We conclude this section with an example.

Example 6. We give a system state for the framework $\mathcal{MALL}$, starting from the following closure of class predicates:

$$\begin{aligned} Maceys.MALL(x) &\leftrightarrow x = Maceys \\ Maceys.CAR(x) &\leftrightarrow x = spider \vee x = spitfire \\ Maceys.PERSON(x) &\leftrightarrow x = Dustin \vee x = Ann \\ Maceys.DRIVER(x) &\leftrightarrow x = Dustin \\ Maceys.CONSUMER(x) &\leftrightarrow x = Ann \end{aligned}$$

That is, in the mall *Maceys* there are two persons *Dustin* and *Ann* and two cars *spider* and $spitfire.Dustin$ is a driver, and *Ann* a consumer. In this context, possible closure axioms for *Dustin* are:

$$\begin{aligned} Dustin.DRIVER(x) &\leftrightarrow x = Dustin \\ Dustin.PERSON(x) &\leftrightarrow x = Dustin \\ Dustin.name &= \text{“Dustin Hoffman”} \\ Dustin.pocket &= 0 \\ Dustin.maydrive(x) &\leftrightarrow x = spider \\ \triangleright Dustin.drives(x,m) &\leftrightarrow x = spider \wedge m < 10000 \wedge \triangleright spider.go(m) \\ &\forall x, y \,.\, \neg \odot Dustin.drives(x,y) \end{aligned}$$

where, since *Dustin* inherits from $\mathcal{PERSON}$, it has to inherit the *PERSON* predicate, with closure $Dustin.Person(x) \leftrightarrow x = Dustin$. We omit, for conciseness, the complete closure. Note, however, that there may be many closures, as well as many instances of $\mathcal{MALL}$.

4 Dynamic Aspects of OOD Frameworks

In this section, we introduce dynamic aspects, i.e. time and state changes for systems of objects, by using OOD frameworks with temporal constraints. We interpret such frameworks as *event structures* [20, 14], following the logic MDTL [10], an extension of the TROLL logic [8] for describing dynamic aspects of large object systems. For this purpose, in our formalisation, we need to make sure that the following property is guaranteed: for every class or subframework $\mathcal{C}$ of a framework $\mathcal{F}$, every closure of $\mathcal{F}$ and every object identity o, we can state whether a closure $o.\mathcal{C}$ belongs to the closure of $\mathcal{F}$, and, if it does, then it can be uniquely determined. Of course, if $\mathcal{C}$ is a basic class, then $o.\mathcal{C}$ is an object; and if $\mathcal{C}$ is a subframework, then $o.\mathcal{C}$ is a system of objects. If $o.\mathcal{C}$ belongs to a closure (for some $\mathcal{C}$), we will say that the closure contains the object o.

4.1 Labelled Event Structures

An event structure $E = \langle Ev, \rightarrow^*, \# \rangle$ consists of a set of *events* Ev equipped with two binary relations $\rightarrow^*$ and $\#$ for expressing *causal dependency* and *nondeterminism* respectively. The causal relation $\rightarrow^*$ implies a (partial) order among event occurrences. Intuitively, $e \rightarrow^* e'$ means that the occurrence of event e may *cause* the occurrence of event e'. The $\#$ relation expresses how the occurrence of certain events excludes the occurrences of others. The conflict relation also propagates over causality, i.e. if $e\#e^{'} \rightarrow^* e^{''}$ then $e\#e^{''}$, for all $e, e^{'}, e^{''} \in Ev$. We will use $e \rightarrow e'$ for immediate causality (no other event in between). $e..e'$ will indicate a chain $e \rightarrow e_1 \rightarrow \cdots \rightarrow e_n \rightarrow e'$; the internal part $e_1 \rightarrow \cdots \rightarrow e_n$ may be empty; $e'' \in e..e'$ means that e'' belongs to the internal part. Finally, two events e and e' are *concurrent* if they are not in conflict ($\neg e\#e'$) and are not causally related ($\neg(e \rightarrow^* e' \vee e' \rightarrow^* e)$).

Now we interpret in a framework $\mathcal{F}$ the events of a structure $E = \langle Ev, \rightarrow^*, \# \rangle$, by labelling them by the closures of $\mathcal{F}$.

Definition 2. An *$\mathcal{F}$-labelling* of $E = \langle Ev, \rightarrow^*, \# \rangle$ is a partial function λ, with domain $dom(\lambda) \subseteq Ev$, that maps every event $e \in dom(\lambda)$ into a closure $\lambda(e)$ of $\mathcal{F}$, in such a way that, for every pair e, e' of concurrent events, the objects contained in $\lambda(e)$ are different from the ones contained in $\lambda(e')$.

The language of $\mathcal{F}$ is, so far, a first-order language. We assume that $\mathcal{F}$ has been built in a sound way, i.e., there is at least one closure, and each closure of $\mathcal{F}$ has an isoinitial model. Since a label $\lambda(e)$ is a closure of $\mathcal{F}$, it has an isoinitial model, that represents the state of the system described by $\mathcal{F}$ at event e. We will write $\lambda(e) \models H$, to indicate that a sentence H of the system language of $\lambda(e)$ is true in the isoinitial model of $\lambda(e)$. When a closure of $o.\mathcal{C}$ belongs to the closure $\lambda(e)$, the formulas H of the language of $o.\mathcal{C}$ have interpretation in the isoinitial model of $\lambda(e)$. With this in mind, we can define temporal $\mathcal{F}$-models using $\mathcal{F}$-labelled event structures.

Definition 3. Let $\mathcal{F}$ be a framework. An *$\mathcal{F}$-labelled event structure* is a pair $\langle E = \langle Ev, \rightarrow^*, \# \rangle, \lambda \rangle$, where E is an event structure, and λ is a $\mathcal{F}$-labelling of E such that: for every event $e \in dom(\lambda)$, every $o.\mathcal{C}$ that belongs to the closure $\lambda(e)$ and every ground action term t in the language of $o.\mathcal{C}$, if $\lambda(e) \models \odot t$, then there is an event $e' \in dom(\lambda)$, such that $e' \rightarrow e$, $o.\mathcal{C}$ belongs to $\lambda(e')$ and $\lambda(e') \models \rhd t$.

The existence of an $e' \rightarrow e$ such that t is enabled in e' forces the intended meaning of $\rhd t$ and $\odot t$: an action t must be enabled, in order to occur. That is, $\rhd$ and *hocc* are not temporal operators, but their intended temporal meaning follows from the constraints that we impose on the class of possible $\mathcal{F}$-labelled event structures (according to the approach of [7]).

4.2 A Temporal Constraint Language

Now we define a temporal language for constraints, that is interpreted on our temporal models, namely $\mathcal{F}$-labelled event structures. In this language we distinguish between *framework* and *system* formulas.

Temporal framework formulas are built from the framework formulas defined in the previous section, that we will call *static formulas*.

Temporal formulas use the operators **U** (*until*) and **S** (*since*). Intuitively, $A\mathbf{U}B$ means that A has to hold from the next state, until B becomes true. $A\mathbf{S}B$ means that A has been true starting from a past state where B was true. In both cases, A is not obliged to hold in the current state. The set of temporal formulas of a framework $\mathcal{F}$ is the set of formulas generated from the static formulas of $\mathcal{F}$, by the previous temporal operators and by the usual logical connectives and quantifiers.

Formulas with the temporal operators **S** and **U** are called *temporal formulas*. Other temporal operators like *next* (X), *sometime in the future* (F), *always in the future* (G), *sometime in the past* (P), and *always in the past* (H) can be derived from **S** and **U** (cf. [7] for details).

The instances of a framework $\mathcal{F}$ are defined as in Section 3.2, because the closure axioms do not contain temporal operators. As before, each closure has an isoinitial model, that interprets the *static system formulas*, namely the system formulas defined in Section 3.2.

The *temporal system formulas* of a framework instance are built through the temporal operators, the logical connectives and the quantifiers, starting from the static system formulas.

A temporal framework $\mathcal{F}$ is one whose constraints may be temporal framework formulas. Temporal constraints cannot be interpreted in a single closure of $\mathcal{F}$. They have meaning only in the context of a temporal model, i.e., a $\mathcal{F}$-labelled event structure $M = \langle E = \langle Ev, \rightarrow^*, \# \rangle, \lambda \rangle$. At this level, static constraints are already satisfied, because the closures $\lambda(e)$ satisfy them by definition (of closure).

To interpret temporal constraints, we define the relation $M, e, v \models H$, which means that a system formula H, evaluated in a state e of an $\mathcal{F}$-labelled event structure M, is true at e for the valuation v of its (possible) free variables. v maps each $x : s$ of $dom(v)$ into a value $v(x) \in s^{\lambda(e)}$, where $s^{\lambda(e)}$ is the interpretation of the sort s in the isoinitial model of $\lambda(e)$. Since all the sort symbols belong to the signature Σ_P (see Section 3), their meaning is stated by the axioms $X_P \cup X_{Act} \cup X_{Obj}$, that is, we have constant domains. The constant domain $s^{\lambda(e)}$ will be indicated by s^P, to reflect that it does not depend on e and is fixed by the axioms for Σ_P. By $v[x/\alpha]$ we denote the valuation v' such that $dom(v') = dom(v) \cup x$, $v'(x) = \alpha$ and $v'(y) = v(y)$, for $y \neq x$ in $dom(v)$.

Definition 4. Let $\mathcal{F}$ be a framework and $M = \langle E = \langle Ev, \rightarrow^*, \# \rangle, \lambda \rangle$ be an $\mathcal{F}$-labelled event structure. Let e be an event in the domain $dom(\lambda)$, H be a formula that belongs to $L(\lambda(e))$, and v be an assignment of the free variables of H. Then $M, e, v \models H$ holds iff one of the following inductive clauses applies:

- *Basis.* H is a *static* formula true for the valuation v in the isoinitial model of $\lambda(e)$.

– *Step.* H is a *temporal* formula of the form $A\mathbf{U}B$, $A\mathbf{S}B$, $\neg A$, ..., and, according to the cases:[7]

$$
\begin{array}{ll}
M,e,v \models A\mathbf{U}B & \text{iff there is } e..e' \text{ s.t. } M,e',v \models B \in L(\lambda(e')) \\
& \quad \text{and for } e'' \in e..e'\ M,e'',v \models A \in L(\lambda(e'')) \\
M,e,v \models A\mathbf{S}B & \text{iff there is } e'..e \text{ s.t. } M,e',v \models B \in L(\lambda(e')) \\
& \quad \text{and for } e'' \in e..e'\ M,e'',v \models A \in L(\lambda(e'')) \\
M,e,v \models \neg A & \text{iff } M,e,v \not\models A \\
M,e,v \models A \wedge B & \text{iff } M,e,v \models A \text{ and } M,e,v \models B \\
M,e,v \models A \vee B & \text{iff } M,e,v \models A \text{ or } M,e,v \models B \\
M,e,v \models A \rightarrow B & \text{iff } M,e,v \models A \ => \ M,e,v \models B \\
M,e,v \models \forall x : s \,.\, A & \text{iff } M,e,v[x/\alpha] \models A \text{ for every } \alpha \in s^P \\
M,e,v \models \exists x : s \,.\, A & \text{iff } M,e,v[x/\alpha] \models A \text{ for at least one } \alpha \in s^P
\end{array}
$$

As usual, if a formula H is closed, we omit the assignment and we simply write $M,e \models H$. Now, we consider the temporal constraints of a framework $\mathcal{F}$. As for static constraints, in every closure $\lambda(e)$, each temporal framework constraint instantiates to a corresponding system constraint, or a finite set of system constraints, if it is a (guarded) constraint schema.

Temporal constraint satisfaction is defined as follows. We say that a constraint H of a class $\mathcal{C}$ in a framework $\mathcal{F}$ is *satisfied* by a $\mathcal{F}$-labelled event structure M if, for every $e \in \lambda$, every instance $o.\mathcal{C}$ of $\lambda(e)$ and every instance H_o of H contained in $o.\mathcal{C}$, we have that $M,e \models H_o$.

We say that an $\mathcal{F}$-labelled event structure is a *model* of $\mathcal{F}$ if it satisfies the temporal constraints of $\mathcal{F}$.

4.3 Example: A *Catalysis* OOD Framework

To illustrate our formalisation of OOD frameworks, we now show how to specify an OOD framework in the style of the CBD methodology *Catalysis* [6].

Example 7. Consider the PersonAsEmployeeConsumer OOD framework in *Catalysis* depicted in Figure 6.

In PersonAsEmployeeConsumer, a person plays the role of an employee of a company. A person as an employee has an attribute *pocket* representing the amount of money he possesses, and two actions *receive_pay* and *work*. In this example, a person as an employee only works if he has less than £500 (precondition for *work*). A person only receives a payment if he has worked before (pre-condition for *receive_pay*). If a person receives a payment, the money in his pocket increases by the amount received.

In PersonAsEmployeeConsumer, a person also plays the role of a consumer. As a consumer, a person has an action $buy(p)$, where p represents the price of the item bought.

We have formalised the class $\mathcal{PERSON}$ in Example 5. Here, for conciseness, we will omit the attribute *.name*, and use only *.pocket*. Also in Example 5, we

[7] $M,e,v \models F \in L(\lambda(e'))$ abbreviates $F \in L(\lambda(e'))$ and $M,e,v \models F$.

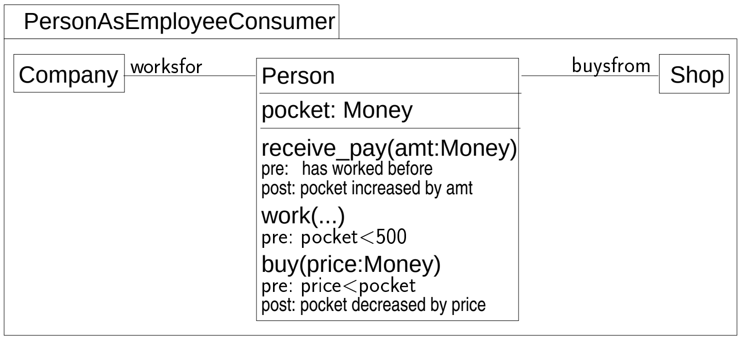

Fig. 6. PersonAsEmployeeConsumer OOD framework.

have formalised the class $\mathcal{CONSUMER}$, but without any temporal constraints. Now we add the following temporal constraint:

$$\odot .buy(p) \wedge Y(.pocket = n) \rightarrow .pocket = n - p.$$

This constraint says that, after buying an item, the money in the pocket of a consumer decreases by the amount of money spent.

The framework $\mathcal{EMPLOYEE}$ can be formalised as follows.

Framework $\mathcal{EMPLOYEE}$;
IMPORT: $\mathcal{PERSON}$
ACTIONS : $.receive\_pay : Int \rightarrow Act;$
$.work \quad : [\,] \rightarrow Act;$
CONSTRS: $\rhd .receive\_pay(a) \rightarrow P \odot .work;$
$\odot .receive\_pay(a) \wedge Y(.pocket = n) \rightarrow .pocket = n + a;$
$\rhd .work \rightarrow .pocket < \pounds 500.$

The first constraint states that if the action *receive_pay* is enabled, then sometime in the past (temporal operator P) the action *work* must have occurred.

The second constraint says that the occurrence of action $receive\_pay(a)$ implies that if in the previous state (i.e., yesterday Y) the value of *pocket* was n, then its current value is $n + a$.

The third constraint states that if the action *work* is enabled (it might occur in the next state) then the value of *pocket* must be less than £500.

Now we can build the framework $\mathcal{EMPLOYEECONSUMER}$:

Framework $\mathcal{EMPLOYEECONSUMER}$;
IMPORT: $\mathcal{EMPLOYEE}, \mathcal{CONSUMER}$
COMMON: $\mathcal{PERSON}$
ACTIONS : $.bePerson \quad : [\,] \rightarrow Act;$
$.beEmployee : [\,] \rightarrow Act;$
$.beConsumer : [\,] \rightarrow Act;$

CONSTRS: $.EmployeeConsumer(x) \to .Person(x)$
$(\triangleright self.bePerson \leftrightarrow self.Consumer(self) \vee self.Employee(self)$
$(\triangleright self.beEmployee \leftrightarrow \neg self.Employee(self) \wedge self.pocket < 500)$
$(\triangleright self.beConsumer \leftrightarrow \neg self.Consumer(self) \wedge self.pocket > 400)$
$(\odot self.bePerson \to \neg self.Consumer(self) \wedge \neg self.Employee(self)$
$(\odot self.beEmployee \to self.Employee(self)$
$(\odot self.beConsumer \to self.Consumer(self)$

Here *self* is a special meta-variable for objects with the usual meaning.

The first constraint says that the class EmployeeConsumer inherits from Person. However, nothing is said about Consumer and Employee. These roles can be assumed (by a person) dynamically, one at a time or both together.

The other constraints give the meaning of the actions to change the role of an object of class EmployeeConsumer. A Consumer or an Employee may always become a Person (the action *.bePerson* is enabled, second constraint). If he becomes a Person, then he is no longer a Consumer or an Employee (fifth constraint). The other constraints can be interpreted in a similar fashion.

Now for the framework $\mathcal{EMPLOYEECONSUMER}$, we can give an example of a model. Consider the event structure in Figure 7 for a person *Joe* who can play the roles of an employee and a consumer. In general, when we represent event

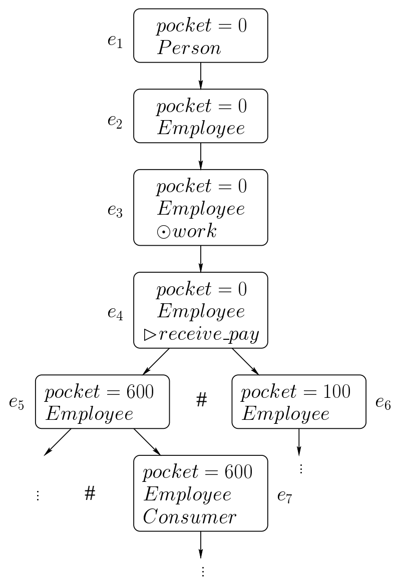

Fig. 7. Event structure for *Joe* as an employee and a consumer.

structures graphically, *boxes* denote events $\{e_1, e_2, \ldots\}$, *arrows* between boxes represent *causality*, and # denotes *conflict*. Event e_1 corresponds to the initial state. The occurrence of e_2 depends on the previous occurrence of e_1. The state of the object at a given event is written inside the box as a state formula. In

Figure 7, for simplicity, the states are just indicated by the current value of *Joe*'s *pocket* and the current role(s) played by him.

In Figure 7, in state e_1, *Joe* starts as a simple person, with $Joe.pocket = 0$:

$$\begin{aligned}\lambda(e_1) = {} & Joe.EmployeeConsumer(x) \leftrightarrow x = Joe \\ & \neg Joe.Consumer(x) \\ & \neg Joe.Employee(x) \\ & Joe.Person(x) \leftrightarrow x = Joe \\ & Joe.pocket = 0 \\ & \rhd Joe.beEmployee \wedge \neg \odot Joe.beEmployee \\ & \neg \rhd Joe.beConsumer \wedge \neg \odot Joe.beConsumer\end{aligned}$$

Since his pocket is 0, *Joe* may become an employee (constraint 3), but not a Consumer (constraint 4). No action has occurred, because there is no previous state where any action was enabled.

In the next state e_2, the action $Joe.beEmployee$ has occurred, so *Joe* is now an Employee. It acquires the attributes of class Employee, and $Joe.work$ is enabled. *Joe* works ($e_2 \rightarrow e_3$), but only in e_4 may he receive pay ($\rhd receive\_pay(x) \leftrightarrow x < 800$).

$$\begin{aligned}\lambda(e_2) = {} & Joe.EmployeeConsumer(x) \leftrightarrow x = Joe \\ & \neg Joe.Consumer(x) \\ & Joe.Employee(x) \leftrightarrow x = Joe \\ & Joe.Person(x) \leftrightarrow x = Joe \\ & Joe.pocket = 0 \\ & \rhd Joe.work \wedge \neg \odot Joe.work \\ & \neg \rhd Joe.receive\_pay(x) \wedge \neg \odot Joe.receive\_pay(x) \\ & \neg \rhd Joe.beEmployee \wedge \odot Joe.beEmployee \\ & \neg \rhd Joe.beConsumer \wedge \neg \odot Joe.beConsumer\end{aligned}$$

$$\begin{aligned}\lambda(e_3) = {} & Joe.EmployeeConsumer(x) \leftrightarrow x = Joe \\ & \neg Joe.Consumer(x) \\ & Joe.Employee(x) \leftrightarrow x = Joe \\ & Joe.Person(x) \leftrightarrow x = Joe \\ & Joe.pocket = 0 \\ & \rhd Joe.work \wedge \odot Joe.work \\ & \neg \rhd Joe.receive\_pay(x) \wedge \neg \odot Joe.receive\_pay(x) \\ & \neg \rhd Joe.beEmployee \wedge \neg \odot Joe.beEmployee \\ & \neg \rhd Joe.beConsumer \wedge \neg \odot Joe.beConsumer\end{aligned}$$

$$\begin{aligned}\lambda(e_4) = {} & Joe.EmployeeConsumer(x) \leftrightarrow x = Joe \\ & \neg Joe.Consumer(x) \\ & Joe.Employee(x) \leftrightarrow x = Joe \\ & Joe.Person(x) \leftrightarrow x = Joe \\ & Joe.pocket = 0 \\ & \rhd Joe.work \wedge \neg \odot Joe.work \\ & (\forall x \,.\, \rhd Joe.receive\_pay(x) \leftrightarrow x < 800) \wedge \\ & (\forall x \,.\, \neg \odot Joe.receive\_pay(x))\end{aligned}$$

$$\neg \rhd \mathit{Joe.beEmployee} \wedge \neg \odot \mathit{Joe.beEmployee}$$
$$\neg \rhd \mathit{Joe.beConsumer} \wedge \neg \odot \mathit{Joe.beConsumer}$$

The next two states $e_4 \to e_5$ and $e_4 \to e_6$ are in conflict (only one of them may occur). In e_5 the pay is 600 and *Joe* may become a Consumer, whereas in e_6 the pay is 100, and he cannot become a Consumer.

$$\begin{aligned} \lambda(e_5) = \ldots & \\ & \mathit{Joe.pocket} = 600 \\ & (\forall x \,.\, \neg \rhd \mathit{Joe.receive\_pay}(x)) \\ & \wedge(\forall x \,.\, \odot \mathit{Joe.receive\_pay}(x) \leftrightarrow x = 600) \\ & \ldots \\ & \rhd \mathit{Joe.beConsumer} \wedge \neg \odot \mathit{Joe.beConsumer} \end{aligned}$$

$$\begin{aligned} \lambda(e_6) = \ldots & \\ & \mathit{Joe.pocket} = 100 \\ & (\forall x \,.\, \neg \rhd \mathit{Joe.receive\_pay}(x)) \\ & \wedge(\forall x \,.\, \odot \mathit{Joe.receive\_pay}(x) \leftrightarrow x = 100) \\ & \ldots \\ & \neg \rhd \mathit{Joe.beConsumer} \wedge \neg \odot \mathit{Joe.beConsumer} \end{aligned}$$

In Figure 7, we also show a state e_7 where $Joe.beConsumer$ has occurred. In some states, some attributes change without any explicit action occurrence, because we allow specifications of open modules, where some other parts are not modelled. For example, the action *receive_pay* is enabled by *Joe*'s company, and it occurs through a bank, but the cooperations with those objects is left implicit.

Finally, it is easy to see that this event structure is indeed a model of the framework $\mathcal{EMPLOYEECONSUMER}$, since all the (system instances of the) temporal constraints of the framework are satisfied.

5 Conclusion

In this paper we have described our preliminary effort to formalise dynamic aspects of OOD frameworks. So far our formalisation of OOD frameworks does not deal with the dynamic aspects of framework composition. This is future work. Our eventual goal is a semantics for composing OOD frameworks with state transitions in the manner depicted in Figure 3.

Our work is closely related to Troll [8], which is used for specifying large distributed/concurrent object systems, and to [3], which formalises an algebraic semantics for object model diagrams in OMT [19]. The main difference is that they take the traditional view of objects (Figure 2), whereas we adopt the multiple-role, more reusable approach (Figure 3). Another fundamental difference is that their semantics is based on initial theories, as opposed to isoinitial theories that we use. Initial semantics is convenient for defining computations at the operational level (a good example is Maude [16, 17]), whereas isoinitial semantics is more suited to declarative semantics for specifications.

Overall, our approach is model-theoretic, whereas other approaches are mostly proof- or type-theoretic. For example, our model-theoretic characterisation of states and objects stands in contrast to the type-theoretic approach, e.g., [1]. Our model-theoretic approach also enables us to define a notion of correctness that is preserved through inheritance hierarchies, which is particularly suitable for component-based software development.

Acknowledgements

We wish to thank Don Sannella for his helpful comments and suggestions during our discussion at the workshop, and Alberto Pettorossi for pointing out important technical issues.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. A. Bertoni, G. Mauri, and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2):127–170, 1983.
3. R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799-821, 1995.
4. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.
5. S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.
6. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, October 1998.
7. H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for specifying concurrent information systems. G. Saake and J. Chomicki, editors, *Logics for Databases and Information Systems*, pages 167–198, Kluwer Academic, 1998.
8. A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger and H.-D. Ehrich. The TROLL approach to conceptual modelling: syntax, semantics and tools. In T.W. Ling, S. Ram and M.L. Leebook, editors, *Proc. 17th Int. Conference on Conceptual Modeling*, *LNCS* 1507:277-290, Springer, 1998.
9. R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* 25(10) (*Proc. ECOOP/OOPSLA 90*).
10. J. Küster Filipe. Fundamentals of a module logic for distributed object systems. *J. Functional and Logic Programming* 2000(3), 2000.
11. K.-K. Lau and M. Ornaghi. Isoinitial models for logic programs: A preliminary study. In J.L. Freire-Nistal, M. Falaschi, and M. Vilares-Ferro, editors, *Proceedings of the 1998 Joint Conference on Declarative Programming*, pages 443-455, A Coruña, Spain, July 1998.
12. K.-K. Lau and M. Ornaghi. On specification and correctness of OOD frameworks in computational logic. In A. Brogi and P. Hill, editors, *Proc. 1st Int. Workshop on Component-based Software Development in Computational Logic*, pages 59-75, September 1998, Pisa, Italy.
13. K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR'98*, *LNCS* 1559:101-123, Springer-Verlag, 1999.

14. R. Loogen and U. Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae* XIV(1):39–73, January 1991.
15. R. Mauth. A better foundation: development frameworks let you build an application with reusable objects. *BYTE* 21(9):40IS 10-13, September 1996.
16. Maude home page. http://maude.csl.sri.com/
17. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
18. R. Pooley and P. Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.
19. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
20. G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, LNCS* 255:325–392, Springer, 1987.