Correct OO Systems in Computational Logic

Kung-Kiu Lau[1] and Mario Ornaghi[2]

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

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

Abstract. Object Oriented Design Frameworks (OOD frameworks) are groups of interacting objects. We have formalised them in computational logic as open systems of interacting objects. Our formalisation is based on steadfast logic programs in the context of open specification frameworks. However, we have considered only the static aspects, namely the specification of constraints and the correctness of queries (programs that do not update the current state). In this paper we extend this static model, by introducing actions that update the current state.

1 Introduction

In Object Oriented Analysis and Design (OOA&D), the traditional unit of design is of course the class. However, the observation that in the real world, design artefacts are rarely about single classes, but about groups of objects and the way they interact, has led to the introduction of *OOD frameworks*, which represent such groups of interacting objects. It is widely recognised that OOD frameworks are better design (and reuse) units than single classes.

Different characterisations of OOD frameworks as components have been proposed, using UML (Unified Modelling Language) [19]), e.g. *frameworks* in Catalysis [7] and *Komponents* in KobrA [2]. These approaches model OOD frameworks as UML subsystems. However, they do not define OOD frameworks (or components) precisely or formally.

A natural candidate for formalising OOD frameworks in UML would be the Object Constraint Language (OCL) [22]. OCL aims to become a standard for the formal specification of OO systems. It is being formalised in its various aspects [17] and there are Java tools (e.g., [6]), but its syntax and semantics are not yet fixed in a definitive and stable way.

Independently, we have been working on the formalisation of OOD frameworks in computational logic [12, 14]. In [14] we started with a more logically oriented formalisation of OO systems, because we believe that computational logic can offer a solid base for a formalisation of OO systems (and hence OOD frameworks). Moreover, we believe that OCL would benefit from a separate and more logically oriented approach. It would inherit a lot of knowledge, like Abstract Data Types, temporal logic, proof theoretical tools, and so on. We are

M. Bruynooghe (Ed.): LOPSTR 2004, LNCS 3018, pp. 34–53, 2004.

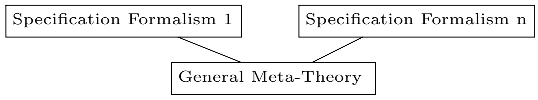

Fig. 1.

looking for a situation like the one depicted in Fig.1, where we can design specification formalisms for OO systems, on top of a meta-theory, i.e., a sound and general underlying logic formalisation.

To formalise OOD frameworks as open systems of interacting objects, we started from our previous work on steadfast programs in the context of open specification frameworks [15]. A specification framework is an open first-order theory that formalises a class of intended models. Specification frameworks can be instantiated (or reused) by theory morphisms, like in algebraic ADT's [23], and steadfast programs are open programs that correctly instantiate in every framework instance. Thus steadfast programs have the properties desired for the methods of correctly reusable units, and specification frameworks specifying and containing steadfast programs become good candidates for the formalisation of OOD frameworks.

In OO systems, instances are systems of interacting objects, that we call object-level instances. To give a precise definition of object-level instances, while preserving the features of open steadfast programs, we designed a three-tier approach, with an object, a specification and a meta level. The object level gives the semantics for our object-level instances. The specification level allows us to specify OO systems. The meta-level allows us to reason about OO systems, in particular, to reason about steadfast programs.

Although our aim was not the formalisation of UML, our three-level structure turns out to be very close to the multi-level approach of UML. The UML object and model levels correspond to our object and specification levels, while our meta-level has the role of the meta-model and meta-meta-model levels of UML. As a consequence, the approach developed in [14] allows us to formalise class and object UML diagrams in the context of computational logic. However, in that paper, we could only reason about the open correctness of queries, that is, open logic programs that do not update the current system state. In this paper we extend our approach by adding the ability to specify programs that update the current state. Moreover, we generalise our three-tier approach. We consider the meta level and the object level as general meta-theory of Fig.1, and we allow many different specification formalisms, to be built on top of it.

This paper is organised as follows. First we briefly review our past work in Section 2, and we discuss the general features of our approach. In Section 3, we introduce the object level, and we discuss the object-level semantics. In Section 4, we introduce the meta-level, we introduce projections (suitable semantic maps between object and meta levels). All this is used in Section 5, to show how intermediate specification formalisms, tailored to different aspects, can be defined, together with their object- and meta-level semantics. The paper is centred on the basic features of OO systems, which can provide a basis for important aspects like classes, polymorphism and modularity, as briefly discussed in the conclusion.

2 Static Aspects of OO Systems

In this section we give an overview of the static approach to OO systems given in [14], where OO systems are formalised on the basis of open steadfast programs and specification frameworks [15, 11].

A specification framework is an open first-order theory, where we distinguish *open* and *defined* symbols, and we have two kinds of axioms: *constraint* and *definition* axioms. The former constrain the possible interpretations of the open symbols, and the latter fix the intended meaning of the defined symbols in terms of the open ones. It may happen that the meaning of some defined symbols is completely fixed, independently from the open symbols. These symbols will be called *closed symbols*, and the axioms fixing their intended meaning will be called *closed axioms*.

A closed specification framework contains only closed symbols and closed axioms. In an open specification framework, the sub-signature of its closed symbols, together with their closed axioms, forms a closed sub-framework, that we will call the (closed) *kernel*.

The model-theoretic semantics of closed and open specification frameworks is based on isoinitial models [13], and we require the following *adequacy* [12] conditions.

A closed specification framework $\mathcal{F}$ is *adequate* if it has a *reachable isoinitial model* i. i is unique up to isomorphism and represents the intended meaning of the signature of $\mathcal{F}$. For an open specification framework $\mathcal{F}$, a *pre-interpretation* is an interpretation of the open and closed symbols, that interprets the closed ones according to the kernel and the open ones according to the constraints. $\mathcal{F}$ is *adequate* if for every pre-interpretation j, there is a *j-isoinitial model* of $\mathcal{F}$, indicated by $\mathsf{j}^{\mathcal{F}}$. $\mathsf{j}^{\mathcal{F}}$ is unique up to isomorphism and gives the interpretation of the defined symbols that corresponds to j. The *intended models* of $\mathcal{F}$ are the models $\mathsf{j}^{\mathcal{F}}$ determined by the pre-interpretations j. Finally, specification frameworks can be built incrementally, by means of *adequate extensions* [11].

Example 1. First-order arithmetic $\mathcal{NAT}$ is a closed specification framework. The standard structure $\mathcal{N}$ of natural numbers is the (unique up to isomorphism) isoinitial model of $\mathcal{NAT}$. It is the kernel of the following open framework, which axiomatises lists with generic elements X and a generic total ordering $\lhd$ on X:

SpecFramework $\mathcal{LIST}(X, \lhd)$ **with** KERNEL $\mathcal{NAT}$;
ODECL: $X : sort$; $\lhd : [X, X]$;
DDECL: $ListX : sort$;
$nil : [\,] \to ListX$; $. : [X, ListX] \to ListX$;
$\_@(\_,\_) : [X, Nat, ListX]$; $nocc : [X, ListX] \to Nat$;
DAX1: $\mathrm{FreeAx}(nil, . : ListX)$;

$$
\begin{array}{ll}
\textsc{dax}2: & \text{for } x, y : X, i : Nat, l, m : ListX : \\
& x@(0,l) \leftrightarrow \exists y, m \,.\; l = y.m \wedge x = y; \\
& x@(si,l) \leftrightarrow \exists y, m \,.\; l = y.m \wedge x@(i,m); \\
& nocc(x, nil) = 0; \\
& x = y \rightarrow nocc(x, y.l) = nocc(x,l) + 1; \\
& \neg x = y \rightarrow nocc(x, y.l) = nocc(x,l); \\
\textsc{constr}: & \lhd : TO(\lhd).
\end{array}
$$

X and $\lhd$ are declared as open in ODECL, and the defined symbols are declared in DDECL. The constraints CONSTR contain the total ordering axioms $TO(\lhd)$ for $\lhd$. The definition axioms DAX1 contain the *freeness axioms* [20] and induction for list constructors. DAX2 axiomatise $x@(i,l)$ (x occurs at position i in l) and $nocc(x,l)$ (number of occurrences of x in l).

A specification framework can be instantiated (or reused) by theory morphisms, and steadfast programs are open programs that correctly instantiate in every framework instance. Thus steadfast programs have the properties desired for the methods of correctly reusable units, and specification frameworks specifying and containing steadfast programs become good candidates for the formalisation of reusable units. A *reusable unit* is composed of an open *specification framework*, that formalises the problem context, a set of *specifications*, and a set of *steadfast* (i.e., correct) open programs. Each program predicate p has a declarative specification S_p, and each program P has a program specification $P : S_\pi \Rightarrow S_\delta$, which means that P correctly computes the predicates δ specified by S_δ, whenever it is composed with programs correctly computing π specified by S_π. Reusing a unit means instantiating it in the context of reuse. The inherited programs can be reused by composing them with other programs, according to their specifications.

Because of the properties of steadfast open programs, in [14] we considered reusable units as good candidates for a formalisation of OO systems. In OO systems, instances are systems of interacting objects, that we call object-level instances. The first problem here is to give a precise definition of object-level instances, while preserving the features of open steadfast programs. We studied different possibilities, that we briefly discuss here, together with their correspondence to UML static diagrams.

The simpler approach is the renaming approach, that is, the dot notation is considered as a renaming notation. This allows a simple and formal definition of the instance semantics by theory morphisms. For example, the left-hand side of Fig. 2 shows an UML diagram with a Car class and two Car objects; in our formalisation the Car class is an open unit, and the the two car objects are obtained by instantiating Car by the theory morphisms μ_{c1} and μ_{c2}, as shown in the right-hand side. At the class level, the attribute $.km$ is open and the invariant states that it must be greater than 0 in every instance. μ_{c1} renames $.km$ by $c1.km$ and closes it by the axiom $c1.km = 3050$, while satisfying the invariant. Similarly for the morphism μ_{c2}.

The renaming approach has a clean semantics in terms of theory morphisms, but has problems when considering associations and groups of related objects.

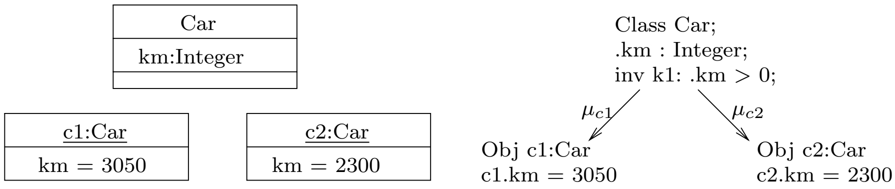

Fig. 2. Renaming.

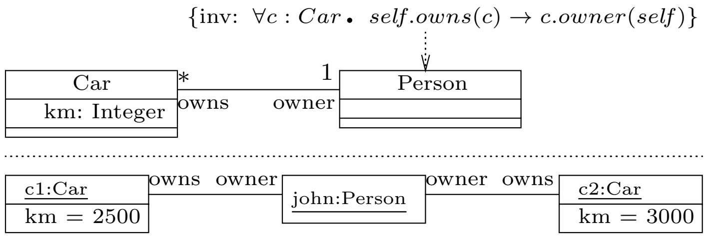

Fig. 3. Extra-argument.

For example, in the invariant inv of Fig.3 we have to quantify over c, but quantified renaming does not belong to the syntax and semantics of first order logic.

For this reason, usually dot is considered as a syntax for introducing c as an extra-argument,that is, $\forall c : \mathit{Car} \centerdot\ \mathit{self}.\mathit{owns}(c) \leftrightarrow c.\mathit{owner}(\mathit{self})$ represents a syntactical variant of $\forall c : \mathit{Car} \centerdot\ \mathit{owns}(\mathit{self}, c) \leftrightarrow \mathit{owner}(c, \mathit{self})$.

But extra-arguments also have problems. For example, by introducing an extra-argument, the $c1.km$ attribute becomes a function $km(c1)$ and it cannot be completely known unless we know all the Car objects. We believe that this is not natural in OO and in UML object diagrams, where an attribute like km should represent a *local* property of the single, isolated car object. Finally, in the extra-argument approach the semantics based on theory morphisms is lost.

Thus in [14] we came up with a many-level solution, with an object-level, a meta-level, and an intermediate specification level. At the *object level* dot works as renaming, each object has its own signature and a system can be considered as the union of its objects. This level corresponds to the object level of UML/OCL. At the *meta-level*, dot introduces an extra-argument of a special sort Obj, and an OO system $\mathcal{S}$ is formalised by a *meta-unit* $\mathcal{U}(\mathcal{S})$, which allows us to predicate over live objects and their classes, and to reason about steadfastness. Its framework has the role of the UML meta- and meta-meta-model. Since the syntax of the meta-level is redundant, we introduce the intermediate specification level, to allow simpler and more natural specifications. It is similar to the UML/OCL model level. The three levels are linked by syntactic maps and semantic maps. Syntactic maps work in a way similar to theory morphisms, and semantic maps have properties similar to the properties of reducts. This guarantees that the meta-level can be used to reason about the other levels.

In the next sections, we extend our previous static model, by introducing programs that update the current state and explicit time.

3 The Object Level

We model an *OO system* as a possibly distributed set of live *objects*, that dynamically evolve through a sequence of *states*, and collaborate to achieve a common goal. Each object has an *identity* and is created by a *template class*, which fixes the *signature* of its attributes and methods (programs) and the *implementation* of methods. We may have *query*, *action* and *activity* methods. Each object has one template (class)[1], and we write $o : T$ to indicate an object with identity o and template T.

We distinguish between *data* and *objects*. For simplicity, we assume that data are represented by the ground terms of a *predefined* data type $\mathcal{D}$. To represent *object identities*, $\mathcal{D}$ is always assumed to contain a closed framework $\mathcal{OBJ}$, axiomatising a special sort OBJ, generated by a denumerable set of constants. An object identity is a constant o of sort OBJ.

3.1 States

A state can be formalised as a first-order interpretation, or, under the hypothesis that each datum is represented by a unique *normal* ground term of $\mathcal{D}$, as a propositional interpretation. Here we consider the latter case, because it allows simpler definitions and is compatible with approaches based on propositional temporal logic. Moreover it is not too restrictive in the context of standard logic programs, where our propositional interpretations coincide with Herbrand models.

The attribute declarations of a template class T may be constant declarations $.c : [\,] \to s$, or relation declarations $.r : [s_1, \ldots, s_n]$, where $s, s_1, \ldots, s_n$ are sorts of $\mathcal{D}$. In the propositional approach *ground atoms* are considered as proposition symbols, and the attribute signature of an object $o : T$ is defined as follows.

Definition 1. (Object Attribute Signature). *Let $o : C$ be an object with class C. The* attribute signature *of $o : C$ is the propositional signature containing two kinds of symbols:* assignments*, of the form $o.c\,\mathrm{is}(t)$, and* attribute atoms*, of the form $o.p(t_1, \ldots, t_n)$, where $.c : [\,] \to s$ and $.r : [s_1, \ldots, s_n]$ are attribute declarations of C, and $t, t_1, \ldots, t_n$ are normal ground terms of $\mathcal{D}$.*

A population $\mathbf{P}$ is a set of objects, where each object has one template. The attribute signature of a population is the union of the signatures of its objects, and a state is characterised by a population $\mathbf{P}$ and by an interpretation of the signature of $\mathbf{P}$. We will indicate by $\mathbf{P}.Atb$ the signature of a population $\mathbf{P}$. The special syntax $o.c\,\mathrm{is}(t)$ is used only for the assignments. We will call $o.c$ a *constant* of $\mathbf{P}.Atb$.

Definition 2. (States). *A* state *of an OO system is a pair $S = (\mathbf{P}, \mathbf{I})$, where $\mathbf{P}$ is a population and $\mathbf{I} \subseteq \mathbf{P}.Atb$ is a $\mathbf{P}$-interpretation where each constant has one assignment.*

[1] Classes, interfaces, inheritance and polymorphism can be introduced, but we do not have space to consider them here.

The proposition symbols of an object o represent its local properties. Their current interpretation $\mathbf{I}$ represents the current state of o, as shown in Example 2, which also illustrates the correspondence of states with UML object diagrams.

Example 2. The classes of Fig. 3 introduce the following attribute declarations:

$$\begin{array}{lll} \textbf{Class } Car: & \text{NAVATB}: & .owner : [\,] \to \text{OBJ};\\ & \text{ATB}: & .km : [\,] \to Integer;\\ \textbf{Class } Person: & \text{NAVATB}: & .owns : [\text{OBJ}]; \end{array}$$

where navigation attributes NAVATB are a special kind of attributes, representing links to other objects.

The attribute signature of a car object c contains the assignments $c.km \,\text{is}(n)$ and $c.owner \,\text{is}(o)$, with $o : \text{OBJ}$ and $n : Integer$. The attribute signature of a person object p contains the atoms $p.owns(o)$, with $o : \text{OBJ}$. The *navigation attributes* are declared according to the multiplicities of the class diagram of Fig.3, $.owner$ is declared as a constant (by multiplicity 1 each car has one owner), and $.owns$ as a predicate (by multiplicity $*$ a person may own many cars)[2].

The object diagram of Fig. 3 corresponds to the state with population $\mathbf{P} = \{c1 : Car, c2 : Car, john : Person\}$ and interpretation $\mathsf{i} = \{c1.owner \text{ is } john, c1.km \text{ is } 2500, c2.owner \text{ is } john, c2.km \text{ is } 3000, john.owns(c1), john.owns(c2)\}$.

The navigation attributes $c1.owner$ is $john$ and $c2.owner$ is $john$ link $c1$ and $c2$ to $john$, and $john.owns(c1)$, $john.owns(c2)$ link $john$ to $c1, c2$.

The example introduces the *navigation attributes*, which link objects. Links are defined as follows.

Definition 3. (Links). *Let $o, o' : \text{OBJ}$ be two object identities. A* link *from o to o' is an assignment $o.n \,\text{is}(o')$ or an atom $o.n(\underline{\tau})$, where n is a navigation attribute and o' occurs in $\underline{\tau}$.*

A link from o to o' allows o to send messages to o', i.e., call methods of o'.

3.2 Queries

Since we are in computational logic, we consider relational queries, that is, a template T may contain query declarations of the form $.q : [s_1, \ldots, s_n]$, where $s_1, \ldots, s_n$ are sorts of $\mathcal{D}$.

For every object $o : T$, the query atoms of o are the ground terms of the form $o.q(\underline{\tau})$, and (in the propositional approach) query methods are propositional programs.

Definition 4. (Query Methods). *Let $o : T$ be an object and $.q$ be a query declared in T. A query method of o for $.q$ is a set of ground clauses of the form $o.q(\tau) \leftarrow \mathcal{B}$, where the body $\mathcal{B}$ may contain assignments, attribute atoms and query atoms.*

[2] In OCL, the type of $.owns$ would be $Set(Person)$.

An object may call queries or attributes of other objects. In this case, the clauses of the query method must be navigable. A navigation from o_1 to o_n is a sequence of navigation atoms $o_1.N_1, \ldots, o_k.N_k$ such that, for $1 \leq i < k$, $o_i.N_i$ is a link from o_i to o_{i+1}. Navigability is defined as follows.

Definition 5. (Navigable Objects and Clauses). *Let $\mathcal{B}$ be the body of a clause. An object o' is $\mathcal{B}$-navigable from o iff $o = o'$, or $\mathcal{B}$ contains a navigation sequence from o to o'; a clause $o.Q \leftarrow \mathcal{B}$ is navigable iff, for every atom $o'.A$ in $\mathcal{B}$, o' is $\mathcal{B}$-navigable from o.*

For each object $o : C$ and each query $.q$ declared in C, there is a set of clauses defining $.q$ in the context of o. To model object collaboration, we consider a population as a whole:

Definition 6. (Distributed Programs). *Let $S = (O, I)$ be a state. Let Q_S be the set of all clauses defining the queries in the contexts of the objects of O; we will call Q_S the* distributed program *of S.*

The semantics of Q_S can be defined by minimum I-models:

Definition 7. (Minimum I-models). *Let Q_S be the distributed program of a state $S = (O, I)$; the* minimum I-model *of Q_S is the minimum model of $Q_S \cup I$.*

We say that a ground query $o.q(\tau)$ is *true* (succeeds) in S, iff it is true in the minimum I-model of P_S. To indicate that $o.q$ is true in S, we will write $S \models o.q$, The truth of a set F of compound formulas is defined in the usual way and will be indicated by $S \models F$.

Example 3. In the class *Car* of Example 2 we declare the query $.sameOwner$: [OBJ] and implement it by the program

$$self.sameOwner(C) \leftarrow self.owner(P) \wedge P.owns(C).$$

In the state of Example 2, $.sameOwner$ is represented at the object level (see Section 5) by the distributed program

$$\begin{array}{l}
c1.sameOwner(c1) \leftarrow c1.owner \text{ is } john \wedge john.owns(c1). \\
c1.sameOwner(c2) \leftarrow c1.owner \text{ is } john \wedge john.owns(c2). \\
c2.sameOwner(c1) \leftarrow c2.owner \text{ is } john \wedge john.owns(c1). \\
c2.sameOwner(c2) \leftarrow c2.owner \text{ is } john \wedge john.owns(c2).
\end{array}$$

where the first two clauses define $.sameOwner$ in the context of $c1$ and the other two clauses define $.sameOwner$ in the context of $c2$.

3.3 Actions and Events

In a template T, an action declaration is of the form $.a : [s_1, \ldots, s_n] \rightarrow$ ACTS, where $s_1, \ldots, s_n$ are sorts of $\mathcal{D}$ and ACTS is a special sort introduced for the actions. The *actions* of an object o are all the ground terms $o.a(\tau)$, for every action declaration $o.a : \alpha \rightarrow$ ACTS of o.

Definition 8. (Action Methods). *Let $o : T$ be an object and $.a$ be an action declared by T. An* action method *of o for $.a$ is a set of* update-clauses *of the form $U \leftarrow \mathcal{E}$, where $\mathcal{E}$ is the enabling condition and U is of one of the forms $new(o.a, o_n : T)$, $del(o.a, o_d)$, $toT(o.a, o'.c\,\mathrm{is}(t))$, $toT(o.a, o'.p_t)$, $toF(o.a, o.p_f)$.*

The enabling condition $\mathcal{E}$ has the syntax of a body navigable from o, and states when $o.a$ may occur. It is the same in all the update-clauses of the same action. The update clauses define the updates caused by the action $o.a$.

When $o.a$ occurs, all the updates of $o.a$ occur simultaneously and give rise to a new state. We say that o_d is an object *deleted* by $o.a$, $o_n : T$ is an object *created* by $o.a$ with template T, $o'.c\,\mathrm{is}(t)$ is an assignment *forced* to true by $o.a$, $o'.p_t$ is an attribute atom *forced* to true by $o.a$, and $o.p_f$ an attribute atom *forced* to false by $o.a$. If an assignment $o'.c\,\mathrm{is}(t)$ is forced to true with a new value t, then the old assignment is *implicitly* forced to false.

A created object is added to the current population, while a deleted one is eliminated. To avoid inconsistency, an attribute atom cannot be forced to both true and false. If an atom or an assignment is not forced, its truth value is preserved. If $o'.p_t$ is forced to true by $o.a$, then o' must coincide with o or be one of the objects created by $o.a$, because we assume that $o.a$ acts *locally*, i.e., it may only initialise the proposition symbols of the new objects and update truth values of the proposition symbols of o. For the same reason, a proposition symbol forced to false must be an attribute of o. Finally, all the constants $o.c$ of a created object must be assigned.

We allow simultaneous actions. To handle them, we introduce *events*:

Definition 9. (Events). *An* event ϵ *is a set $\epsilon = \{o_1.a_1, \ldots, o_k.a_k\}$ of actions. ϵ is* admissible *in a state $S = (\mathbf{P}, \mathbf{I})$ iff it contains actions of the signature of $\mathbf{P}$ and $o_1, \ldots, o_k$ are different identities.*

We associate with each event ϵ the following sets:

- $\mathcal{E}_\epsilon$ is the conjunction of the enabling conditions of its actions;
- $\mathcal{NO}_\epsilon$ is the set of objects created by its actions;
- $\mathcal{DO}_\epsilon$ is the set of objects deleted by its actions;
- $\mathcal{T}_\epsilon$ is the set of proposition symbols forced to true by its actions;
- $\mathcal{F}_\epsilon$ is the set of all attribute atoms forced to false by its actions, all the assignments implicitly forced to false by its actions, and all the proposition symbols of the objects deleted by its actions.

Events *semantics* is formalised by state transitions, as follows.

We say that ϵ is *enabled* in a state $S = (\mathbf{P}, \mathbf{I})$ iff $S \models \mathcal{E}_\epsilon$ and the objects of $\mathcal{NO}_\epsilon$ are not objects of the current population $\mathbf{P}$, that is, they are new. Moreover, an object can be created by one action. This requirement, together with the locality condition, prevents the possibility of inconsistent updates. If ϵ is enabled in $S = (\mathbf{P}, \mathbf{I})$, then it may fire a transition $(\mathbf{P}, \mathbf{I})\ \epsilon \mapsto\ (\mathbf{P}', \mathbf{I}')$:

Definition 10. (OO State Transitions). *For every state $(\mathbf{P}, \mathbf{I})$ and every admissible event ϵ enabled in it, the firing of ϵ produces the state transition*

$$(\mathbf{P}, \mathbf{I})\ \epsilon \mapsto\ ((\mathbf{P} \cup \mathcal{NO}_\epsilon) \setminus \mathcal{DO}_\epsilon, (\mathbf{I} \cup \mathcal{T}_\epsilon) \setminus \mathcal{F}_\epsilon)$$

Example 4. The event $\epsilon = \{p1.sells(c1, p0), p0.buys(c1, p1), c1.sold(p1, p0)\}$, containing the actions specified below, represents a contract of purchase.

$$toF(p1.sells(c1, p0), p1.owns(c1)) \leftarrow p1.owns(c1).$$
$$toT(p0.buys(c1, p1), p0.owns(c1)) \leftarrow$$
$$toT(c1.sold(p1, p0), c1.owner \text{ is } p0) \leftarrow$$

Starting from the state with population $c1 : Car$, $p0, p1 : Person$ and interpretation $c1.owner$ is $p1$, $p1.owns(c1)$, the corresponding transition leads to the state with the same population and interpretation $c1.owner$ is $p0$, $p0.owns(c1)$.

3.4 Observation Sequences, Activities, Time and Space

We assume that we can ideally observe the system at successive observation times $t_1, \ldots, t_n, \ldots$, in such a way that no two consecutive actions $o.a_1$ and $o.a_2$ by the same object o occur in a single time interval t_i, t_{i+1}. Under this hypothesis an observation sequence can be modelled as a sequence of state transitions.

Definition 11. (Observation Sequences and Behaviours). *An* observation sequence *is a sequence* $S_0 \;\; \epsilon_0 \mapsto \;\; S_1 \;\; \epsilon_1 \mapsto \;\; S_2 \ldots$ *of state transitions such that, if it is finite, then no admissible event is enabled in its final state* S_n. *A* behaviour *is a class of observation sequences starting from the same initial state* S_0.

The purpose of a system specification is to define the system behaviour, by specifying *populations, queries, actions and activities.* Informally, an activity is a sequence of actions performed by a group of objects to do something. The enabling conditions may not be sufficient to impose the desired sequence, and a further level of control may be needed. Activities can be modelled as *constraints* on the behaviour. For example, we can model the activity *contract of purchase* considered in Example 4 by the synchronisation constraint imposing that the three actions have to occur simultaneously.

We will come back to the specification issue in Section 5. We conclude this section by considering space and time in an OO system. Time is related to updates, while space is related to the "geometry" of the communication among objects.

Navigability represents one of the space "dimensions". We can use the classical *visibility constraints* like "public" and "private", to constrain navigability.

A second space dimension is the *observes* relation: an object o observes an attribute $o'.A$ of an object o' if o is interested in the updates of $o'.A$. The main reason for observing $o'.A$ is its use in the body of a query or in the enabling condition of an action. However there may be many reasons for observing an attribute, as highlighted for example by the *observer-observable* pattern [9].

To specify dynamic aspects and to reason about them, we introduce time. In an observation sequence $S_0 \;\; \epsilon_0 \mapsto \;\; S_1 \;\; \epsilon_1 \mapsto \;\; S_2 \ldots$, each state S_i $(i \geq 0)$ has time i. We call i the *global time*, since observation sequences represent a global view of the system.

Besides global time, it is useful to consider *local time*, corresponding to the local view of an object or a subsystem. For example we can use $\forall c : Car\,\boldsymbol{.}\; c@0.km = 0$ to indicate that the odometer of a car is 0 at local time 0, i.e., when it is created, independently from the global time. Local time is also useful when considering single objects or, by a hierarchical organisation of locality, *modules*.

In this paper we do not consider modules, and we introduce local time only for objects, by means of *timed objects*. A timed object with identity o and time t will be indicated by $o@t$. The time of an object *increases* whenever it observes that something is *changed*.

A *timed population* is a set $\mathbf{T}$ of timed objects that does not contain pairs $o@t$, $o@t'$ with the same object and different times. That is, each object has one time. The signature of a timed population $\mathbf{T}$ is that of its objects, where o is an object of $\mathbf{T}$ iff there is t such that $o@t$ belongs to $\mathbf{T}$.

A *timed state* is a pair $(\mathbf{T}, \mathbf{I})$, where $\mathbf{T}$ is a timed population and $\mathbf{I}$ is an interpretation of its propositional symbols. Each timed state $(\mathbf{T}, \mathbf{I})$ has an associated (non-timed) OO state $(\mathbf{P}_\mathbf{T}, \mathbf{I})$, where $\mathbf{P}_\mathbf{T}$ is obtained from $\mathbf{T}$ by forgetting time.

An object o *observes* an event ϵ if ϵ modifies an attribute observed by o or the population observed by o. Timed transitions trace the observability relation:

Definition 12. (Timed Transitions). *Let* $(\mathbf{T}, \mathbf{I})$ *and* $(\mathbf{T}', \mathbf{I}')$ *be two timed states.* $(\mathbf{T}, \mathbf{I})\ \epsilon \mapsto\ (\mathbf{T}', \mathbf{I}')$ *is a* timed transition *iff* $(\mathbf{P}_\mathbf{T}, \mathbf{I})\ \epsilon \mapsto\ (\mathbf{P}_{\mathbf{T}'}, \mathbf{I}')$ *is a state transition and for every* $o@t \in \mathbf{T}'$ *one of the following cases holds:* $o@t \in \mathbf{T}$ *and* o *does not observe* ϵ*, or* $o@t-1 \in \mathbf{T}$ *and* o *observes* ϵ*, or* $t = 0$ *and* $o \in \mathcal{NO}_\epsilon$*.*

A *timed observation sequence* is a sequence of timed transitions, and a *timed behaviour* is a set of timed observation sequences.

In timed observation sequences, the local time of objects implicitly traces the observability relation and is useful for devising independent subsystems, namely subsystems that can be characterised independently from their environment. Independence becomes a central notion when considering modularity aspects. We can also use a local time to trace different independent activities separately.

4 The Meta-level and the Meta-framework

At the meta-level, an OO system $\mathcal{S}$ is represented by a reusable unit $\mathcal{U}(\mathcal{S})$, that we call the *meta-representation* of $\mathcal{S}$. $\mathcal{U}(\mathcal{S})$ is an extension of the *general meta-unit* $\mathcal{U}$ outlined in Fig. 4. $\mathcal{U}$ contains a specification framework $\mathcal{M}$, that we call the *meta-framework*, and an open steadfast program MP, that we call the *meta-program*. The meta-representation $\mathcal{U}(\mathcal{S})$ is obtained by adding suitable *representation axioms* and *clauses*, as we will outline in the next section. Here, we explain the general features of the meta-framework $\mathcal{M}$.

$\mathcal{M}$ axiomatises the general assumptions on OO systems discussed in the previous section, and is the basic component of the general meta-theory in Fig. 1. It contains $\mathcal{OBJ}$, to represent object identities, and $\mathcal{SET}(\text{ACT})$, to axiomatise the defined sort EV of events by finite sets of actions; the subframework $\mathcal{NAT}$ of $\mathcal{SET}(\text{ACT})$ is also used to represent time by natural numbers. The intended

unit $\mathcal{U}$;

MetaFramework $\mathcal{M}$; $\textsc{import}\ \mathcal{OBJ}, \mathcal{SET}(\textsc{act})[set(\textsc{act}) \mapsto \textsc{ev}]$;

$\textsc{osorts}$: $\textsc{atbs}, \textsc{acts}, \textsc{qrys}, \textsc{tpls}$;

$\textsc{dsorts}$: $\textsc{obj}\mathrm{T}, \textsc{atb}, \textsc{qry}, \textsc{act}$;

$$
\begin{array}{lll}
\textsc{odecl}: & dcl : [\textsc{atb}, \textsc{tpls}] | [\textsc{qry}, \textsc{tpls}] | [\textsc{act}, \textsc{tpls}]; & \\
 & ini : [\textsc{obj}, \textsc{tpls}]; & iniT : [\textsc{atb}]; \\
 & t : [\textsc{nat}, \textsc{qry}]; & toT, toF : [\textsc{nat}, \textsc{act}, \textsc{atb}]; \\
 & new : [\textsc{nat}, \textsc{act}, \textsc{obj}, \textsc{tpls}]; & del, obs : [\textsc{nat}, \textsc{act}, \textsc{obj}]; \\
 & enb : [\textsc{nat}, \textsc{act}]; & occ : [\textsc{nat}, \textsc{ev}];
\end{array}
$$

$$
\begin{array}{ll}
\textsc{ddecl}: & . : [\textsc{obj}, \textsc{atbs}] \to \textsc{atb} | [\textsc{obj}, \textsc{qrys}] \to \textsc{qry} | [\textsc{obj}, \textsc{acts}] \to \textsc{act}; \\
 & @ : [\textsc{obj}, \textsc{nat}] \to \textsc{obj}\mathrm{T}; \quad live : [\textsc{nat}, \textsc{obj}\mathrm{T}, \textsc{tpls}]; \quad t : [\textsc{nat}, \textsc{atb}];
\end{array}
$$

$\textsc{constr}$: for $o, o' : \textsc{obj}, n : \textsc{nat}, a, b : \textsc{acts}, p : \textsc{atbs}, e, e' : \textsc{ev}, A, B : \textsc{act}$:

$$
\begin{array}{l}
occ(n,e) \wedge occ(n,e') \to e = e'; \\
occ(s(n),e) \wedge o.a \in e \to enb(n, o.a) \wedge \exists t \,.\; live(n, o@t); \\
occ(n,e) \wedge o.a \in e \wedge o'.b \in e \to \neg o = o';
\end{array}
$$

$\textsc{ax}1$: $\mathrm{FreeAx}(@ : \textsc{obj}\mathrm{T}, . : \textsc{atb}, . : \textsc{qry}, . : \textsc{act})$;

$\textsc{def}$: for $o : \textsc{obj}; n : \textsc{nat}, p : \textsc{atbs}, a : \textsc{acts}, tp : \textsc{tpls}$:

$$
\begin{array}{ll}
toNew(n,o,tp) & \leftrightarrow \exists e : \textsc{ev} \,.\; occ(s(n),e) \wedge \exists a \in e \,.\; new(n,a,o,tp); \\
toT(n, o.p) & \leftrightarrow \exists e : \textsc{ev} \,.\; occ(s(n),e) \wedge \exists a \in e \,.\; toT(n,a,o.p); \\
notToF(n, o.p) & \leftrightarrow \exists e : \textsc{ev} \,.\; occ(s(n),e) \wedge \forall a \in e \,.\; \neg toF(n,a,o.p) \wedge \neg del(n,a,o); \\
obsEv(n,o) & \leftrightarrow \exists e : \textsc{ev} \,.\; occ(s(n),e) \wedge (\exists a \in e \,.\; obs(n,a,o)) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall a \in e \,.\; \neg del(n,a,o); \\
notObsEv(n,o) & \leftrightarrow \exists e : \textsc{ev} \,.\; occ(s(n),e) \wedge \forall a \in e \,.\; \neg del(n,a,o) \wedge \neg obs(n,a,o);
\end{array}
$$

$\textsc{ax}2$: for $o : \textsc{obj}; n, t : \textsc{nat}, tp : \textsc{tpls}$:

$$
\begin{array}{ll}
live(0, o@t : tp) & \leftrightarrow ini(o,tp) \wedge t = 0; \\
live(s(n), o@0 : tp) & \leftrightarrow toNew(n,o,tp) \ \vee\ notObsEv(n,o) \wedge live(n, o@0 : tp); \\
live(s(n), o@s(t) : tp) & \leftrightarrow obsEv(n,o) \wedge live(n, o@t : tp) \vee \\
 & \qquad notObsEv(n,o) \wedge live(n, o@s(t) : tp);
\end{array}
$$

$\textsc{ax}3$: for $n : \textsc{nat}, p : \textsc{atb}$:

$$
\begin{array}{ll}
t(0,p) & \leftrightarrow iniT(p); \\
t(s(n),p) & \leftrightarrow toT(n,p) \vee notToF(n,p) \wedge t(n,p);
\end{array}
$$

$\textsc{repr}$:

Representation axioms and specifications of $\mathcal{S}$;

$$
\begin{array}{ll}
\textsc{prog}\ \mathrm{MP}: & live(0, O@0, Templ) \leftarrow ini(O, Templ). \\
 & live(s(N), O@0, Templ) \leftarrow toNew(N, O, Templ). \\
 & live(s(N), O@0, Templ) \leftarrow notObsEv(s(N), O) \wedge live(N, O@0, Templ). \\
 & live(s(N), O@s(T), Templ) \leftarrow obsEv(s(N), O) \wedge live(N, O@T, Templ). \\
 & live(s(N), O@s(T), Templ) \leftarrow notObsEv(s(N), O) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad live(N, O@s(T), Templ). \\
 & t(0,P) \leftarrow iniT(P). \\
 & t(s(N),P) \leftarrow toT(N,P). \\
 & t(s(N),P) \leftarrow notToF(N,P) \wedge t(N,P). \\
 & \ldots \\
 & \text{representation clauses}
\end{array}
$$

Fig. 4. General meta unit.

models of $\mathcal{M}$ will be called *meta-models*. The open symbols allow us to *represent* OO systems and OO system specifications, by the representation axioms. We distinguish *population* meta-symbols and the *behavioural* meta-symbols.

The *population meta-symbols* are TPLS, ATBS, ACTS, QRYS (meta-sorts for template identifiers and attribute, action and query signatures) and *dcl* ($dcl(X, T)$ means that X is declared by the template T). They allow us to represent the object-level populations of an OO system $\mathcal{S}$ and their signatures, by adding the suitable representation axioms to $\mathcal{M}$. Since populations depend on signatures, the extension $\mathcal{M}(\Sigma_{\mathcal{S}})$ obtained in this way will be called the *signature representation* of $\mathcal{S}$. We will assume that it satisfies the following regularity property:

Definition 13. (Regularity). *Let $\mathcal{M}(\Sigma_{\mathcal{S}})$ be the signature representation of a system $\mathcal{S}$, i be a meta-model of $\mathcal{M}(\Sigma_{\mathcal{S}})$, and n be a natural number. Let $\mathbf{P}_n$ be the population containing the objects $o : tp$ such that $i \models \exists t \,\bullet\, live(n, o@t : tp)$, and $\mathbf{I}_n$ be the set containing the ground terms $o.p$ such that $i \models t(n, o.p)$. We say that $\mathcal{M}(\Sigma_{\mathcal{S}})$ is* regular *if, for every n, $(\mathbf{P}_n, \mathbf{I}_n)$ is an OO state of $\mathcal{S}$.*

The *behavioural meta-symbols* are introduced to represent queries, actions, events and observation sequences, and are interpreted within a regular signature representation $\mathcal{M}(\Sigma_{\mathcal{S}})$. The intended correspondence with the object-level observation sequences, i.e., the object-level semantics, is given by the following definition.

Definition 14. (Correspondence). *Let $\mathcal{M}(\Sigma_{\mathcal{S}})$ be a regular representation of the population signatures of an OO system $\mathcal{S}$. A pre-interpretation* j *of $\mathcal{M}(\Sigma_{\mathcal{S}})$* corresponds *to a timed observation sequence $S_0 \;\; e_0 \mapsto \;\; S_1 \;\; e_1 \mapsto \;\; \ldots$ of $\mathcal{S}$ iff:*

1. *j $\models toT(n, a, P)$ iff P is forced to true by the action a in S_n;*
2. *j $\models toF(n, a, P)$ iff P is forced to false by the action a in S_n;*
3. *j $\models new(n, a, o, tp)$ iff $o : tp$ is created by the action a in S_n;*
4. *j $\models del(n, a, o)$ iff o is deleted by the action a in S_n;*
5. *j $\models ini(o, tp)$ iff o is an object of $S_0 = (\mathbf{T}_0, \mathbf{I}_0)$, that is, $o@0 : tp \in \mathbf{T}_0$;*
6. *j $\models iniT(P)$ iff P is true in $S_0 = (\mathbf{T}_0, \mathbf{I}_0)$, that is, $P \in \mathbf{I}_0$;*
7. *j $\models enb(n, a)$ iff the enabling condition of a is true in S_n;*
8. *j $\models t(n, Q)$ iff the query Q is true in S_n;*
9. *j $\models obs(n, a, o)$ iff the updates caused by $a \in \epsilon_n$ are observed by o in S_n;*
10. *j $\models occ(s(n), e)$ iff the event e has just occurred at time $s(n)$, i.e., $e = e_n$.*

The constraints CONSTR correspond to the general assumptions on signatures, actions and events discussed in the previous section. For conciseness, we have indicated only the constraints on events. The constraints guarantee that for a regular signature representation $\mathcal{M}(\Sigma_{\mathcal{S}})$ and a pre-interpretation j that corresponds to an observation sequence Ω, the intended meaning of the defined symbols is axiomatised by the definition axioms as follows.

– AX1 state that the defined sort OBJT is freely generated by @, and ATB, QRY, ACT by ".". A meta-term $o@t$ indicates an object o with local time t. The dot operator $x.\sigma(\ldots)$ represents at the meta-level (by the extra-argument approach) the object-level renaming.

- DEF introduce useful abbreviations by explicit definitions.
- AX2 axiomatise the meta-predicate $live(n, o@t, tp)$, with the following intended meaning: at global time n, o is a live object with local time t and template tp. We use the more intuitive syntax $live(n, o@t : tp)$.
- AX3 axiomatise the meta-predicate $t(n, p)$, with the following intended meaning: at global time n, the attribute p is true.

In particular, the defined symbols $live(n, o@t : tp)$ and $t(n, p)$ represent the timed states by the *projection*

$$\mathsf{i} \downarrow n = (\mathbf{T}_n, \mathbf{I}_n)$$

that maps each meta-model i and time n into the timed OO state $(\mathbf{T}_n, \mathbf{I}_n)$ such that $o@t : tp \in \mathbf{T}_n$ iff $\mathsf{i} \models live(n, o@t : tp)$ and $o.p \in \mathbf{I}_n$ iff $\mathsf{i} \models t(n, o.p)$. Thus, once we are able to represent an observation sequence Ω by a corresponding pre-interpretation j, the definition axioms allow us to reconstruct the states of Ω and to reason about them at the meta level. This is what is stated by the next theorem and what will be used in Section 5 to represent object-level properties at the meta level.

Theorem 1. *Let $\mathcal{M}(\Sigma_{\mathcal{S}})$ be a regular representation of the population signatures of an OO system $\mathcal{S}$. Let j be a pre-interpretation of $\mathcal{M}(\Sigma_{\mathcal{S}})$ that corresponds to an observation sequence Ω of $\mathcal{S}$ and $\mathsf{j}^{\mathcal{M}(\Sigma)}$ be the corresponding j-isoinitial model. Then the sequence $\mathsf{j}^{\mathcal{M}(\Sigma)} \downarrow 0 \ \ e_0 \mapsto \ \mathsf{j}^{\mathcal{M}(\Sigma)} \downarrow 1 \ \ e_1 \mapsto \ \ldots$ such that $\mathsf{j} \models occ(j, e_j)$ (for $j = 0, 1, \ldots$) is a timed observation sequence that coincides with Ω.*

Proof. Since j corresponds to Ω, transitions $\ e_j \mapsto \ $ are already defined in the right way. We have to show that the axioms of $\mathcal{M}$ define $live(n, o@t : tp)$ and $t(n, o.p)$ (for an attribute symbol p) in such a way that they represent Ω by the projections $\downarrow n$. The sketch of the proof is the following. Each $\mathsf{j}^{\mathcal{M}(\Sigma)} \downarrow n$ is a timed OO state. By induction on n, we can prove that it is the state with global time n of Ω. The basis of the induction follows from the fact that ini and $iniT$ are interpreted according to Definition 14. The induction step follows from the fact that the open symbols toT, toF, ... are interpreted according to Definition 14 and from the constraints and the axioms of $\mathcal{M}(\Sigma)$.

5 Representing OO System Specifications

The object and meta levels can be used for representing OO system specifications and for reasoning about them. For lack of space, we will only outline the general approach and illustrate it by some representative examples. The general picture is illustrated in Fig. 5. $\mathcal{S}$ is the *system specification*, $\mathcal{O}(\mathcal{S})$ is its *object-level representation*, and $\mathcal{M}(\mathcal{S})$ its *meta-level representation*. $\mathcal{O}(\mathcal{S})$ defines the operational semantics of $\mathcal{S}$ as a *behaviour* (Definition 11). $\mathcal{M}(\mathcal{S})$ is an adequate extension of $\mathcal{M}$. It has a model-theoretic (thus declarative) semantics, and observation

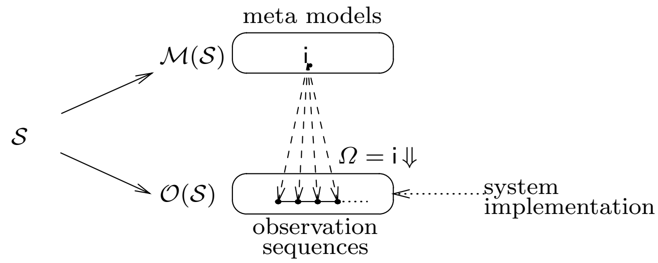

Fig. 5.

sequences are represented by meta-models. The dashed arrows indicate the $\downarrow n$-projections mapping a meta-model i into an observation sequence $\Omega = \mathsf{i}\Downarrow$, the solid arrows indicate syntactic maps, and the dotted one an abstraction process, where observation sequences represent ideal observations of the running system at a given granularity.

We distinguish behavioural and correctness specifications.

Behavioural specifications define the system behaviour. The object-level and the meta-level representations have to satisfy the conditions:

(a) $\mathcal{M}(\mathcal{S})$ includes a regular signature representation $\mathcal{M}(\Sigma_{\mathcal{S}})$ of $\mathcal{S}$.
(b) Let i be a finitary meta-model of $\mathcal{M}(\mathcal{S})$; then $\mathsf{i}\Downarrow =_{def} \mathsf{i}\downarrow 0 \;\; e_0 \mapsto \;\; \mathsf{i}\downarrow 1 \;\; e_1 \mapsto \ldots$, where $\mathsf{i} \models occ(i, e_i)$, is an observation sequence of $\mathcal{O}(\mathcal{S})$.
(c) For every observation sequence Ω of $\mathcal{O}(\mathcal{S})$, there is at least one meta-model i of $\mathcal{M}(\mathcal{S})$ such that $\mathsf{i}\Downarrow = \Omega$.

Correctness specifications state properties to be satisfied by the system behaviour, that is, a correctness specification S_P defines a predicate $P : \mathcal{O}(\mathcal{S}) \to Boolean$. We say that $\mathcal{S}$ *is correct with respect to* S_P iff $P(\Omega) = true$, for every $\Omega \in \mathcal{O}(\mathcal{S})$. A correctness specification S_P can be studied by representing it by a formula $S_P \uparrow$ of $\mathcal{M}(\mathcal{S})$. We will call $S_P \uparrow$ the meta-level representation of S_P. We require that:

(d) for every meta-model i, if $\mathsf{i} \models S_P \uparrow$, then $P(\mathsf{i}\Downarrow) = true$.

If (a), (b), (c), (d) are satisfied and $S_P \uparrow$ is a theorem of $\mathcal{M}(\mathcal{S})$, then $\mathcal{S}$ is correct with respect to S_P. (a) and (d) must be ensured by the representation of signatures and specifications. Correctness can be treated in $\mathcal{M}(\mathcal{S})$ by *proving* $S_P \uparrow$ and the *steadfastness* of the meta-program. Indeed, (c) and (d) are ensured by the steadfastness of the extension of MP by the representation clauses. Furthermore, MP gives rise to a *reflective environment*, where we can perform meta-level computations.

We conclude this section with some examples that illustrate the approach outlined above.

Example 5. **Representing Signatures, Queries and Actions**. In this example we consider languages where the object-level representation of a specification $\mathcal{S}$ can be formalised as a quadruple $\mathcal{O}(\mathcal{S}) = \langle \mathcal{D}, \mathcal{P}, \mathcal{Q}, \mathcal{A} \rangle$, where: $\mathcal{D}$ is the data type of $\mathcal{S}$, $\mathcal{P}$ is the set of its populations, and $\mathcal{Q}$ and $\mathcal{A}$ are semantic maps that associate every population $\mathbf{P}$ with the queries $\mathcal{Q}(\mathbf{P})$ and the actions $\mathcal{A}(\mathbf{P})$ of $\mathbf{P}$. The specified behaviour $\mathcal{O}(\mathcal{S})$ contains the observation sequences $(\mathbf{P}_1, \mathbf{I}_1)\epsilon_1 \mapsto (\mathbf{P}_2, \mathbf{I}_2)\epsilon_2 \mapsto \ldots$ such that $\mathbf{P}_i$ are populations of $\mathcal{P}$ and the actions and queries of $\mathbf{P}_i$ are given by the maps $\mathcal{A}$ and $\mathcal{Q}$. We will call this behaviour the *free behaviour of* $\mathcal{S}$.

We show a possible Prolog-like specification language of this kind, by means of the following example, where queries and actions are added to the template classes *Car* and *Person* of Example 2.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{Class } \mathit{Car} :} \\
\text{NAVATB} : & .owner : [\,] \to \text{OBJ}; \\
\text{ATB} : & .km : [\,] \to Integer; \\
\text{QRY} : & \mathit{self.sameOwner}(c) \leftarrow \mathit{self.owner}\,\mathrm{is}(p) \wedge \mathit{Person}(p) \wedge p.owns(c); \\
\text{ACT} : & \mathit{toT}(\mathit{self.sold}(p_1, p_2), \mathit{self.owner}\,\mathrm{is}(p_2)) \leftarrow \\
\multicolumn{2}{l}{\textbf{Class } \mathit{Person} :} \\
\text{NAVATB} : & .owns : [\text{OBJ}]; \\
\text{ACT} : & \mathit{toT}(\mathit{self.buys}(c, p), \mathit{self.owns}(c)) \leftarrow; \\
 & \mathit{toF}(\mathit{self.sells}(c, p), \mathit{self.owns}(c)) \leftarrow \mathit{self.owns}(c);
\end{array}
$$

A *query clause* is defined in a template T and its atoms may be of the form $\mathit{self}.A$, where A is a query or an attribute of T, or of the form $d.A$, where the *message destination* d (p in the Example) is different from *self*, must be bounded by a *template predicate* $T'(d)$ ($\mathit{Person}(p)$ in the Example) and A must be declared in T'. An action clause has a similar syntax, but now the head is an update atom and the enabling condition must be the same in all the clauses defining the same action.

Now we build the object-level representation $\mathcal{O}(\mathcal{S})$. The data type and populations are defined as in Example 2. We associate each population $\mathbf{P}$ with the actions $\mathcal{A}(\mathbf{P})$ and the queries $\mathcal{Q}(\mathbf{P})$ of $\mathbf{P}$, as follows.

$\mathcal{Q}(\mathbf{P})$ is obtained by grounding each query clause Q by replacing *self* by the identities o such that $o : T \in \mathbf{P}$, and the (possible) message destinations d by the identities o' such that $o' : T' \in \mathbf{P}$. $\mathcal{A}(\mathbf{P})$ is obtained by the same grounding process. For instance, the query given in Example 3 is the object level semantics of the query clause for *sameOwner* in the population there considered, and the actions in Example 4 are the object-level semantics of the above action clauses.

To build the meta-level representation $\mathcal{M}(\mathcal{S})$, firstly we have to extend $\mathcal{M}$ into a regular signature representation $\mathcal{M}(\Sigma_{\mathcal{S}})$. For this, we include the data type $\mathcal{INTEGER}$ and we close the population signature by the declarations and axioms[3]:

$$
\begin{array}{l}
\mathit{Car}, \mathit{Person} : [\,] \to \text{TPLS} \;\; \mathit{km}\,\mathrm{is} : [\mathit{Integer}] \to \text{ATBS} \\
\mathit{owner}\,\mathrm{is} : [\mathit{Obj}] \to \text{ATBS} \;\; \mathit{owns} : [\mathit{Obj}] \to \text{ATBS}
\end{array}
$$

[3] The syntax $\mathit{km}\,\mathrm{is}(n)$ has not been considered in Section 4 for conciseness. It can be treated by introducing in $\mathcal{M}$ suitable meta-symbols and constraints.

$$\begin{array}{l}\mathrm{FreeAx}(\mathit{Car}, \mathit{Person} : \textsc{tpls}, \mathit{km}\,\mathrm{is}, \mathit{owner}\,\mathrm{is}, \mathit{owns} : \textsc{atbs}, \mathit{nil} : \textsc{qrys}, \mathit{nil} : \mathit{acts})\\ \mathit{dcl}(a, \mathit{Car}) \leftrightarrow (\exists n .\; a = \mathit{km}\,\mathrm{is}(n)) \vee (\exists p .\; a = \mathit{owner}\,\mathrm{is}(p))\\ \mathit{dcl}(a, \mathit{Person}) \leftrightarrow \exists c .\; a = \mathit{owns}(c))\end{array}$$

We can show that this way of closing the population signature is general and gives rise to regular representations, considering only templates. A suitable meta-theory can be introduced in $\mathcal{M}$, to treat classes and polymorphism.

To complete the meta-level representation $\mathcal{M}(\mathcal{S})$, we add the representation axioms of specifications of queries and actions, according to Example 7.

Finally, to complete the meta-program, we translate the query and action clauses of the specification into meta-clauses defining the open meta-predicates t, toT, toF, ... The translation introduces the global time n, replaces *self* by a fresh variable x, and x is bounded in the body by $live(n, x@t : T)$, where T is the template of *self*. Similarly, the template predicate $T'(d)$ for a destination d is translated by $live(n, d@t' : T')$. The resulting meta-clauses are added to the meta-program MP. For lack of space we omit the details. To give an example, the meta-clauses representing the query and action clauses considered before are:

$$\begin{aligned} t(n, x.sameOwner(c)) &\leftarrow live(n, x@t : Car) \wedge t(n, x.owner\,\mathrm{is}(p))\\ &\qquad \wedge live(n, p@t : Person) \wedge t(n.p.owns(c));\\ toT(n, x.sold(p_1, p_2), x.owner\,\mathrm{is}(p_2)) &\leftarrow live(n, x@t : Car);\\ toT(n, x.buys(c, p), self.owns(c)) &\leftarrow live(n, p@t : Person);\\ toF(n, x.sells(c, p), self.owns(c)) &\leftarrow live(n, p@t : Person) \wedge t(n, x.owns(c)). \end{aligned}$$

Example 6. **Activities.** An activity specification is a *behavioural specification.* It is expressed as a constraint on the free behaviour of a system specification. Here we show the example of a possible syntax for expressing synchronisation constraints. The example shows a collaboration of three objects implementing a contract of purchase.

$$\begin{array}{l}\textbf{Collaboration}\; p_1, p_2 : Person; c : Car :\\ \textsc{activity}\;\; contractOfPurchase :\\ sync(c.sold(p_1, p_2), [p_1.sells(c, p_2), p_2.buys(c, p_1)]) \leftarrow;\\ sync(p_1.sells(c, p_2), [c.sold(p_1, p_2), p_2.buys(c, p_1)]) \leftarrow;\\ sync(p_2.buys(c, p_1), [c.sold(p_1, p_2), p_1.sells(c, p_2)]) \leftarrow;\end{array}$$

The statement $sync(a, l)$ means that action A synchronises the actions in l. The object-level representation of synchronisation clauses is given in the way illustrated for queries and actions. Their semantics is the constraint imposing that an event containing an action A synchronising B must also contain B. This constraint is expressed at the meta level by the meta-axiom:

$$occ(e, n) \wedge a \in e \wedge sync(n, a, s) \wedge b \in s \rightarrow b \in e$$

and we have to add the meta-level representation of the synchronisation clauses.

Example 7. **Correctness Specifications and Invariants.** There are various kinds of correctness specifications. To deal with open correctness (steadfastness) queries and actions must have specifications. Actions are specified by a combined use of pre/post-conditions and invariants. Query specifications can be treated as in [14].

Each kind of specification has a specific meta-level representation. We consider only invariants, for lack of space. Invariants represent properties that must hold in every state, or at least in chosen states, determined for example by the local time or by spy points. An example is the invariant

$$\begin{array}{l}\textbf{Context } \mathit{Car} \\ \textsc{inv } \mathit{zeroKm}: \ \mathit{self}@0.\mathit{km} = 0\end{array}$$

Like in OCL [22], a constraint is given in the context of a template in a specification. Here the informal correctness property P (see condition (d)) expressed by the invariant is: "for every state S and object $c : \mathit{Car}$, $c.\mathit{km}\,\mathrm{is}(0)$ holds in S if the local time of c is 0". Instead of giving a separate object-level semantics, we can give directly a meta-level representation $S\uparrow$ and assume condition (d) as defining the object-level semantics. In our example $\mathit{zeroKm}\uparrow$ is

$$\forall n, c \,.\; \mathit{live}(n, c@0 : \mathit{Car}) \rightarrow t(n, c.\mathit{km}\,\mathrm{is}(0))$$

We can see that the above formula is true in a meta-model i iff, in the observation sequence $\Omega = \mathsf{i}\Downarrow$, $c.\mathit{km}\,\mathrm{is}(0)$ is true in every state S_n with a timed population containing $q@0 : \mathit{Car}$. This formalises the informal predicate P.

Example 8. **General-purpose Specifications**. In general, a separate specification formalism is useful to express specifications in a more compact and natural way, and it is convenient to choose different formalisms for different purposes. There are cases where it is convenient to directly use the meta level as a specification formalism. This choice may be adequate, for instance, to express constraints that are not domain specific. For example, fairness is expressed by the constraint

$$\forall n : \textsc{nat}, a : \textsc{act} \,.\; \mathit{enb}(n, a) \rightarrow \exists m \,.\; m > n \wedge \mathit{occ}(m, a)$$

This constraint does not depend on the application domain, but it may be seen as a constraint for the implementations of the programming language. If we assume it, then we require that implementations apply a fair scheduling.

6 Conclusion

We have introduced a three-level formalisation of OO systems, with an object level, a meta level and many possible specification levels, linked by the maps illustrated in Fig. 5. In this context, the specification and meta levels can both be used to write down specifications, while the object-level semantics relates specifications to the corresponding system configurations.

Our approach is very similar, in spirit, to the multi-level approach of UML [19]. In UML, the official specification language is OCL [22]. Various efforts have been made to give a formal semantics to OCL and some Java implementations have been developed [18]. Although we start from a different background, one that is closer to formalising UML by ADTs [3, 8], our multi-level approach has many similarities to the approaches developed in the UML and OCL community. In particular, we have in common the static diagrams, namely class and object diagrams, including invariants, and our behavioural specifications are similar, in a broad sense, to the UML behavioural diagrams. However, our approach is logically based, and is in this regard close to approaches for formalising UML and OCL. A first formalisation of OCL in Isabelle HOL has been proposed in [4]. The approach of [5] is based on the specification language Z. There is a working group on precise UML [21].

Concerning the dynamic aspects, at the object level our model is similar to the approaches based on temporal logic [8]. Indeed, behaviours can be considered as a particular kind of Kripke Transition Systems [16], where each Kripke world has an associated population. Populations allow us to model open systems, in a way similar to [1]. Instead of adding temporal operators to the object level, we have preferred to model time at the meta-level, because this allows us to extend our previous work on open specification frameworks and steadfast open programs [15]. Moreover, temporal logic operators could be introduced at the specification level. That is, our approach is not alternative, but complementary to temporal logic approaches.

In this paper, the main effort has been to extend our previous three-level formalisation of static OO systems by introducing time. The proposed formalisation is the basis for long-term work, involving many related aspects:

(a) *Classes, types and polymorphism.* With a slight extension of the meta-language and of the related constraints, we can define classes and types, together with inheritance and overriding polymorphism.
(b) *Implementing the object and the meta-levels.* It is possible to build a Prolog interpreter for the object level and the meta-level, by introducing a syntax for representing classes, attributes and methods. We are looking for a tool that is useful for understanding specifications (e.g., by animating them), rather than a tool for implementing OO systems.
(c) *Specification languages.* In Section 5, we have shown some examples of specification formalisms, at an intermediate level between the meta- and the object level. The idea is that different aspects are treated in a more natural way by different formalisms (in this, we are in line with the general ideas underlying the UML approach), and that the meta-framework is the unifying framework.
(d) *Correctness.* Correctness can be defined in terms of steadfastness at the meta level, and can be proved using the results of [15]. We have not yet studied specific proof strategies, but we can reasonably expect that different specification languages will underly different proof methods.
(e) *Modules.* We have only some ideas, to be developed further. Essentially, the idea is to formalise modules by enriching the structure of local time.

Our next steps will be (a) and (b), but really all the previous points have to be developed if we want to reach our final goal, which is to model OOD frameworks

[12] by open reusable units. Clearly this paper represents just a step in that direction.

References

1. R. Alur, T.A. Henzinger, O. Kupferman. Alternating-time Temporal Logic. *JACM*, 49(5):672–713, 2002.
2. C. Atkinson *et al. Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.
3. R.H. Bourdeau, B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.
4. A. Brucker, B. Wolf. A proposal for a formal OCL semantics in Isabelle/HOL. *LNCS 2410*, pages 99–114, Springer-Verlag, 2002.
5. T. Clark, A. Evans. Foundations of the Unified Modeling Language. *2nd BCS-FACS Northern Formal Methods Workshop*. http://ewic.bcs.org/conferences/1997/papers/paper6.htm
6. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net
7. D.F. D'Souza, A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
8. H.-D. Ehrich. Object specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 12, pages 435–465. Springer, 1999.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.
10. G. Larsen. Designing component-based frameworks using patterns in the UML. *Comms. ACM*, 42(10):38–45, October 1999.
11. K.-K. Lau, M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. *LNCS 883*, pages 104–121. Springer-Verlag, 1994.
12. K.-K. Lau, M. Ornaghi. OOD frameworks in component-based software development in computational logic. *LNCS 1559*, pages 101–123. Springer-Verlag, 1999.
13. K.-K. Lau, M. Ornaghi. Isoinitial semantics for logic programs. *LNAI 1861*, pages 223–238. Springer-Verlag, 2000.
14. K.-K. Lau, M. Ornaghi. Correct object-oriented systems in computational logic. *LNCS 2372*, pages 168–190. Springer-Verlag, 2002.
15. K.-K. Lau, M. Ornaghi, S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
16. M. Müller-Olm, D. A. Schmidt, B. Steffen. Model-checking: A tutorial introduction. *LNCS 1694*, pages 330–354. Springer, 1999.
17. Response to the UML 2.0 OCL RfP (ad/2000-09-03). em OMG Document ad/2003-01-07. http://www.omg.org/docs/ad/03-01-07.pdf
18. M. Richters, M. Gogolla. OCL: Syntax, semantics, and tools. *LNCS 2263*, pages 42–68. Springer, 2002.
19. J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
20. J.C. Shepherdson. Negation as failure: a comparison of Clark's completed data base and Reiter's closed world assumption. *J. Logic Programming*, 1:51–79, 1984.
21. The precise UML group http://www.puml.org
22. J. Warmer, A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.
23. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.