Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues

Ivica Crnkovic

Department of Computer Engineering, Mälardalen University
721 23 Västerås, Sweden
E-mail: Ivica.Crnkovic@mdh.se

Juliana Küster Filipe*

Abt. Informationssysteme, Informatik, Technische Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, Germany
E-mail: J.Kuester-Filipe@tu-bs.de

Magnus Larsson

ABB Automation Products AB, LAB
721 59 Västerås, Sweden
E-mail: Magnus.Larsson@mdh.se

Kung-Kiu Lau

Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
E-mail: kung-kiu@cs.man.ac.uk

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 this paper, we show how to formally specify OOD frameworks, and briefly discuss their implementation and configuration management.

Keywords: Object-oriented design frameworks, component-based software development.

1. Introduction

Object-Oriented Design (OOD) frameworks are groups of (interacting) objects. For example, in the CBD (Component-based Software Development) methodology *Catalysis* [10], a driver

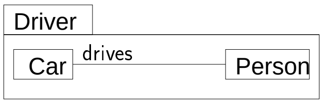

Figure 1. The Driver OOD framework.

may be represented as the OOD framework shown in Figure 1.[1] A driver is a person who drives a car, or in OOD terminology, 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. [12, 20]). 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 [6] and Syntropy [8]) 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 roles in different

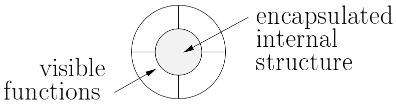

Figure 2. Traditional view of an object.

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

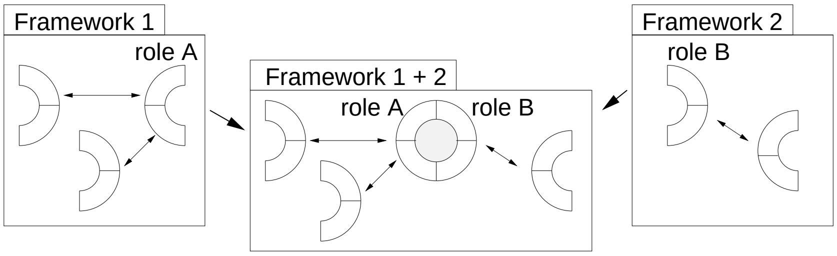

Figure 3. Objects by composing OOD frameworks.

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 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 composite object of the kind depicted in Figure 3.

OOD frameworks should play a crucial role in the design and implementation of next-generation component-based software systems. In this paper, we show how to formally specify them, and briefly discuss their implementation (in COM) and configuration management.

*The second 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. [21].

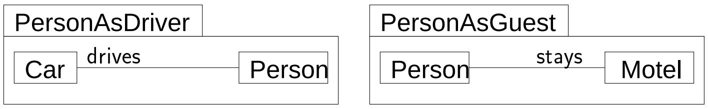

Figure 4. PersonAsDriver and PersonAsGuest OOD frameworks.

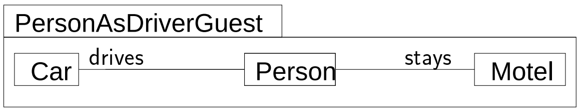

Figure 5. PersonAsDriverGuest OOD framework.

2. Formal Specification of OOD Frameworks

In this section, we describe formal specification of OOD frameworks. First we consider the *static* aspects, i.e. *without* time or state transitions, then we consider the *dynamic* aspects, i.e. *with* time and state transitions.

2.1. Static Aspects

We have considered the static aspects of OOD frameworks in [17, 18]. In this section, we briefly outline this semantics.

As we have seen in Section 1., OOD frameworks are composite objects/classes. In our approach, we define OOD frameworks and objects/classes in terms of a basic entity that we call a *specification framework*, or just a *framework*, for short.[2]

A *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 purpose of a framework is to axiomatise a problem domain and to reason about it. In our approach, a problem domain contains the ADT's and classes needed to define the objects 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:[3]

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. The relationship between open and closed frameworks plays a crucial role in our interpretation of objects. Roughly speaking, in object-oriented programming terminology, open frameworks represent *classes*, and closed frameworks represent their instances, i.e. *objects*.

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.

Open frameworks can be *closed*, i.e made into closed frameworks, by instantiating its open symbols. We will use only open frameworks which have reachable isoinitial models for all their

[2]To avoid confusion with OOD frameworks, in this section we will use 'framework' to refer to a specification framework only, and not to an OOD framework.

[3]See [18] for a justification of this choice and [3, 16] for a discussion of isoinitial theories.

instances. Such frameworks are called *adequate*, and they can be constructed incrementally from small (adequate) closed frameworks (see [17, 18]).

Example 2..1 The Car class in Figures 4 and 5 can be defined as the following *open* framework:

$$
\begin{array}{lrcl}
\multicolumn{4}{l}{\textbf{OBJ-Framework } \mathcal{CAR}(.km, .option);} \\
\text{IMPORT:} & \multicolumn{3}{l}{\mathcal{INT}, year96;} \\
\text{DECLS:} & .km & : & [\,] \rightarrow Int \\
 & .option & : & [year96.opts] \\
\text{CONSTRS:} & \multicolumn{3}{l}{.km \geq 0}
\end{array}
$$

where $\mathcal{INT}$ is a predefined ADT of integers, and $year96$ is an object that contains the sort $year96.opts$ of the possible options for a car in the year 96. The constraint $.km \geq 0$ is an axiom for the open symbol $.km$.

We call a framework like this an *OBJ-framework*, since it is a class of objects. To obtain objects we instantiate an OBJ-framework, i.e. by closing the OBJ-framework.

The axioms used to close $\mathcal{F}(\Omega)$ into an object represent the *state* of the object, and are called *state axioms*. State axioms can be updated, i.e. an object is a dynamic entity.

An object of class $\mathcal{CAR}$ is created, for example, by:

$$
\begin{array}{lrcl}
\multicolumn{4}{l}{\text{NEW } spider : \mathcal{CAR};} \\
\text{CLOSE:} & spider.km & \text{BY} & spider.km = 25000; \\
 & spider.option & \text{BY} & spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond.
\end{array}
$$

where $spider.km = 25000$ and $spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond$ are (explicit) definitions that close the constant $spider.km$ and the predicate $spider.option(x)$ respectively.

The state of a spider object can be updated, by redefining its state axioms:

$$
\begin{array}{l}
\text{UPDATE } spider : \mathcal{CAR}; \\
spider.km = 27000 \\
spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond
\end{array}
$$

As we can see, the constant $spider.km$ has been changed.

An OOD framework is a composite OBJ-framework. It can be viewed as a *system* of objects, in which objects can be created (and deleted) and updated dynamically.

Example 2..2 The PersonAsDriver OOD framework in Figure 5 can be formalised as the following framework:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{OOD-Framework } \mathcal{DRIVER}[\mathcal{PERSON}, \mathcal{CAR}];} \\
\text{DECLS:} & .drives : [obj]; \\
\text{CONSTRS:} & `X.drives(c) \rightarrow PERSON(`X) \wedge `X.age \geq 18 \wedge CAR(c); \\
 & (\exists c : obj)(.drives(c));
\end{array}
$$

where obj is a reserved sort symbol that contains the set of names of all existing objects in the system; $`X$ is a meta-symbol that stands for any object name; and the OBJ-framework $\mathcal{PERSON}$ may be something like:

OBJ-Framework $\mathcal{PERSON}$;

IMPORT ...;

DECLS: $.name : \rightarrow string;$
$.age : \rightarrow int; \ldots$

CONSTRS: $.age \geq 0$

Here the composite object is built via links between its components, which constrain object creation (and deletion) methods. We cannot create an object $n.\mathcal{DRIVER}$, if n is not a person. Furthermore, we need at least one car c.

2.2. Dynamic Aspects

In this section, we consider how to introduce time and state changes. We will combine the static formalisation outlined in the previous section with the logic MDTL presented in [13]. MDTL is an extension of the TROLL logic [11] for describing dynamic aspects of large object systems.

2.2.1. State Transitions.

In MDTL, an OBJ or OOD framework has a local logic consisting of a *home* and a *communication* logic. The *home* logic allows us to express internal state changes, whereas the *communication* logic describes framework interactions. The *home* logic of a framework is mainly a first-order temporal logic with (true) concurrency. We do not deal with concurrency explicitly in this paper, and so we will use axioms that are just first-order temporal formulae. Also, in this paper, we will not use the communication logic, since we do not deal with framework interactions.

In MDTL, in addition to attributes, an object also has *actions*,[4] which will affect its current state. Actions may be either *enabled* or *occurring* in a particular state. The *state* of an object is given by the current values of the attributes, and the current status of its actions. Thus in MDTL, a state formula is a conjunction of facts (the current values of the attributes) and actions (enabled or occurring).

If an action is *enabled*, then it may occur in the next state. When an action occurs, the state of the object changes. In the logic, $\odot a$ is used to denote the occurrence of action a, and $\triangleright a$ that the action a is enabled. If an action occurs, then it must have been enabled in the previous state: $\odot a \Rightarrow Y \triangleright a$. In this formula, Y is the temporal operator *Yesterday* referring to the previous state. Enabling ($\triangleright$) is useful for expressing *preconditions*, and occurrence ($\odot$) for expressing *postconditions*.

In the sequel, we shall also use the temporal operators X (next state), F (sometime in the future including the present), and P (sometime in the past including the present).

The state of an OOD framework is given by the states of the current objects belonging to the framework.

We illustrate how to specify state transitions of an OOD framework in MDTL with a simple example.

Example 2..3 Consider the OOD framework for employees as depicted in Figure 6, in which 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*.

[4]More commonly known as *methods* in object-oriented programming.

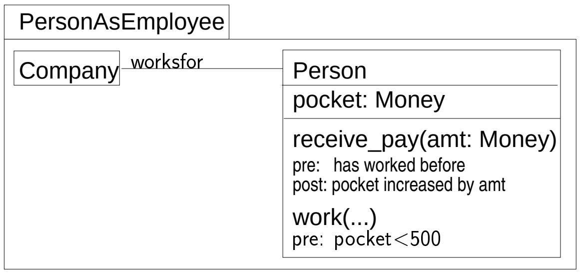

Figure 6. PersonAsEmployee OOD framework.

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 (precondition for *receive_pay*). If a person receives a payment, the money in his pocket increases by the amount received. Here, we express the pre- and postconditions only informally, and we omit the parameters and the postcondition of *work*, as well as the definition of the OBJ-framework for Company.

The class Person might be formalised by the following OBJ-framework:

OBJ-Framework $\mathcal{PERSON}$;

IMPORT: $\mathcal{MONEY}$

DECLS:
$$\begin{aligned} .pocket &: [\,] \rightarrow Money; \\ .receive\_pay &: [Money]; \\ .work &: [\ldots]; \end{aligned}$$

AXIOMS: ...

ST-AXIOMS:
$$\begin{aligned} &\forall_a \rhd .receive\_pay(a) \Rightarrow P \odot .work(\ldots); \\ &\forall_{a,n} \odot .receive\_pay(a) \Rightarrow Y(.pocket = n) \Rightarrow .pocket = n + a; \\ &\rhd .work(\ldots) \Rightarrow .pocket < £500. \end{aligned}$$

where the ST-AXIOMS are the *state transition axioms.*

Pre- and postconditions allow us to define the state transitions of a framework. The state transition axioms do not affect the (static) isoinitial model of the OBJ-framework, and are relevant only for the behaviour model. The first axiom states that if the action *receive_pay* is enabled, then sometime in the past (temporal operator P) the action $work(\ldots)$ must have occurred. The second axiom says that the occurrence of action $receive\_pay(a)$ implies that if in the previous state the value of *pocket* was n, then its current value is $n + a$. Finally, the third axiom 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.

We can create an object *joe* of Person class as follows:

NEW $joe : \mathcal{PERSON}$;

CLOSE:
$$\begin{aligned} joe.pocket &\quad \text{BY} \quad joe.pocket = £100; \\ joe.work &\quad \text{BY} \quad \rhd joe.work(\ldots); \\ joe.pay &\quad \text{BY} \quad \neg \rhd joe.receive\_pay(a). \end{aligned}$$

When an object is created, its initial state is defined. In the initial state of *joe*, (attribute) *pocket* is £100, (action) $receive\_pay(a)$ is disabled for any a, and (action) $work(\ldots)$ enabled.

In MDTL we can also express general properties of objects. For example,[5]

$$joe.pocket < \pounds 500 \Rightarrow F\ \exists_a \odot joe.receive\_pay(a)$$

means 'if *joe* has less than £500 then he will receive a payment sometime'.

2.2.2. *Event Structures.*

MDTL is interpreted over *labelled prime event structures* ([24]). A labelled prime event structure is thus a model for an OOD framework if it satisfies all the axioms of the framework (both the static and the state transition axioms).

A labelled prime event structure consists of a prime event structure and a labelling function. Prime event structures can be used to describe distributed computations as event occurrences together with a *causal* and a *conflict* relations between them. The causal relation implies a (partial) order among event occurrences, and the conflict relation denotes a choice. Events in conflict cannot belong to the same *run* or *life cycle*. The labelling function associates each event with a state.

Example 2..4 Consider the event structure in Figure 7. It shows a small part of a (sequential)

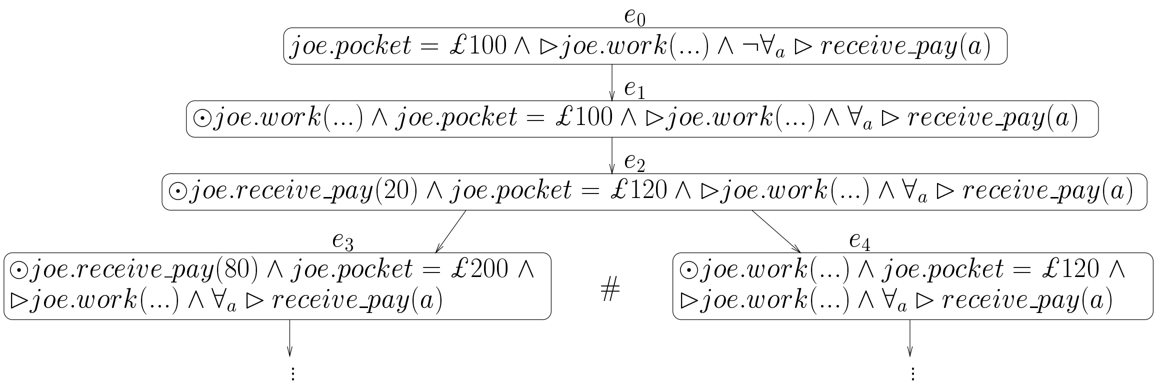

Figure 7. Event structure for *joe* as an employee.

behaviour model for *joe*.

In general, in event structures, *boxes* denote events $\{e_0, e_1, \ldots\}$, *arrows* between boxes represent event *causality*, and # denotes event *conflict*. The state of the object at a given event is written inside the box as a state formula.

For the object *joe*, the events in the event structure are labelled by the formulae of the state logic of the Person class. Event e_0 corresponds to the initial state. The occurrence of e_1 depends on the previous occurrence of e_0. With the occurrence of action $joe.receive\_pay(20)$ at event e_2, the current value of attribute *pocket* changes to £120. Events e_3 and e_4 are in conflict, which means that either one or the other occurs but not both. A conflict thus denotes a choice.

There are therefore two life cycles for *joe* in Figure 7. One consists of events $\{e_0, e_1, e_2, e_3, \ldots\}$ and the other $\{e_0, e_1, e_2, e_4, \ldots\}$. In the former, *joe* receives two payments after working. In the latter, *joe* works, then receives a payment and then works again.

Finally, it is easy to see that this event structure satisfies the state transition axioms of the OBJ-framework $\mathcal{PERSON}$.

[5] We are not saying that this property necessarily follows from the ST-axioms of the object *joe*.

In general, labelled event structures provide models for concurrent computations. Other such models include transition systems, Petri nets, traces, and synchronisation trees. Petri nets and transition systems allow an explicit representation of the (possibly repeating) states in a system, whereas trees, traces and event structures abstract away from such information, and focus instead on the behaviour in terms of patterns of occurrences of actions over time. Furthermore, event structures are a "true" concurrency model, as opposed to transition systems that model systems as non-deterministically interleaved sequential computations. A detailed survey and comparison of some of these models can be found in [24].

2.2.3. Composing Event Structures.

In order to create objects by composing OOD frameworks with state transitions in the manner depicted in Figure 3, we need to be able to compose event structures.

Example 2..5 In the previous example, a person plays the role of an employee. This partial definition of person could be combined with another view of a person, e.g., a person as a consumer. The PersonAsConsumer OOD framework in Figure 8 defines this role for a person

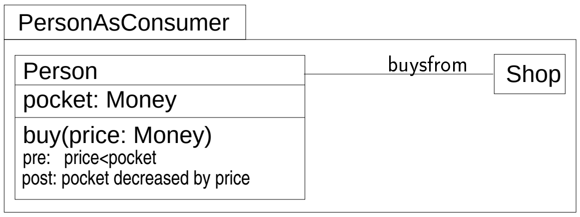

Figure 8. PersonAsConsumer OOD framework.

object. The class Person here can be defined by the same OBJ-framework $\mathcal{PERSON}$ in Example 2..3, but with the action $buy(p)$ instead of the actions $receive\_pay(a)$ and $work(\ldots)$. We omit the definition of the OBJ-framework for Shop.

A consumer has an action $buy(p)$, where p represents the price of the item bought. Pre- and postconditions for this action are the expected ones. A consumer may only buy something if he has enough money, and after buying an item the money in his pocket decreases by the amount of money spent. The state transition axioms for $buy(p)$ are thus:

$$\forall_p \rhd .buy(p) \Rightarrow .pocket > p$$
$$\forall_{p,n} \odot .buy(p) \Rightarrow Y(.pocket = n) \Rightarrow .pocket = n - p$$

Let joe be a person playing now the role of a consumer. In the event structure for joe as a consumer, a life cycle is a linear sequence of buy events starting from the initial state in which $joe.pocket$ is initialised. In Figure 9, we show two possible life cycles with distinct initial states.

We may compose the OOD frameworks for PersonAsEmployee and PersonAsConsumer, to obtain a person with both roles together. A person now has all the actions of both roles, namely $receive\_pay$, $work$ and buy, and the attribute $pocket$ in both roles. The composition is illustrated by Figure 10.

The composite PersonAsEmployeeConsumer framework contains the union of the state transition axioms of its component OOD frameworks. An event structure, i.e. a model, for a person as an employee and consumer is obtained by composing a model for person as an employee with one for person as a consumer in a special manner. Several constructions for sequential and parallel composition of event structures have been defined in the literature, e.g. [23, 19]. What we

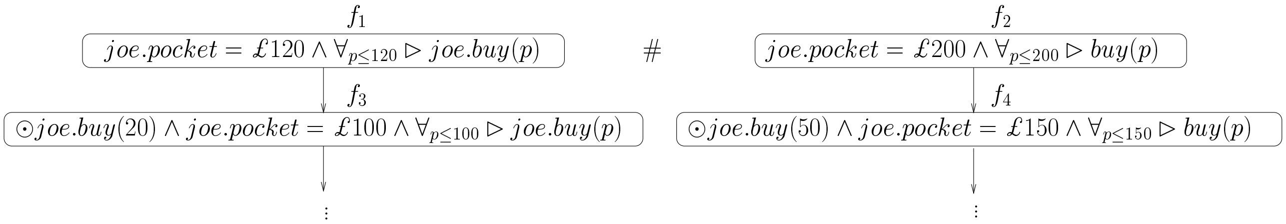

Figure 9. Event structure for *joe* as a consumer.

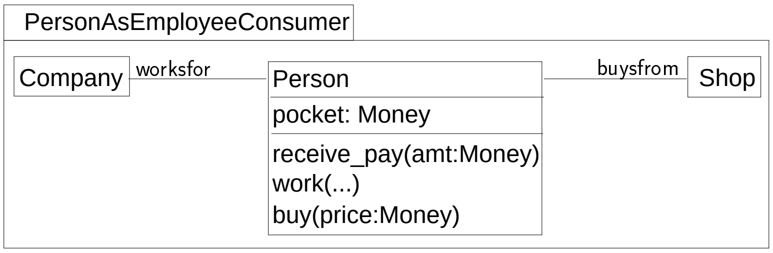

Figure 10. PersonAsEmployeeConsumer OOD framework.

need for composing roles is in fact a combination of *interleaving* of the models and *synchronisation*. Interleaving (sequential composition), because we are combining models for the same object in different roles (and objects are considered to behave sequentially), and therefore the composed model must be sequential. Synchronisation, because some attributes and/or actions for distinct roles may be identified as the same (e.g. *pocket*).

Figure 11 shows the composed model for *joe* as an employee and consumer based on the models of Figures 7 and 9 for *joe* as an employee and *joe* as a consumer respectively. In this

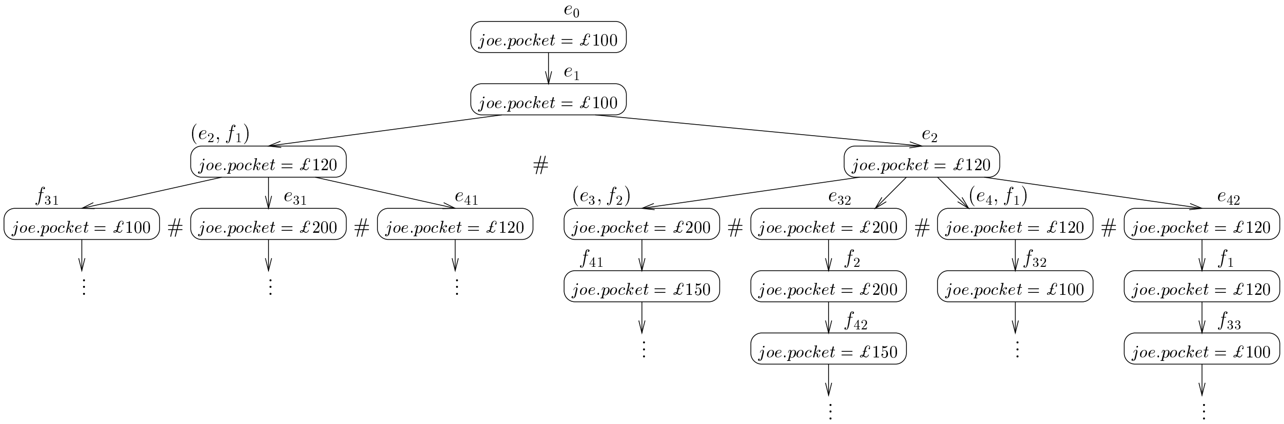

Figure 11. Event structure for *joe* as an employee and a consumer.

case, event synchronisation is done over the value of the common attribute *pocket*. That is, only those events of both models that have the same value for pocket may be synchronised, e.g., events e_2 and f_1, e_3 and f_2, and e_4 and f_1. Synchronisation is indicated by the pairs (e_2, f_1), etc.

Normally, synchronisation is done over actions, but we have a new situation here by combining roles, namely that the attribute *pocket* of a person as an employee is to be identified with the attribute *pocket* of a person as a consumer, whereas the actions *receive_pay*, *work* and *buy* are all distinct. Synchronisation is not always necessary, and some of the life cycles in

Figure 11 show just interleaving.

The construction of composite event structures sometimes leads to the duplication of events, e.g., event e_3 has been duplicated and corresponds to events e_{31} and e_{32}. The labels of these events are the same as e_3. In Figure 11, the states are just indicated by the value of the attribute *pocket* for simplicity. The label of event (e_2, f_1) is given by the conjunction of the labels e_2 and f_1, i.e., it corresponds to the formula

$$joe.pocket = \pounds 120 \wedge \odot receive\_pay(20) \wedge \rhd joe.work \wedge \forall_a receive\_pay(a) \wedge \\ \forall_{p \leq 120} \rhd buy(p)$$

Parallel composition of event structures with synchronisation is useful for modelling interacting frameworks (e.g., [15]), whereas without synchronisation it models non-interacting frameworks.

3. Some Implementation Issues

Having shown how to formally specify OOD frameworks, we now turn to practical concerns. In particular, we will discuss how we might construct such frameworks in practice using currently available technology, and the issues involved in such constructions.

Of the current technologies for developing component-based software systems, COM [5] seems to lend itself most readily to the implementation of OOD frameworks. Therefore, we will briefly show how to implement OOD frameworks in COM.

The fact that OOD frameworks are composite objects/classes means that constructing these frameworks creates problems for *configuration management*. Therefore, we will consider some of these problems, in the case of COM implementations.

3.1. Implementing OOD Frameworks in COM

In this section, we show how to use COM to implement the OOD frameworks in Figures 6, 8 and 10. COM suits multiple roles because it can use multiple interfaces for each role. We will use the aggregation mechanism in COM to compose OOD frameworks. First, we implement the Person object, which corresponds to the encapsulated internal structure in Figure 2. The Person object is constructed so it supports aggregation of role objects and it has one IPerson interface (see Figure 12).

Figure 12. A COM object for the person object and the consumer role.

Secondly, the consumer and employee roles are implemented so they support being aggregated into a person object. Figure 12 shows the consumer role with one IConsumer interface. The consumer object also needs a reference to the person object to be able to work on the pocket variable. The person reference is set up when the consumer is aggregated into the person object (see Figure 13). In a similar way the employee role is implemented. Using aggregation we can reuse the different components that we have created. Figure 13 shows how Person aggregates the two already defined COM objects. Frameworks are created at run-time by adding roles to an object.

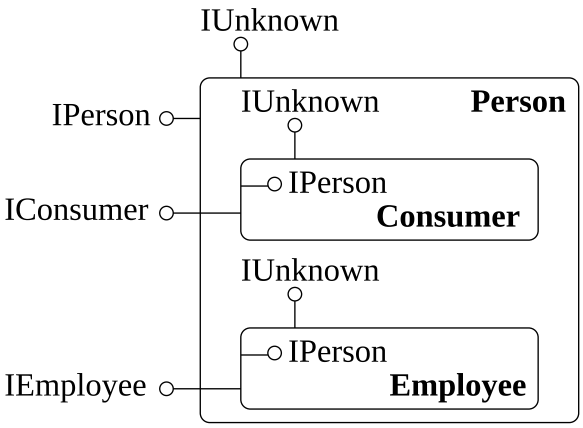

Figure 13. The Consumer and Employee roles are aggregated into the Person object.

The COM implementation of the framework concept has some limitations. The COM model defines frameworks as aggregates of the completed objects created at run-time, while a general framework model allows us to use incomplete objects (at run-time) or classes (at build-time).

3.2. Configuration Management

Using OOD frameworks instead of traditional objects yields several advantages, but it also introduces an additional level of complexity when building these frameworks. Frameworks are composite types of entities – they have an internal structure which is built from objects, or from parts of them. A framework entity also has relations to other frameworks, and can be composed from other (sub)frameworks. The definition and creation of such a composite entity introduces configuration problems. Some of them will be illustrated here for a COM implementation.

Let us consider the following cases:

- Sharing objects in several frameworks;
- Composing frameworks from objects and frameworks.

3.2.1. Sharing Objects in Several Frameworks.

Suppose framework F_1 includes objects O_1 and O_2 with a relation R_{12} between them, and framework F_2 contains objects O_1 and O_3 with a relation R_{13}. The object O_1 is shared by two frameworks:

$$F_1 = \{O_1\ O_2\ ;\ R_{12}\},\ \ F_2 = \{O_1\ O_3\ ;\ R_{13}\} \tag{1}$$

Suppose we now add a new property to the object O_1, a property that is required in (an improved version of) framework F_2. This creates a new version of the object $O_{1;v2}$, ($v2$ denotes the new version) which is included into the framework F_2:

$$F_2 = \{O_{1;v2}\ O_3\ ;\ R_{13}\}$$

However, if we do not take versioning into consideration, then the framework specifications will remain the same. In this case, we can be aware of the change of the object O_1 in the context of framework F_2, but not necessarily in that of F_1. Our specification of F_1 is defined by (1), but in reality we have

$$F_1 = \{O_{1;v2}\ O_2\ ;\ R_{12}\}$$

If the role of the object $O_{1;v2}$ used in F_1 is changed, then the behaviour of F_1 will be changed unpredictably, and a system using F_1 can fail. To avoid these unpredictable situations we can introduce basic configuration management methods – a version management of objects and configuration of frameworks [7]:

- An object is identified by its name and version.
- A framework is identified by a name and a version. A new framework version is derived from object versions included in the framework.

These rules imply that new versions of frameworks will be configured when a new object version is created, as shown in our example:

$$F_{1;vi} = \{O_{1;vm}\ O_{2;vn}\ ;\ R_{12}\},\quad F_{2;vk} = \{O_{1;vm}\ O_{3;vk}\ ;\ R_{13}\}$$

$$F_{1;vi+1} = \{O_{1;vm+1}\ O_{2;vn}\ ;\ R_{12}\},\quad F_{2;vk+1} = \{O_{1;vm+1}\ O_{3;vk}\ ;\ R_{13}\}$$

As several frameworks can share one object, and a framework can contain several objects, the number of generated frameworks can grow explosively. It is, however, possible to limit the number of interesting configurations. Typically, in a development process, we would implement the changes on all the objects we want, collect the versions of objects we want in a baseline and derive the frameworks from the baselined object versions. In such a case, experience for similar cases [2] shows that the number of derived entities does not necessarily grow rapidly.

A shared object is not necessarily completely shared, but different parts of the object, defined by the object's roles, are used in the frameworks. In the COM implementation a complete object will be included, but a part of it will be used. In a general framework model, a class (or an object at run-time) includes only those parts which are specified in the object's role. When we define a new role for an object in a framework or re-define the existing one, we need to change a specific part of the object class. We call this specific part an object aspect. The change of an object aspect will affect only those frameworks where the aspect is included. Other frameworks, though containing the object (or part of it), are not affected by the change. In this case, it is better to keep version control on the aspect level, and relate a framework configuration to the object aspects.

If we declare an aspect as a subset of an object $A_i(O_k) \subseteq O_k$, then an object version is defined as a set of aspect versions:

$$O_{i;vk} = \{A_{j;vl}\}$$

and a framework version is defined as a set of aspect versions with relations between the aspects:

$$F_{vk} = \{\{A_{j;vl}(O_{i;vk})\}\ ;\ R_{jl}\}$$

Having control over changes on the aspect level, we can gain control over the changes on the framework level. Now we can more precisely identify the frameworks being affected by changes in object roles.

3.2.2. Composing Frameworks from Objects and Frameworks.

In the framework model it is possible to compose new frameworks from existing frameworks. A new framework is a superset of the classes and relations from the frameworks involved. If a new framework is created at run-time, as in a COM implementation, then the objects from the selected frameworks comprise the new framework. The following example illustrates the merging process of two frameworks F_1 and F_2 into $F3$:

$$F_1 = \{O_1\ O_2\ ;\ R_{12}\},\quad F_2 = \{O_1\ O_3\ ;\ R_{13}\},\quad F_3 = \{O_1\ O_2\ O_3\ ;\ R_{12}, R_{13}, R_{23}\}$$

The composition works fine as long as we do not need to consider the changes of objects within one framework.

Suppose we create a new object version (or a new object aspect version) in F_2 and keep the old version of the same object in F_1:

$$F_{1;vi} = \{O_{1;v1}\ O_{2;vk}\ ;\ R_{12}\},\quad F_{2;vj} = \{O_{1;v1+1}\ O_{3;vl}\ ;\ R_{13}\}$$

In the merging process we have to recognise if different versions of the same objects are included in the frameworks being merged. If that is the case, we have two possible solutions:

- Selecting one specific version of the object (for example the latest):

$$F_{3;v1} = \{O_{1;v1+1}\ O_{2;vk}\ O_{3;v1}\ ;\ R_{12}, R_{13}, R_{23}\}$$

- Selecting both versions and enable their consistence in the new framework:

$$F_{3;v1} = \{O_{1;v1}\ O_{1;v1+1}\ O_2; vk\ O_{3;v1}\ ;\ R_{12}, R_{13}, R_{23}\}$$

For the second case there must be support for identifying object versions. This support can be provided by introducing an identification interface [14] as the standard interface of an object. There must also be support for managing different versions of the same object in the running system.

4. Conclusion

In this paper we have shown how to formally specify OOD frameworks using MDTL and event structures. In particular, we have shown a semantics for composing OOD frameworks with state transitions in the manner depicted in Figure 3.

Our work here is closely related to TROLL [11], which is used for specifying large distributed/concurrent object systems, and to [4], which formalises an algebraic semantics for object model diagrams in OMT [22]. 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). Their semantics is based on initial theories, as opposed to isoinitial theories that we use.

Overall, our approach to specification 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.

We have also presented a possible implementation of OOD frameworks using the COM technology. This implementation has some limitations, and we need to do further work to investigate how to improve this implementation.

Finally we have discussed configuration management for frameworks. We emphasise a need for using configuration management methods for managing frameworks as composite objects. The configuration management issues are complicated and need further investigation: Questions of managing relations, concurrent versions of frameworks, inclusion of change management [9], etc., must be addressed. Since aspects and objects are not entities recognised by standard configuration management tools (which recognise entities such as files, directories, etc.), new, semantic-based rules must be incorporated into such tools. For different object-oriented and component technologies, different tools have to be made. How different do they need to be, and are there possibilities to define common rules and implementation? These are questions for future investigation.

Acknowledgements

We would like to thank the referee who pointed out some minor mistakes.

References

[1] M. Abadi and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag.

[2] U. Asklund, L. Bendix, H.B. Cristensen, and B. Magnusson (1999). The unified extensional versioning model. In J. Estublier, editor, *Proc. System Configuration Managemnt SCM-9*, pages 17–33, Springer.

[3] A. Bertoni, G. Mauri, and P. Miglioli (1983). On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2):127–170.

[4] R.H. Bourdeau and B.H.C. Cheng (1995). A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, **21**(10):799-821.

[5] D. Box (1998). *Essential COM*. Addison-Wesely.

[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes (1994). *Object-Oriented Development: The Fusion Method*. Prentice-Hall.

[7] R. Conradi and B. Westfechtel (1998). Version models for software configuration management. ACM Computing Surveys, **30**(2):232–282.

[8] S. Cook and J. Daniels (1994). *Designing Object Systems*. Prentice-Hall.

[9] I. Crnkovic (1997). Experience with change oriented SCM Tools. In R. Conradi, editor, *Proc. Software Configuration Management SCM-7*, pages 222–234, Springer.

[10] D.F. D'Souza and A.C. Wills (1998). *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley.

[11] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger and H.-D. Ehrich (1998). 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.

[12] R. Helm, I.M. Holland, and D. Gangopadhay (1990). Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* **25**(10) (*Proc. ECOOP/OOPSLA 90*).

[13] J. Küster Filipe (2000). Fundamentals of a module logic for distributed object systems. *J. Functional and Logic Programming* **2000**(3).

[14] M. Larsson and I. Crnkovic (1999). New challenges for configuration management. In J. Estublier, editor, *Proc. System Configuration Management SCM-9*, pages 232–243, Springer.

[15] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills (1998). Interacting frameworks in *Catalysis*. In J. Staples, M. Hinchey and S. Liu, editors, *Proc. Second IEEE Int. Conf. on Formal Engineering Methods*, pages 110-119, IEEE Computer Society Press.

[16] K.-K. Lau and M. Ornaghi (1998). 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.

[17] K.-K. Lau and M. Ornaghi (1998). 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.

[18] K.-K. Lau and M. Ornaghi (1998). OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR'98*, *LNCS* **1559**:101-123, Springer-Verlag.

[19] R. Loogen and U. Goltz (1991). Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae* **XIV**(1):39–73.

[20] R. Mauth (1996). A better foundation: development frameworks let you build an application with reusable objects. *BYTE* **21**(9):40IS 10-13.

[21] R. Pooley and P. Stevens (1999). *Using UML: Software Engineering with Objects and Components*. Addison-Wesley.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.

[23] F.W. Vaandrager (1989). A simple definition for parallel composition of prime event structures. Technical Report CS-R8903, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

[24] G. Winskel and M. Nielsen (1995). Models for concurrency. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications.