Object Oriented Software Systems defined by Constructive Logical Methods

Galán F.J. and Toro M.

Abstract

There exist many formal and semiformal notations for specifying object oriented software systems and many programming techniques that support object oriented concepts. However, correct, extensible and reusable software is difficult to achieve. We are interested in translating non executable system specifications into executable ones and how constructive methods can be used in formal OO software system development.

Keywords: OO system, component, adt, isoinitial models, constructive proofs.

1 Introduction

A widely used technique in modern software engineering is to model a system by a combination of different, but semantically compatible "views" of the system [CD94]. The primary benefit of such approach is to keep very complex systems manageable and to detect misconceptions or inconsistencies at an early stage. Normally, software systems are divided into three models: structural, behavioural and functional models. *Structural model* describes the relationship between the classes of components used in the system as well as the actual configuration of the system components themselves. *Behavioural model* describes the life-cycle of components, i.e. the different configurations of components in the time and *functional model* comprises data transformations and data invariants. In an object-oriented system (based on imperative programming), the basic component is called *class*, which describes an implementation of an abstract data type (ADT), including a set of *methods* that implement operations on the ADT. A class can be composed with others to define new larger (and complex) classes (*generalizations*, *aggregations*,...). At execution time, instances of a class are created dynamically. Such instances are called *objects*. An object contains actual *data* (of the type of its ADT). Its method, when invoked by *messages* sent by other objects, are used to manipulate *its data*.

Reusability arises through *inheritance*. Inheritance allows code reuse, since the code for the methods in a class is generated only once and "shared" by all the objects of any subclass. *Extensibility* results from the ability to obtain new classes adding new characteristics to existing ones. *Reliability* results from the ability to monitor *assertions* and *invariants* contained in classes. Current object-oriented programming technology support these concepts but does not support formal system development. A first requirement to

Dep. Lenguajes y Sistemas Informáticos, Facultad de Informática. Avenida Reina Mercedes s/n c.p.: 41012-SEVILLA. Fax: +34 5 4557139 e-mail: galanm@arturo.lsi.us.es

achieve this objective is to define a system specification language with a precise notation and semantic. Our starting point is a well-defined unit called *component*.

A component is a theory that embodies our knowledge of an element in a system. Within this theory we define semantically notions similar to classes, associations, generalizations, aggregations,... etc.

Correctness of executable systems is established at two levels. Firstly, at component level, models of executable components are isomorphic to models of component specifications. At system level, models of software systems are constructed from models of components in a compositional way.

The paper is organised as follows. Firstly, we introduce preliminary definitions and describe the basic units in the system, the *components*. In the next there sections, we show how constructive methods can be used to obtain correct, extensible and reusable executable systems and finally, we present conclusions and future tasks of our preliminary work.

2 Definitions

Definition 1: A *Type-Component Specification* $\mathcal{T}$ could be regarded as an enriched axiomatisation of a collection of ADT's. However, it has more than the usual ADT's since it can also contain axioms for reasoning about the domain (domain axioms) and reasoning about adt's (induction schemes).

In general, a type-component defines a new abstract data type T starting from predefined type-components. The syntax of a type-component $\mathcal{T}$ is similar to that used in algebraic abstract data types. $\mathcal{T}$ has *function symbols* and *relation symbols*. We consider the partition of the set of function symbols into a set C of *constructors* and a set D of *defined functions*. Constructors are used to build data types, whereas defined functions operate on these data types. Let R be a set of *relation symbols* including the binary equality relation =. A *literal* $r(t_1, ..., t_n)$ is a n-ary relation symbol applied to n terms. An *equation* is a literal with = as relation symbol. *Conditional equations* are equations of the form $c \to l = r$. *Unconditional equations* are equations of the form $l = r$. Equations will be used only from left to right, we call them *rewrite rules*. An *extension operation* introduces a new defined symbol q in a type-component $\mathcal{T}$ through a set D_q of axioms, that we call a *definition* of q. We consider *recursive definitions* and *explicit definitions*.

Definition 2: A *Recursive Definition* of a function or relation is like a primitive recursive definition. If q is a function symbol then D_q is a set of *confluent* and *terminating* rewrite rules and it defines a *total function*. If q is a relation symbol then D_q defines a *decidable relation*.

Definition 3: An *Explicit Definition* of a (new) relation q has the form $\forall x(q(x) \leftrightarrow R(x))$, where R is a formula of the old language, that we call *defining formula*. $R(x)$ is *quantifier free* or contains only *bounded quantifiers*. For example, let x and y be natural variables, in $\forall x, y(q(x, y) \leftrightarrow y = x + 1)$, $y = x + 1$ is quantifier free and in $\forall x, y(q(x, y) \leftrightarrow \forall i < x(y = i + 1))$, $\forall i < x(y = i + 1)$ is bounded quantified.

Definition 4: An *isoinitial model* I of $\mathcal{T}$ is a reachable model such that for any relation r defined in $\mathcal{T}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{T}$. A reachable model is one where each element of the domain can be represented by a ground term.

The *intended model* of type component specification $\mathcal{T}$ is an *isoinitial* model.

The existence of isoinitial models is of course not guaranteed but following the work of [MMO94], we can construct type-components with isoinitial models. A brief explanation of the method is: Firstly, we start with a small component $\mathcal{T}_0$ with an obvious isoinitial model I_0. If the freeness axioms hold in T, then $\mathcal{T}_0$ consists of just the constructor symbols in $\mathcal{T}$ together with their freeness axioms. Then we sucessivelly *extends* $\mathcal{T}_i$ into $\mathcal{T}_{i+1}$ by adding a new function or relation symbol q, together with their axioms D_q (in a *explicit* or *recursive manner*) in such a way that I_i can be expanded into an isoinitial model I_{i+1} of $\mathcal{T}_{i+1}$. For example, if q is a function symbol and D_q is its definition then $\mathcal{T}_{i+1} = \mathcal{T}_i \cup D_q \cup \{\neg t = t'\}$ such that t and t' are ground terms and $\mathcal{T}_i \cup D_q \not\vdash t = t'$. Notice that $\mathcal{T}_{i+1}$ is a recursive axiomatisation iff equality on ground terms is decidable.

Example of object-component and type-component:

Object-component $\mathcal{P}erson$;
`IMPORT`: $\mathcal{O}id, \mathcal{N}at$;
`SORTS`: $Oid, Nat, Person$
`FUNCTIONS`:
for all $oid : Oid, age : Nat, money : Nat$
$<oid, age, money> \Rightarrow Person$
$credit : (Person) \Rightarrow Nat$
`RELATIONS`:
$purch : (Person, Nat)$ (possible purchase)
`AXIOMS`: for all $p : Person, c : Nat$
$p.age \leq s^{(17}(0) \rightarrow credit(p) = 0$
$\neg p.age \leq s^{(17}(0) \rightarrow credit(p) = 2 * p.money$
$p.age \leq s^{(17}(0) \rightarrow purch(p, y) \leftrightarrow$
$\quad \forall i \leq p.money(y = i)$
`BEHAVIOUR`:
$birthday : (Person)$
$birhtday(p.oid, p.age, p.money) \leftrightarrow$
$(<p.oid, p.age, p.money>, <p.oid, s(p.age), p.money>)$
$buy : (Person, Nat)$
$buy(p, c) \leftrightarrow c \leq credit(p) \wedge +purch(p, c)$

Type-component $\mathcal{L}istNat$;
`IMPORT`: $\mathcal{N}at$
`SORTS`: $Nat, ListNat$
`FUNCTIONS`:
$nil :\Rightarrow ListNat$
$. : (Nat, ListNat) \Rightarrow ListNat$
$nocc : (Nat, ListNat) \Rightarrow Nat$
`RELATIONS`:
$elemi : (ListNat, Nat, Nat)$
`AXIOMS`:
$\neg nil = a.B$
$a.B = c.D \rightarrow a = c \wedge B = D$
$H(nil) \wedge \forall a, J(H(J) \rightarrow H(a.J))$
$\quad \rightarrow \forall L(H(L))$
$nocc(x, nil) = 0$
$a = b \rightarrow nocc(a, b.L) = nocc(a, L) + 1$
$\neg a = b \rightarrow nocc(a, b.L) = nocc(a, L)$
$elemi(L, 0, x) \leftrightarrow \exists B(L = x.B)$
$elemi(L, s(i), x)$
$\quad \leftrightarrow \exists b, B(L = b.B \wedge elemi(B, i, x))$

Definition 5: An *Object-Component Specification* $\mathcal{O}$ could be regarded as a type-component specification $\mathcal{T}$ with the following differences: terms in $\mathcal{O}$ (objects) have unique *identity*. Identity is represented by ground terms of the sort Oid (defined in the type component specification $\mathcal{O}id$). We consider $< v_{Oid}, v_{T_1}, ..., v_{T_k} >$ as the unique *object constructor* where $v_{Oid}, v_{T_1}, ..., v_{T_k}$ are variables of sorts $Oid, T_1, ..., T_k$ respectively. $Oid, T_1, ..., T_k$ sorts are defined in $\mathcal{O}id, \mathcal{T}_1, ..., \mathcal{T}_k$ type-component specifications and included in the `IMPORT` section of $\mathcal{O}$. *(Total) functions* and *relations* in $\mathcal{O}$ have profiles $f_i(S_j, ..., S_k) \Rightarrow S_m$ and $r_j(O, S_j, ..., S_k)$ respectively where for all m, j and k, S_r, S_j, S_k are included in section `SORT` of $\mathcal{O}$. We consider a new section called `BEHAVIOUR` where we define behaviour relations of the objects in $\mathcal{O}$.

If-and-only-if definitions are often too restrictive and inflexible, and we would prefer weaker forms of specifications that admit multiple interpretations, corresponding to different program behaviours that are all correct for the problem at hand. To this end, we accept *conditional definitions* (i.e. $\forall x(c(x) \rightarrow r(x) \leftrightarrow d(x))$).

The *state* of any object o of $\mathcal{O}$ is represented by ground instances of its object constructor and the meaning of its relations. Object constructors change in the time, preserving oid values, (i.e. from $<oidvalue, v_1, ..., v_k>$ to $<oidvalue, v_1', ..., v_k'>$). The meaning of equality relation is *fixed* (totally defined functions). The rest of relations, the meaning of "iff" relations is *fixed* (constant) in all the life-cycle of o and the meaning of conditional relations is *variable* in the following sense: $\forall x(c(x) \rightarrow r_1(x) \leftrightarrow r_2(x))$, can always be given as a pair of implications:

$$\forall x(r_1^{min}(x) \rightarrow r_1(x))$$

$$\forall x(r_1(x) \rightarrow r_1^{max}(x))$$

where $r_1^{min}(x) \leftrightarrow c(x) \wedge r_2(x)$ and $r_1^{max}(x) \leftrightarrow \neg c(x) \vee r_2(x)$. At each moment in the life-cycle of o, we can accept any definition for r_1 such that $r_1^{min} \subseteq r_1 \subseteq r_1^{max}$.

Definition 6: Following definition 4, the *Intended model* I of an object component specification $\mathcal{O}$ is a reachable model such that for any relation r defined in $\mathcal{O}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{O}$. We consider object components O as *aggregates* of type components (Oid,T_1,...,T_k). If type components have isoinitial models then each object can be represented by a ground term (reachability). Functions and relations in $\mathcal{O}$ are defined as explicit definitions (quantifier free or bounded quantification) in the language $\{T_1 \cup ... \cup T_k\}$ hence for any relation r defined in $\mathcal{O}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{O}$. Conditional specifications in $\mathcal{O}$ can have many interpretations then there are many expansions of the intended model of $\mathcal{O}$, we will call $Models(\mathcal{O})$ to the set of all expansions of the intended model of $\mathcal{O}$. `BEHAVIOUR` section in $\mathcal{O}$ defines:

1. *Elementary Transitions*, et_k, are explicit definitions of the form:

   $$et_k(l) \leftrightarrow (<l>, <f(l)>)$$

   where l is an *object constructor pattern* and f is an aggregate of functions (type components functions) applied to elements of $<l>$. Elementary transitions preserve object identifications (Oid). Left and right components in et_k represent the *initial* and *final state* of the transition respectively. If $et_k(l)$ is an elementary transition defined in $\mathcal{O}$ and $<c>$ is the (ground) object constructor of any object o of $\mathcal{O}$ and there exists a ground substitution σ such that $<c> = \sigma <l>$ then transition et_k on o can be defined as:

   $$o_{new} = o_{old} \mid_{<c>}^{<f(c)>}$$

   where o_{old} is the object at the initial state of et_k and o_{new} is the same object replacing its constructor $<c>$ by $<f(c)>$.

2. *Complex Transitions*, ct_j, built from relations, r_i, defined in `RELATIONS` section and from elementary transitions, et_k, defined in `BEHAVIOUR` section.

   Complex transition patterns are of the form:

   $$(*)\ \forall x(ct_j(x) \leftrightarrow r_i(x) \wedge et_k(...) \wedge ...)$$

   In (*), if r_i is defined by a conditional specification then r_i *can* be prefixed with either a "+" symbol or a "−" symbol. Roughly, a "+" prefix in r_i means that if the intended model of $\mathcal{O}$, $I \not\vdash r_i(t)$, (t ground), and if there exists an intended model $I_{new} \in Models(\mathcal{O})$ such that $I_{new} \vdash r_i(t)$ then o "changes its model" from I to I_{new}. In a similar way, a "−" prefix in r_i means that if $I \vdash r_i(t)$, (t ground), and if there exists an intended model $I_{new} \in Models(\mathcal{O})$ such that $I_{new} \not\vdash r_i(t)$ then o "changes its model" from I to I_{new}.

Definition 7: the *life cycle* of an object $o \in \mathcal{O}$ is represented by a sequence of object constructors, intended models of $\mathcal{O}$ and elementary and complex transitions that are true in I.

$$life\ cycle = \{< c >, I, \{et_k, ct_j\}\}_i \quad \text{for } i = 1..\infty$$

We must proceed with special care in the construction of the intended model I of $\mathcal{O}$ with `BEHAVIOUR` section. The behaviour of an object in $\mathcal{O}$ ($\{et_k, ct_j\}$) depends on its state ($<c>$ and I). Hence, the construction of intended models I is fundamental in order to prevent bizarre behaviours.

3 Model Construction Method

The intended model of type and association component specifications is an *isoinitial model*. We construct it following the ideas of [MMO94] (Definition 4).

We present the following algorithm in order to construct the intended models of the rest of components (object and extension components, we will refer them as $\mathcal{O}$):

1. `IMPORT` sections in $\mathcal{O}$ must include only type-components with isoinitial models.

2. Only ground constructor terms are allowed for objects in $\mathcal{O}$.

3. Each function f in $\mathcal{O}$ is defined by a set of confluent and terminating equations D_f.

4. Each relation r_i in $\mathcal{O}$ is defined by explicit or recursive definitions. In a incremental construction of the intended model, (consider $I_{1..3}$ as the intended model of any component, (incrementally constructed from 1)-3) previous steps), if we want to expand $\mathcal{O}$ with a relation r_i defined by a conditional specification then there exist *many expansions* of $I_{1..3}$. For example, $I_{1..4}$ may be any intended model where r_i is interpreted as r_i^{max} or may be an isoinitial model where r_i is interpreted as r_i^{min},... etc.

5. Let $I_{1..4}$ be the isoinitial (incrementally constructed for $\mathcal{O}$) from 1)-4) previons steps. Let E be the set of elementary transitions in $\mathcal{O}$. At this point, we are interested in the expasion of $I_{1..4}$ with E. **If** for all pair of elementary transitions et_i, et_k in E, left-hand sides are not unifiable (non-ambiquity) and pattern variables in right-hand sides are included in pattern variables in left-hand sides ($et_k(l) \leftrightarrow (<l>, <f(l)>)$) **then** the expansion of $I_{1..4}$ with E is defined as:

$$\begin{aligned} I_{1..5} \quad = \quad & I_{1..4} \cup \{et_k(t) \mid \text{ for all } t \text{ ground and } t = \sigma l\} \\ & \cup \\ & \{\neg et_k(t') \mid \text{ for all } t' \text{ ground and not unifiable with } l\} \end{aligned}$$

   else the expansion is not defined.

6. Let $I_{1..5}$ be the model of $\mathcal{O}$ (incrementally constructed from 1)-5) previous steps). At this point, we expand $I_{1..5}$ adding the axioms which define complex transitions ct_j.

   Cases:

(a) **If** only an elementary transition $et_k(m)$ with m as an instance of l ($et_k(l) \rightarrow$ ($<l>, <f(l)>$) and not prefixed relations are present in the body of ct_j (i.e. $ct_j(x) \leftrightarrow et_k(m) \wedge Rest(x)$)

then

$$\begin{aligned} I_{1..6} &= I_{1..5} \cup \{ct_j(t) \mid t \text{ ground and } I_{1..5} \vdash et_k(...) \wedge rest(t)\} \\ &\quad \cup \\ &\quad \{\neg ct_j(t') \mid t' \text{ ground and } I_{1..5} \nvdash et_k(...) \wedge rest(t')\} \end{aligned}$$

(b) If only elementary transitions and not prefixed relations are present in the body of ct_j ($ct_j(x) \leftrightarrow et_1(m) \wedge et_2(n) \wedge ... \wedge et_n(s) \wedge rest(x)$) then the expansion is defined **if** elementary transitions in the body of ct_j form a chain Ch. The complex transition ct_j can be replace by $ct_j(x) \leftrightarrow Ch \wedge rest(x)$ and then the expansion proceed as in 1) with this new definition **else** the model is not defined.

A set of elementary transitions St is considered as a chain if there exists a permutation of St: $perm(St) = \{et_1(m) \leftrightarrow (<m>, <f_1(m)>, et_2(n) \leftrightarrow (<n>, <f_2(n)>), ..., et_n(s) \leftrightarrow (<s>, <f_n(s)>)\}$ such that m is an instance of the object constructor pattern in et_1 definition, n is an instance of the object constructor pattern in et_2 definition,..., s is an instance of the object constructor pattern in et_n definition and for all $t_1 = \sigma_1 m$ ground, $f_1(t_1)$ is an instance of n and, ..., and $f_{n-1}(t_{n-2})$ is an instance of r and for all $t_{n-1} = \sigma_{n-1} r$, ground, $f_n(t_{n-1})$ is an instance of $f_1(m)$ then $perm(St)$ can be considered as a new elementary transition

$$Ch(m) \leftrightarrow (<m>, <f_n(f_{n-1}(....f_1(m))...)>)$$

(c) **If** only + prefixed and not prefixed relations are present in the body of ct_j (i.e. $ct_j(x) \leftrightarrow +r(m) \wedge rest(x)$) where r is a relation defined by a conditional specification and m is a term pattern. The model I of $\mathcal{O}$ *can contain* any of the ground instances of $r(m)$, $r(t)$, such that $r(t)$ is not true in the (initial) interpretation of r ($I_{1..4}$) and $r(t) \in r^{max}$. Let $Ins(r,m)$ be the set of all the ground instances of $r(m)$ that are in r^{max} but they are not true in $I_{1..4}$. Let I_{exp} be the $I_{1..6}$ model of $\mathcal{O}$ expanded with the set $Ins(r,m)$, ($I_{exp} = I_{1..6} \cup Ins(r,m)$), **then** for all + prefixed relation:

$$\begin{aligned} I_{1..7} &= I_{exp} \cup \{ct_j(t) \mid t \text{ ground and } I_{exp} \vdash r(t) \wedge rest(...)\} \\ &\quad \cup \\ &\quad \{\neg ct_j(t') \mid t' \text{ ground and } I_{exp} \nvdash r(t') \wedge rest(...)\} \end{aligned}$$

Hence, all object $o \in \mathcal{O}$ in a system begin its execution with a behaviour ct_j bounded to the interpretation of r in $I_{1..6}$ but extensible to $r \cup Ins(r,m)$ in a correct way. Any execution of $ct_j(t)$ where $r(t) \in Ins(r,m)$ implies a *dynamic* change in r to $r \cup r(t)$.

(d) **If** only $-$ prefixed and not prefixed relations are present in the body of ct_j (i.e. $ct_j(x) \leftrightarrow -r(s) \wedge rest(x)$) where r is a relation defined by a conditional specification and s is a term pattern. The model I of $\mathcal{O}$ *can not contain* any of the ground instances of $r(s)$, $r(t)$, such that $r(t)$ is true in the (initial) interpretation of r ($I_{1..4}$) and $r(t) \notin r^{min}$. Let $Del(r,s)$ be the set of all the ground instances of $r(s)$ that are in $I_{1..4}$ but they are not true in r^{min}. Let I_{red} be the $I_{1..6}$ model of $\mathcal{O}$ reduced with the set $Del(r,s)$, ($I_{red} = I_{1..6} - Del(r,s)$), **then** for all $-$ prefixed relation:

$$\begin{array}{rcl} I_{1..7} & = & I_{red} \cup \{ct_j(t) \mid t \text{ ground and } I_{red} \vdash r(t) \wedge rest(...)\} \\ & & \cup \\ & & \{\neg ct_j(t') \mid t' \text{ ground and } I_{red} \not\vdash r(t') \wedge rest(...)\} \end{array}$$

Hence, all object $o \in \mathcal{O}$ in a system begin its execution with a behaviour ct_j bounded to the interpretation of r in $I_{1..6}$ but reducible to $r - Del(r,s)$ in a correct way. Any execution of $ct_j(t)$ where $r(t) \in Del(r,s)$ implies a *dynamic* change in r to $r - r(t)$.

(e) **If** $+$ and $-$ prefixed relations and (possible) elementary transitions and (possible) not prefixed relations appear in the body of ct_j (the more general situation) **then** the model is defined **iff** the following condition holds: either the intersection between $+$ and $-$ prefixed relation symbols is empty or is not empty but the implied relations do not have ground instances in common, (i.e. $+r(m) \wedge .... \wedge -r(s)$ and $\not\exists t \mid t$ is a ground instance of m and s). **If not**, the model of $\mathcal{O}$ is not defined.

For all $+$ and $-$ prefixed relation: $I_{expred} = I_{1..6} + Ins(r,m) - Del(r,s)$

$$\begin{array}{rcl} I_{1..7} & = & I_{expred} \cup \{ct_j(t) \mid t \text{ ground and } I_{expred} \vdash r(t) \wedge rest(...)\} \\ & & \cup \\ & & \{\neg ct_j(t') \mid t' \text{ ground and } I_{expred} \not\vdash r(t') \wedge rest(...)\} \end{array}$$

For all (only) $+$ prefixed relations we follow c) step and for all (only) $-$ prefixed relations we follow d) step.

The model construction is an iterative process, at each step, we expand the model with a new complex transition relation ct_j.

4 Extension Components

Extension components are components obtained from other components adding new axioms and, possibly, new symbols. An extension component inherits all the axioms and definitions that have been developed in original components.

Example of extension component:

Extension component $\mathcal{E}Person$;
`EXTEND`: $\mathcal{P}erson$;
`RELATIONS:`
$rich(Person, Nat)$
`AXIOMS:` $p : Person, m : Nat$
$rich(p) \leftrightarrow credit(p) \geq s^{(200}(0)$
`BEHAVIOR:`
$deposit : (Person)$
$deposit(p.oid, p.age, p.money) \leftrightarrow$
$(<p.oid, p.age, p.money>, <p.oid, p.age, p.money + p.money>)$
$ebuy : (Person, Nat)$
$ebuy(p,c) \leftrightarrow \forall i \leq c\,(i \leq s^{(100}(0) \wedge +purch(p,i))$

In this example, we show an extension of $\mathcal{P}erson$ (rich person). If a person p is a rich person then he/she must be interpreted as $Person$ enriched with new structural properties, i.e. *rich* relation and new behavioural properties i.e. *deposit* and *ebuy* transitions. The model of $\mathcal{E}Person$ is defined iff $\mathcal{P}erson+\mathcal{E}Person$ has model applying construction method in previous section. In a + operation between object component specifications, two object component specifications are put together (section by section). Like classes (based on imperative programming), we can construct a hierarchy of components by *extension components*.

5 Association Components

For us, a system is a *set* of related object-components. These relations are established by *association components*. An association component does not exist per se, it need of object components in its definition.

Example of association component:

Association component $\mathcal{P}erson\mathcal{C}ompany$;
IMPORT: $\mathcal{P}erson, \mathcal{C}ompany$;
SORTS: $Person, Company, PersonCompany$;
FUNCTION:
$<Person, Company>\Rightarrow PersonCompany$
RELATIONS:
$invariant_1 : (Person, Company)$
$invariant_2 : (Person)$
$invariant_3 : (Person, Company)$
AXIOMS:$p : Person, c : Company$
$invariant_1(p,c) \leftrightarrow \exists <i,j>: PersonCompany\,(p.oid = i.oid \wedge c.oid = j.oid)$
$p.age \geq s^{(18}(0) \wedge p.age \leq s^{(65}(0) \rightarrow invariant_2(p) \leftrightarrow \exists c(invariant_1(p,c))$
$invariant_3(p,c) \leftrightarrow p.age > s^{(65}(0) \rightarrow \neg invariant_1(p,c)$

Association components do not have BEHAVIOUR sections. We consider association components in a system as a set of invariants. These invariants govern which (ground) instances of the association components are true and which are not true in the system. Quantifications must be understood as *bounded quantifications* on (*finite*) *populations* of object components. At execution time, changes in object components in the association promote changes in association objects (dynamic insertion and deletion of association instances). We can construct the intended model of association components following our model construction method restricted to the 1..4 steps.

Finally, the system must be undestood as a collection ("put together") component specifications then, for us, the intended model of any software system results from the composition of the models of its components. All of the type components, object components and extension components in a system must have intended models. Association components can be considered as the "glue" between the previous components; any object population must preserve invariant relations contained in association components. Finally, each component in a system must be considered as a theory and the composition of these theories represents the system. Hence, intended model of the system is defined iff each component has intended model.

6 Executable Components and Systems

In this section, we briefly show how declarative semantic of components can be intepreted in operational terms. We will specify our operational semantic by the use of definitional trees, a concept introduced by [Ant92] to define efficient normalization strategies. $Tree$ is called defintional tree with pattern $l \to r$ iff one of the following cases holds: $Tree = rule(l \to r)$ where $l \to r$ is a variant of a rewrite rule in the component specification.

$Tree = branch(\pi, p, Tree_1, ...Tree_k)$ where π is a pattern, p is an ocurrence of a variable in π, $c_1...c_k$ are different constructors of the sort of $\pi \mid_p$ (argument in position p of π) $(k > 0)$ and, for $i = 1, ...k$, $Tree_i$ is a definitional tree with pattern $\pi[c_i(x_1, ...x_n)]_p$ where n is the arity of c_i and $x_1, ..., x_n$ are new distinct variables. A definitional tree of an *n-ary defined function* f is a definitional tree $Tree$ with pattern $f(x_1, ..., x_k)$ where $x_1, ..., x_n$ are distinct variables such that for each rule $l \to r$ with $l = f(t_1, ..., t_n)$ there is a node rule $(l' \to r')$ in $Tree$ with l variant of l'.

We define the validity of an equation as a strict equality on terms by the following rules, where $\wedge$ is assumed to be a right-associative infix symbol.

$$
\begin{array}{ll}
c = c \to true & \forall c/0 \in \mathcal{C} \\
c(x_1, ..., x_n) = c(y_1, ..., y_n) \to (x_1 = y_1) \wedge ... \wedge (x_n = y_n) & \forall c/n \in \mathcal{C} \\
true \wedge x \to x &
\end{array}
$$

(note: we write c/n for n-ary constructors)

Example of definitional tree:

$$
\begin{array}{l}
0 + y = y \\
s(x) + y = s(x + y)
\end{array}
$$

its definitional tree is

$$branch(x + y, 1, rule(0 + y \to y), rule(s(x) + y \to s(x + y)))$$

Relations in $\mathcal{O}$ can be translated into a variant of definitional trees extended with *and* and *or* nodes. Let D_q be the set of axioms that defines q, for example,

$$
\begin{array}{l}
elemi(L, 0, x) \leftrightarrow \exists B(L = x.B) \\
elemi(L, s(i), x) \leftrightarrow \exists b, B(L = b.B \wedge elemi(B, i, x))
\end{array}
$$

1. Establish π as $q(x_1, ..., x_n)$ with $x_1, ..., x_n$ distinct variables (i.e. $elemi(x, y, z)$)

2. Construct $Tree$ for all p variable position in pattern π, where existentially quantified variables in the body are replace by *or* nodes and universally quantified variables in the body are replace by *and* nodes.

$$
\begin{array}{l}
Tree_{elemi} = branch(elemi(x, y, z), 1, \\
\quad branch(elemi(nil, y, z), 2, \\
\quad\quad or[rule(elemi(nil, 0, z) \to nil = z.nil), \\
\quad\quad rule(elemi(nil, 0, z) \to nil = z.x_2.L_2)]), \\
\quad\quad or[rule(elemi(nil, s(i), z) \to nil = 0.nil \wedge elemi(nil, i, z)), \\
\quad\quad rule(elemi(nil, s(i), z) \to nil = s(j).nil \wedge elemi(nil, i, z)), \\
\quad\quad rule(elemi(nil, s(i), z) \to nil = 0.x_2.L_2 \wedge elemi(x_2.L_2, i, z)), \\
\quad\quad rule(elemi(nil, s(i), z) \to nil = s(j).x_2.L_2 \wedge elemi(x_2.L_2, i, z))])
\end{array}
$$

$$\begin{array}{l} branch(elemi(x_1.L_1, y, z), 2, \\ \quad or[rule(elemi(x_1.L_1, 0, z) \to x_1.L_1 = z.nil), \\ \quad rule(elemi(x_1.L_1, 0, z) \to x_1.L_1 = z.x_2.L_2)]), \\ \quad or[rule(elemi(x_1.L_1, s(i), z) \to x_1.L_1 = 0.nil \wedge elemi(nil, i, z)), \\ \quad rule(elemi(x_1.L_1, s(i), z) \to x_1.L_1 = s(j).nil \wedge elemi(nil, i, z)), \\ \quad rule(elemi(x_1.L_1, s(i), z) \to x_1.L_1 = 0.x_2.L_2 \wedge elemi(x_2.L_2, i, z)), \\ \quad rule(elemi(x_1.L_1, s(i), z) \to x_1.L_1 = s(j).x_2.L_2 \wedge elemi(x_2.L_2, i, z))])) \end{array}$$

An example of execution for $elemi(s(0).0.s(0).nil, s(0), 0)$: Firstly, this goal unifies with $branch(elemi(x_1.L_1, y, z)$ and then with second *or* node, we inspect each rule in the *or* node. The fourth rule constructs the solution $true$ in a recursive manner on the subgoal $elemi(0.s(0).nil, 0, 0)$.

Induction schemes can be translated into definitional trees in the following manner:

$$(H(nil) \wedge H(I) \to H(a.I)) \to \forall x H(x)$$

is represented by

$$branch(ind(H, x), 2, and[rule(H(nil) \to v_1), rule(H(I) \to true), rule(H(a.I) \to v_2)])$$

where H is bounded to relation symbols, v_1, v_2 are distinct variables bounded to $true$ or $false$ values resulting in the proofs of $H(nil)$, $H(I)$ and $H(a.I)$ respectively.

For example, $\forall x(elemi(x, 0, 0)) \to true$:

$$\begin{array}{l} branch(ind(elemi(x, 0, 0), x), 2, \\ \quad and[rule(elemi(nil, 0, 0) \to v_1), \\ \quad (rule(elemi(I, 0, 0) \to true), (rule(elemi(a.I, 0, 0) \to v_2) \end{array}$$

$$\begin{array}{l} rule(elemi(nil, 0, 0) \to nil = 0.nil \to_{Tree_{elemi}} false) \\ rule(elemi(L_1, 0, 0) \to true) \\ rule(elemi(x_1.L_1, 0, 0) \to nil = 0.nil \to_{Tree_{elemi}} false \\ rule(elemi(x_1.L_1, 0, 0) \to nil = 0.x_2.L_2 \to_{Tree_{elemi}} false \end{array}$$

(Hence, $\forall x(elemi(x.0.0)) \to true$ is not true in $ListNat$).

Elementary transitions $et_k(l) \leftrightarrow (<l>, <f(l)>)$ can be represented as $Tree_{et_k} = rule(l \to f(l))$. Complex transitions are represented as relations including elementary transitions. $Ins(r, m)$ and $Del(r, s)$ sets can be computed at execution time: unification algorithm proves if $r(t)$ is an instance of $r(m)$ or $r(s)$. $+$ and $-$ prefixed relations and state changes can be simulated with an assert-retract Prolog mechanism. Extension and association components are not distinct of object components then we proceed in a similar way. Dynamic creation and destruction of links can be simulated by assert and retract's. Negative literals $\neg r(t)$ can be computed in the following manner: we inspect $r(t)$ if it can be rewrite as $true$ then $\neg r(t)$ is not true and if $r(t)$ can not be rewrite as $true$, then $\neg r(t)$ is true. This is possible because we treat only with decidable relations in component specifications.

Finally, definitional trees, unification algorithms and assert/retract mechanisms are necessary software elements in order to obtain executable system but not sufficient. How govern object transitions? and how stimulate to the system in order to change its state?. First question implies that OO systems need additional controller components that govern transitions and preserve invariants. Second question implies the concept of *event* and link to the previous question. These problems form our future work.

7 Conclusions and Future Work

In this preliminary work, we have shown how constructive method can be used in the construction of executable systems totally correct with respect to system specifications. System construction is automatic, provided type-components and object-components with intended models. Reusability of systems have been increased due to it is achieved at semantic level. Our systems are interpreted as theories, then adding new specifications (in the way explained in our work) is equivalent to expand our systems. Synthesis of logic programs, within the O-O context, is treated in [KO95]. Dr. Lau and Dr. Ornaghi show how from *frameworks* it is possible to obtain programs. There are two difference wrt our work: a) the kinds of specifications in [KO96] are intended for deductive synthesis, however we have oriented our work for constructive synthesis and b) we treat dynamic aspects of objects, however in [KO95] the authors are centered in static aspects mainly.

Our work is only at initial state and much effort is needed in order to define different semantic characterizations of *generalization* and *aggregations* components and *executable systems*.

References

[Ant92] Antoy S., *Definitional Trees.* In Proc. of the Third Int. Conference on Algebraic and Logic Programming. Springer 1992.

[CD94] Cook S. Daniels J. *Designing Object-Oriented Systems* Prentice Hall 1994.

[Gal95] Galan Morillo F.J. and Toro M. *Sintesis de Programas Logicos.* In Proc. of Gulp-Prode 1995. Marina di Vietri. Italy.

[KO95] Lau k.K. and Ornaghi M. *Towards an Methodology for Deductive Synthesis of Logic Programs.* In 5th Int. Workshop LOPSTR'95. Springer 1995.

[KO96] K.K. Lau and M. Ornaghi. *Forms of Logic Specifications: A preliminary Study.* In 6th Int. Workshop LOPSTR'96. Springer 1996.

[MMO94] Miglioli P., Moscato U., Ornaghi M. *Abstract Parametric Classes and Abstract Data Types defined by Classical and Constructive Logical Methods.* J. Symbolic Computation 1994.

[W95] Wieringa R.J. *LCM and MCM. Specification of a control system using dynamic logic and process algebra.* Ed. Lewerentz y T. Linder. LNCS 891. Springer-Verlag 1995.