Formal Specification of Catalysis Frameworks

Juliana Küster Filipe
TU Braunschweig, Germany
J.Kuester-Filipe@tu-bs.de

Kung-Kiu Lau
U Manchester, UK
kung-kiu@cs.man.ac.uk

Mario Ornaghi
U Milano, Italy
ornaghi@dsi.unimi.it

Kenji Taguchi
Uppsala U, Sweden
kenji.taguchi@dis.uu.se

Hirokazu Yatsu
Nihon Unisys Ltd., Tokyo, Japan
Hirokazu.Yatsu@unisys.co.jp

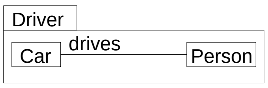
Alan Wills
Trireme Ltd., Manchester, UK
alan@trireme.com

Abstract

Frameworks are increasingly recognised as very useful components in the emerging paradigm of component-based software development (CBD). They are widely accepted as better units of reuse than objects. The CBD methodology Catalysis*, for instance, uses frameworks. However, at present,* Catalysis *frameworks are described only informally, which means we cannot reason formally about frameworks, in particular their composition, and thereby their reuse. For this we would need to specify frameworks (and their composition) formally. In this paper, we describe our approach for doing so.*

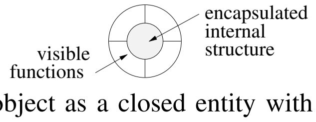
1. Introduction

In Object-oriented Design (OOD), a framework is a group of (interacting) objects. For example, in the CBD (Component-based Development) methodology *Catalysis* [5], a driver may be represented as the framework:

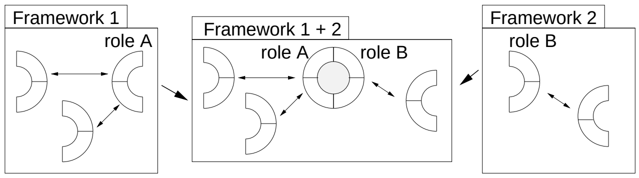


A driver is a person who drives a car, or in Object-oriented Design (OOD) terminology, a driver is a framework composed of a car object and a person object, linked by a 'drives' association (or attribute).

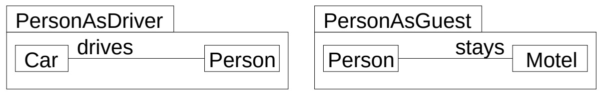
Frameworks are increasingly recognised as very useful components in the emerging paradigm of CBD (see e.g. [20]), in particular as better units of reuse in software development than objects (see e.g. [9, 16]). The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and frameworks can capture this, whereas existing OOD methods (e.g. Fusion [3] and Syntropy [4]) 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 below:



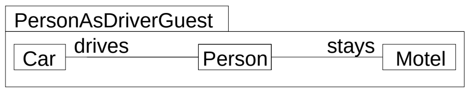
which regards an object as a closed entity with one fixed role. On the other hand, frameworks allow objects that play different roles in different frameworks to be composed by composing frameworks. In *Catalysis*, this is depicted as:



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 frameworks:



If we compose these two frameworks, then we get the PersonAsDriverGuest framework:



In this framework, a person object plays two roles.

However, in this example, the only interactions between objects are associations or relations (e.g. 'drives' and 'stays' in the above examples) between objects. In practice, objects can also interact with other objects via their methods. Moreover, objects in different frameworks can also interact with one another via their methods. Therefore, in order to develop and reuse frameworks, we need to reason about their interactions and thereby their composition.

In *Catalysis*, at present, frameworks are described only informally, and so are intra- and inter-framework interactions.[1] This means that we cannot reason formally about frameworks, in particular their interactions and composition. In this paper, we describe our approach to formal specification of *Catalysis* frameworks.

[1] The former in UML [18], and the latter by pre- and post-conditions expressed in a natural language or in OCL [21].

2. Catalysis Frameworks

In this section, we show how to specify *Catalysis* frameworks. Since details of formal specifications are usually not easily readable, we will not present them all. Instead we will explain them informally, but hopefully clearly.

A *Catalysis* framework is a composite class (in the usual OO sense) with state transitions. We specify classes as (open) ADT theories (with signatures and axioms), and objects as (closed) instances of classes, in the standard way.

Example 2.1 The Car class in the previous example can be specified as follows:

Class $\mathcal{CAR}(.km, .option)$;

IMPORT: $\mathcal{INT}, year96$;

DECLS: $.km \;:\; [\,] \rightarrow Int$
$.option \;:\; [year96.opts]$

CONSTRS: $.km \geq 0$

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 $.km$.

To create objects we instantiate a class by adding axioms. These axioms represent the *state* of the object, and are called *state axioms*. State axioms can be updated, by redefinition for instance. For example, an object called $spider$ of class $\mathcal{CAR}$ can be created by:

NEW $spider : \mathcal{CAR}$;
$spider.km = 25000$;
$spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond.$

Note that $spider.km$ is a constant (integer) and $spider.option(x)$ is a predicate.

A framework can be specified as a composition of classes.

Example 2.2 The PersonAsDriver framework in the previous section can be specified as follows:

Framework $\mathcal{DRIVER}[\mathcal{PERSON}, \mathcal{CAR}]$;

DECLS: $.drives : [obj]$;

CONSTRS: $\hat{X}.drives(c) \rightarrow PERSON(\hat{X}) \wedge$
$\hat{X}.age \geq 18 \wedge CAR(c)$;
$(\exists c : obj)(.drives(c))$;

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

Class $\mathcal{PERSON}$;

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.

2.1. State Transitions

To specify state transitions, we use a first-order temporal logic. The state of an object comprises the current values of its attributes, and the current status of its methods. We will refer to methods as *actions*, in order to have a more uniform terminology, as should become obvious later. Actions may be either *enabled* or *occurring* in a particular state. If an action is *enabled*, then it may occur in the next state. When an action occurs, the state of the object changes. We use $\odot a$ to denote the occurrence of action a, and $\rhd a$ that the action a is enabled. If an action occurs, then it must have been enabled in the previous state: $\odot a \Rightarrow Y \rhd a$. In this formula, Y is the temporal operator *Yesterday* referring to the previous state. Enabling ($\rhd$) is useful for expressing *preconditions*, and occurrence ($\odot$) for expressing *postconditions*.

We 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 a framework is given by the states of the current objects belonging to the framework.

Example 2.3 Consider the *Catalysis* framework for employees depicted in Figure 1, in which a person plays the

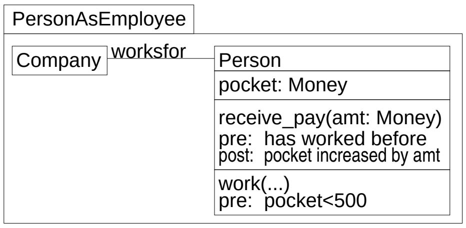


Figure 1. PersonAsEmployee framework.

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 (precondition for $receive\_pay$). If a person receives a payment, the money in his pocket increases by the amount received. We omit the parameters and the postcondition of $work$, as well as the definition of the class for Company.

The pre- and post-conditions for the actions $receive\_pay$ and $work$ can be specified as state transition axioms (ST-AXIOMS) for the class Person as follows:

Class $\mathcal{PERSON}$;

IMPORT: $\mathcal{MONEY}$

DECLS: $.pocket \;:\; [\,] \rightarrow Money$;
$.receive\_pay \;:\; [Money]$;
$.work \;:\; [\ldots]$;

AXIOMS: ...

ST-AXIOMS:

$\forall_a \rhd .receive\_pay(a) \Rightarrow P \odot .work(\ldots);$

$\forall_{a,n} \odot .receive\_pay(a) \Rightarrow Y(.pocket = n) \Rightarrow .pocket = n + a;$

$\rhd .work(\ldots) \Rightarrow .pocket < £500.$

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 $work(\ldots)$ 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 by:

NEW $joe : \mathcal{PERSON}$;
$joe.pocket = £100$;
$\rhd joe.work(\ldots)$;
$\neg \rhd joe.receive\_pay(a)$.

with $work$ enabled and $receive\_pay$ disabled.

2.2. Event Structures

It is not enough to have pre- and post-conditions expressed as state transition axioms. To reason about these axioms, we need to give them semantics. We do so with *event structures*. Such a structure describes 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*. Each event has a label denoting its state.

Example 2.4 Consider the event structure in Figure 2. It

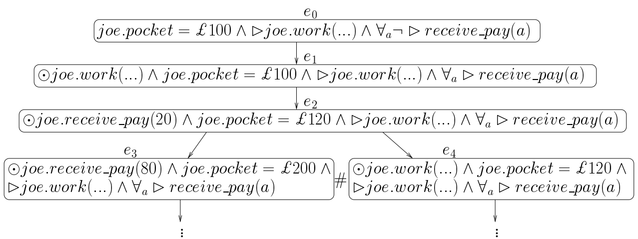


Figure 2. Event structure for joe as an employee.

shows a small part of a (sequential) 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.

There are therefore two life cycles for joe in Figure 2. 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.

2.3. Composing Event Structures

In order to create objects by composing frameworks, 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 framework in Figure 3 defines this role for a person object. The class Person here

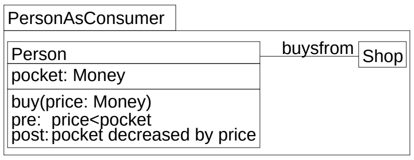


Figure 3. PersonAsConsumer framework.

can be defined as 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 class 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 4, we show

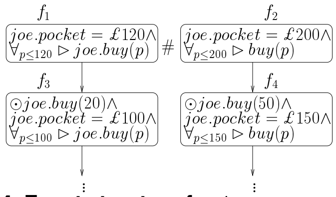


Figure 4. Event structure for joe as a consumer.

two possible life cycles with distinct initial states.

We may compose the 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 5. The composite PersonAsEmployeeConsumer framework contains the union of the state transition axioms of its component frameworks. An event structure, i.e. a model, for a person as an employee and consumer is obtained by

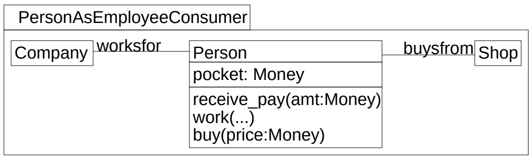


Figure 5. PersonAsEmployeeConsumer framework.

composing a model for person as an employee with one for person as a consumer in a special manner.

What we 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 6 shows the composed model for joe as an employee and consumer based on the models of Figures 2 and 4 for joe as an employee and joe as a consumer respectively. For simplicity, the states show only the value of the

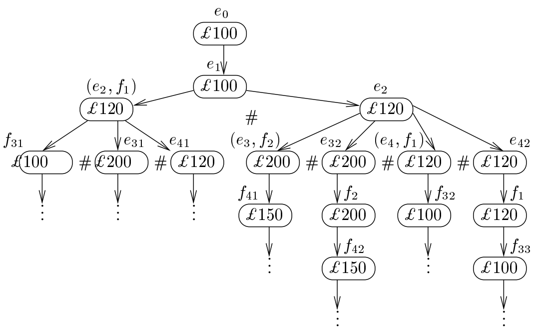


Figure 6. Event structure for joe as an employee and a consumer.

attribute $pocket$. In this 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 6 show just interleaving.

3. Interacting Frameworks

So far, the only form of interactions between objects are associations or relations (e.g. 'worksfor' in Figures 1 and 5, 'buysfrom' in Figures 3 and 5) between objects, and operations on objects which only affect their own local state (e.g. actions 'receive_pay', 'work' and 'buy' in Figures 1, 3 and 5). In practical systems, however, a framework $\mathcal{F}$ may also need to express interactions *between* objects within $\mathcal{F}$, as well as interactions between $\mathcal{F}$ and other frameworks. In this section, we specify such intra- and inter-framework interactions in *Catalysis*, and hence show that we can also specify the composition of frameworks with such interactions. Interacting frameworks in particular are a very powerful modelling tool.

Due to the complexity involved, we will base our presentation on one specific *Catalysis* example, viz. the TradeSupply framework shown in Figure 7. A Retailer is anyone who is in the retail trade (TradeSupply does not specify who a Retailer sells to). Distributors sell only to Retailers. An action $make\_order$ is provided whereby the Retailer can create Orders. The Distributor can $deliver$ Products, fulfilling an Order, and the Retailer can pay for them. The (informal) postconditions of $make\_order$ and $deliver$ show the effects on both the Retailer and Distributor involved in the interaction.

Actions in *Catalysis* are classified as *internal* or *external* according to whether they can be invoked from within or from outside the framework. For example, in TradeSupply, $make\_order$, $deliver$ and pay are *internal* actions, whereas $depletion$ is an *external* action (and is specified in a separate 'external section'). External actions are the means by which frameworks interact with one another.

3.1. Internal Actions

In Catalysis, internal actions involving several objects in the same framework are called *joint actions*.[2] For example, in TradeSupply, the actions $make\_order$, $deliver$ and pay, between Retailer and Distributor are joint. Joint actions in a framework are 'global', in the sense that they involve more than one class in the framework, as opposed to *local actions* such as 'receive_pay', 'work' and 'buy' in Figures 1, 3 and 5 which only affect the local state of an object. Therefore, joint actions are declared at the level of a framework, whereas local actions are declared at the level of a class.

To specify the internal actions of a framework, we therefore need to specify state transition axioms for both joint and local actions in the framework. We write a for a joint action (in a framework), $.b$ for a local action b (in a class), and $i.b$ for a local action b of the object i. We have seen examples of specifications of local actions in Section 2, so now we concentrate on (internal) joint actions.

For example, the ST-AXIOM for $make\_order$ is:

$$\odot make\_order(rtl, dis, pro, q) \Rightarrow$$
$$Y(dis.nextOutOrderNr = n \wedge rtl.orders = lt \wedge$$
$$dis.outstanding = ld) \Rightarrow$$

[2]Joint actions are also called *abstract operations* in older versions of *Catalysis* terminology.

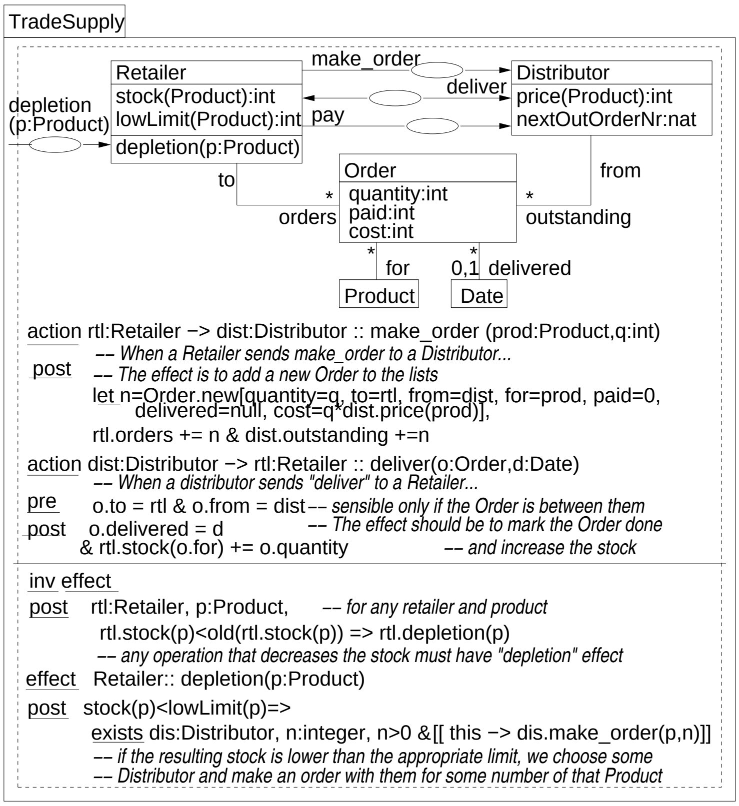


Figure 7. The TradeSupply framework.

$$\odot dis(n).new\_order(q, q \times dis.price(pro))\wedge$$
$$dis(n).to = rtl \ \wedge dis(n).from = dis\wedge$$
$$dis(n).for = pro \wedge dis(n).delivered = null \wedge$$
$$rtl.orders = dis(n)|lt \wedge dis.outstanding = dis(n)|ld \wedge$$
$$dis.nextOutOrderNr = n + 1;$$

This gives a postcondition for $make\_order$. The occurrence of $make\_order$ implies the creation of a new Order (for a given Product pro in quantity q) between a given Retailer rtl and Distributor dis. The identity of the new Order is given by the attribute $nextOutOrderNr$ (a natural number) of the Distributor dis, and is unique for a given Distributor: $dis(1), dis(2), \ldots$ are examples of Order identities where dis are identities of Distributor. The association values of the new Order with the remaining classes are set, e.g., $dis(n).from$ is set to dis. The local effect of the occurrence of action new_order at Order is specified in a state axiom of the corresponding class.

For lack of space, we do not discuss the ST-AXIOMS for $deliver$ and pay.

Figure 8 shows part of a model for TradeSupply, with object instances rtl for a Retailer, dis for a Distributor, and $dis(1)$ and $dis(2)$ for Orders. Here, for simplicity, the states show only the occurring actions. Event e_0 is a local event of object rtl, and corresponds to the occurrence of action $new\_retailer$ creating the new Retailer instance rtl. Similarly, event e_1 is a local event of object dis. Events e_0 and e_1 are concurrent (independent). In contrast, events e_2 and e_4 are shared events. At event e_2, the global action $make\_order$ involving rtl and dis occurs, as well as

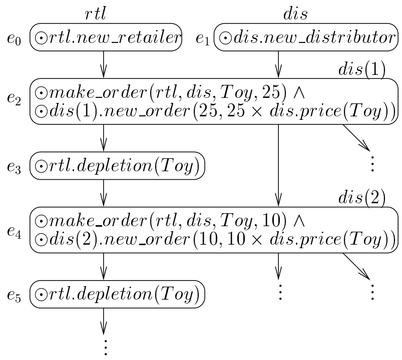


Figure 8. Event structure for retailer rtl, distributor dis, and orders $dis(1)$ and $dis(2)$.

the local action new_order creating $dis(1)$, a new instance of Order. Event e_2 is therefore a shared event of rtl, dis and $dis(1)$. Events e_3 and e_5 are local events of object rtl, where the action $depletion$ occurs. The complete label of event e_2 is:

$\odot make\_order(rtl, dis, Toy, 25)\wedge$
$\exists_{d:|Date|} \rhd deliver(dis, rtl, dis(1), d) \wedge$
%local state for rtl:
$rtl.stock(Toy) = 10 \wedge rtl.lowLimit(Toy) = 10\wedge$
$\rhd rtl.depletion(Toy) \wedge$
%local state for dis:
$dis.price(Toy) = 5 \wedge dis.nextOutOrderNr = 2 \wedge$
% local state for $dis(1)$:
$\odot dis(1).new\_order(25, 125) \wedge dis(1).quantity = 25 \wedge$

$dis(1).paid = 0 \wedge dis(1).cost = 125 \wedge$
$rtl.orders = \{dis(1)\} \wedge dis.outstanding = \{dis(1)\} \wedge$
$dis(1).to = rtl \wedge dis(1).from = dis \wedge$
$dis(1).for = Toy \wedge dis(1).delivered = null$

Since e_2 is a shared event, its label contains the local state of Retailer rtl, Distributor dis and Order $dis(1)$, as well as the global state for TradeSupply (global attribute (association) values and enabled/occurring global actions). A local event of an object at the framework level has a label containing its local state and association values. The local event e_3 of rtl has the following label:

$\odot rtl.depletion(Toy) \wedge rtl.stock(Toy) = 5 \wedge$
$rtl.lowLimit(Toy) = 10 \wedge \rhd rtl.depletion(Toy) \wedge$
$rtl.orders = \{dis(1)\}$

where the action $depletion$ checks for stock depletion against a preset lower limit (see opposite for the ST-AXIOMS for $depletion$).

The stock of Toy for Retailer rtl is below the limit at event e_3, and thus (according to the ST-AXIOMS for $depletion$) in the next state the action $make\_order$ has to occur between Retailer rtl and some Distributor, where rtl asks for some number of product Toy. The above state axiom holds at event e_3, since at event e_4 the action $make\_order$ does indeed occur.

3.2. External Actions

In TradeSupply, the action $depletion$ of Retailer is an external action (and represented as such in Figure 7). We can specify $depletion$ by the following ST-AXIOMS:

$Y(rtl.stock(pro) = n) \wedge rtl.stock(pro) < n \Rightarrow$
$\quad \odot rtl.depletion(pro);$

$\odot rtl.depletion(pro) \Rightarrow$
$\quad rtl.stock(pro) < rtl.lowLimit(pro) \Rightarrow$
$\quad X(\exists_{d:|Distributor|, n \in Int, n>0}$
$\quad\quad \odot make\_order(rtl, d, pro, n)).$

The first one states that if the stock of a Retailer for a given product has decreased, then the action $depletion$ must have occurred. The second axiom states that if the action $depletion$ of a Product pro has occurred for a given Retailer, and if pro's stock is below $lowLimit$, then $make\_order$ between the Retailer and some Distributor asking for some number of the Product occurs in the next state. Thus the $depletion$ action checks that whenever the stock is reduced it does not fall below a pre-determined lower limit.

In *Catalysis*, different frameworks may interact with each other by invoking external actions of one another. Such mutual invocations amount to composition of interacting frameworks, by which objects with multiple roles in distinct frameworks can be formed, as we have seen in Sections 1 and 2.3. In the next section, we will show how TradeSupply can be composed with another framework via its external operation $depletion$ (in Retailer).

3.3. Composing Interacting Frameworks

Consider the *Catalysis* framework PublicVending shown in Figure 9, in which a vendor sells to customers. Now

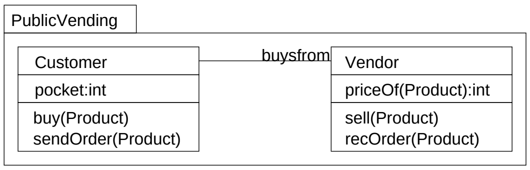


Figure 9. The PublicVending Framework.

consider a Shop as an object which plays the roles of both a retailer and a vendor, formed by combining a Retailer object in TradeSupply with a Vendor object in the PublicVending framework. That is, we can compose the frameworks TradeSupply and PublicVending into the composite framework shown in Figure 10 (with simplifications). In

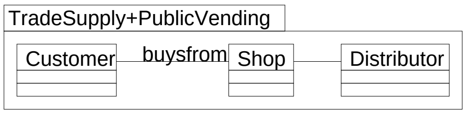


Figure 10. The TradeSupply+PublicVending Framework.

the composed framework, a Shop as a Vendor (in the PublicVending framework) selling products to consumers will reduce the stock of the Shop as a Retailer (in the TradeSupply framework). Any such action in the PublicVending framework (that reduces the stock) must of course conform to the $depletion$ conditions specified in the TradeSupply framework.

The ST-AXIOMS of PublicVending can be formalised as follows:

$\odot c.buy(p) \wedge c.buysfrom = vnd \Rightarrow \odot vnd.sell(p)$

$\odot c.sendOrder(p) \wedge c.buysfrom = vnd \Rightarrow$
$\quad F \odot vnd.recOrder(p).$

The first axiom expresses synchronous communication between the Customer c and the Vendor vnd. The occurrence of the action buy for Customer c implies the synchronous occurrence of action $sell$ for the Vendor where the Customer buys from, namely vnd. The second axiom expresses an asynchronous communication among the classes. If a given Customer sends an Order to the Vendor where he buys his products from, then the Vendor will some time later receive it. Both state transition axioms thus denote internal framework interactions.

Figure 11 shows part of a model for PublicVending, with object instances c for a Customer and vnd for a Vendor. The states in the model are just indicated by the occurring actions for simplicity. Event g_0 is a local event of Customer object c, and corresponds to the occurrence of action $new\_customer$ creating the new Customer instance c. Similarly, event g_1 is a local event of Vendor object vnd. Except for event g_2 all events in the model are local. Event g_2 denotes the synchronous communication between Customer

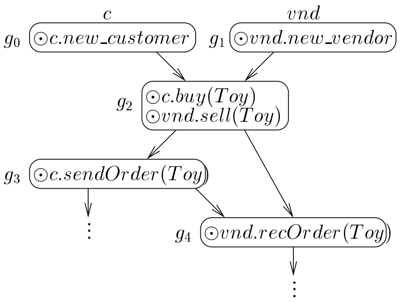


Figure 11. Event structure for PublicVending with a Customer c and a Vendor vnd.

c and Vendor vnd, and actions buy and $sell$ occur simultaneously. Event g_3 is a local event of Customer object c, and denotes the occurrence of action $sendOrder$ starting an asynchronous communication with the Vendor vnd. The causality between events g_3 and g_4 denote the asynchronous communication among the objects in the model. If event g_4 occurs, then some time before, event g_3 must have occurred.

Let $shop$ be an object playing the role of a Vendor in the framework PublicVending, and the role of a Retailer in TradeSupply. An event structure for $shop$ can be formed by composing the models in Figure 8 and Figure 11 with the instance $shop$ instead of rtl and vnd respectively. However, the composition now has to observe additional *communication formulae* that specify inter-framework interactions.

The first communication formula

$$PublicVending :: \odot shop.sell(p) \Rightarrow$$
$$TradeSupply :: \odot shop.depletion(p)$$

describes a synchronous communication between PublicVending and TradeSupply. It states that if object $shop$ in PublicVending sells a product p it corresponds to the depletion of the same product for object $shop$ in the role of a Retailer in TradeSupply. The occurrence of action $sell$ for object $shop$ in PublicVending causes a (synchronous) calling of the action $depletion$ in TradeSupply to occur.

In contrast, the second formula describes an asynchronous communication between the two frameworks:

$$PublicVending :: \odot shop.recOrder(p) \Rightarrow$$
$$F\ TradeSupply :: \odot shop.depletion(p)$$

It states that if object $shop$ in PublicVending receives an Order for a Product p it will eventually correspond to the $depletion$ of the same product for object $shop$ in the role of a Retailer in TradeSupply.

In general, communication formulae for interacting frameworks constrain the behavioural model of their composition. For lack of space, we do not discuss these in detail.

Anyway, combining the models of Figure 8 and Figure 11 according to these communication formulae gives the event structure for $shop$ shown in Figure 12. Here, states are indicated only by the occurring actions for simplicity. Details concerning Orders $dis(1)$ and $dis(2)$ have been omitted.

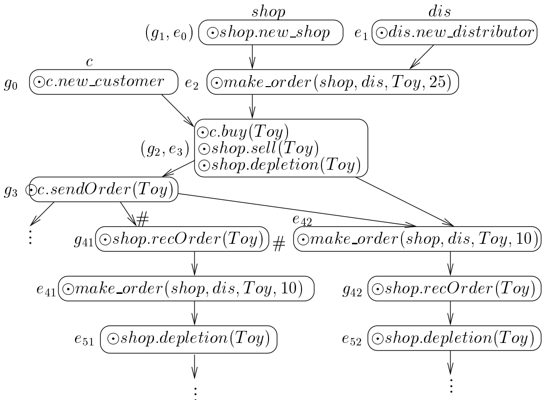


Figure 12. Event structure for $shop$.

The composed event structure of Figure 12 is obtained by parallel composition of the event structures of the separate frameworks with synchronisation and interleaving for $shop$ events. Local events of Customer c and Distributor dis are left concurrent, whereas $shop$ events are interleaved and/or synchronised. Event (g_1, e_0) denotes the simultaneous creation of object $shop$ as a vendor and as a retailer. After being created, the behaviour of $shop$ in both roles is interleaved or synchronised. Event (g_2, e_3) corresponds to a synchronisation, and reflects the simultaneous occurrence of $sell$ and $depletion$ for $shop$, which is imposed by the first communication formula. The second formula (an asynchronous calling) constrains the model by forcing the occurrence of $recOrder$ (events g_{41} and g_{42}) to precede the occurrence of $depletion$ (events e_{51} and e_{52}). It constrains the possible interleaving of $shop$ events.

Several constructions for sequential and parallel composition of event structures have been defined in the literature, e.g. [22, 15]. None of these offers exactly what we need here: sequential composition with synchronisation for events of the same object in distinct roles; parallel composition for events of distinct objects.

4. Conclusion

In [14, 12], we considered frameworks with only ADTs, i.e. no states and hence no objects. In this paper, we have extended this work to the formal specification of *Catalysis* frameworks. For lack of space, we have given here just an informal, but hopefully clear, account with the help of examples. The foundations of the formal semantics that underlies our work are reported elsewhere [13, 10]. This semantics is model-theoretic, and therefore stands in contrast to the type-theoretic approach by Abadi & Cardelli [1] with regard to the characterisation of states and objects.

The work that is most closely related to ours is that of Grau *et al* [7] and Bourdeau & Cheng [2]. [7] uses the TROLL language, based on a distributed temporal logic, to specify (concurrent) distributed object-oriented systems. The logic we have used here is an extension of the TROLL logic.

[2] formalises object models in OMT [17]. Basically they translate object model diagrams into algebraic specifications (and semantics) expressed in Larch [8].

Our specification method for frameworks extends both [7] and [2], which focus on single objects, rather than frameworks, i.e. groups of interacting objects. In other words, they consider objects as the unit of reuse, whereas our formalisation supports frameworks as the unit of reuse. Industry is increasingly recognising that objects are not the best focus for design, and frameworks are becoming widely used as the basic unit of reuse.

Our formalisation of joint actions between objects in a framework and of interactions between frameworks is, as far as we are aware, completely novel.

Other approaches to formal semantics for object-oriented methodologies are mainly so-called integrated methods. For example France *et al* [6] derives Z [19] specifications from Fusion [3] diagrams. At present, such methods do not have completely formal semantics. In any case, like [7] and [2], they also use objects as the basis unit of reuse.

Finally, the development and reuse of frameworks is an important topic in CBD (see, e.g. [11]). The main contribution of this paper is to provide the formal foundations for developing and reusing frameworks in *Catalysis*. We are currently continuing our investigation into all relevant aspects of *Catalysis* frameworks, in particular their composition and reuse. One topic is the precise nature of event structures, in particular their compositionality under various conditions. Here, we expect we will need to examine aspects not hitherto considered in the literature, that are relevant to framework interaction and composition.

References

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

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

[3] D. Coleman *et al*. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[4] S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.

[5] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[6] R.B. France *et al*. Towards a rigorous object-oriented analysis and design method. *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 7-16, IEEE Computer Society, 1997.

[7] A. Grau *et al*. The TROLL approach to conceptual modelling: syntax, semantics and tools. *LNCS* 1507:277-290, Springer, 1998.

[8] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[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, K.-K. Lau, M. Ornaghi, and H. Yatsu, On dynamic aspects of OOD frameworks in component-based software development in computational logic. *LNCS* 1817:43-62, Springer-Verlag, 2000.

[11] G. Larsen. Designing component-based frameworks using patterns in the UML. *CACM* 42(10):38–45, 1999.

[12] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting frameworks in *Catalysis*. In *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*, pages 110-119, IEEE Computer Society, 1998.

[13] K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. *LNCS* 1559:101-123, Springer-Verlag, 1999.

[14] K.-K. Lau, M. Ornaghi, and A. Wills. Frameworks in *Catalysis*: Pictorial notation and formal semantics. In *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 213-220, IEEE Computer Society, 1997.

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

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

[17] J. Rumbaugh *et al*. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[19] J.M. Spivey. *The Z Notation*. Prentice Hall, 2nd edition, 1992.

[20] C. Szyperski. *Component Software*. Addison-Wesley, 1997.

[21] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.

[22] G. Winskel. Event structures. *LNCS* 255:325–392, Springer, 1987.