Behaviour Specification of Active Objects in Open Generative Communication Environments

Tom Holvoet
K. U. Leuven, Belgium
Tom.Holvoet@cs.kuleuven.ac.be

Thilo Kielmann
University of Siegen, Germany
kielmann@informatik.uni-siegen.de

Abstract

Coordination models based on generative communication are well established for modelling interactions between agents in open systems. Because such models focus on shared data spaces and not on the active agents operating on them, they lack adequate modelling power for specifying agent behaviour.

Our work is based on the Objective Linda coordination model [7] which exploits generative communication and object–based modelling in order to meet the requirements of open systems. For Objective Linda, we present a high–level Petri Net formalism for agent behaviour specification that combines an easily understandable, visual representation with the applicability of formal reasoning on agent behaviour. Based on this formalism, we introduce a notion of types and subtyping for active agents. We illustrate the benefits of our work on an example from open systems design.

1. Introduction

In *open systems*, active entities ("agents", or "actors") may dynamically join and leave. They hence constitute evolving, self-organizing systems of interacting agents [1, 2]. Programming open systems is primarily concerned with the *coordination* of concurrently operating active entities. Coordination involves management of the communication between these entities. Coordination models based on *generative communication* are considered to be the most prospective approach to this research domain. Generative communication, as initially introduced in [4], is based on a shared data space in which data items can be stored ("generated") and retrieved later on.

This kind of communication offers two major advantages. First, it is uncoupled: a potential reader of some data item does not have to take care about it (as e.g. with rendezvous mechanisms) until it actually needs it. The reader does not even have to exist at the time of storing. Second, agents (the active entities) are able to communicate although they are anonymous to each other. This uncoupled and anonymous communication style directly contributes to the design of coordination models for open systems, since it allows communication with incomplete knowledge about the system configuration. Uncoupled communication enables to cope with dynamically changing configurations in which agents move or temporarily disappear. Anonymous communication allows to communicate with unknown agents. We define an *open generative communication environment* as an open system in which active objects (called *agents*) communicate in a generative manner.

Coordination models only focus on the operations on object spaces. Our aim is to provide plausible definitions (based on formal specifications) for agent *behaviour* as well as for *types* and *subtypes* of agents. This work is based on Objective Linda [7], a coordination model (based on objects and generative communication) which has been designed in order to meet the requirements of open systems. High–level Petri nets [6] are an attractive formalism for behaviour specifications due to their easily understandable visual representation, their power in expressing concurrency, causality, and non–determinism. A combination of ideas and features from Coloured PNs [5] and Time PNs [9] leads to a well–suited formal model for behaviour specification of active objects in open generative communication environments. This new formalism, called PNSOL, is used for presenting Petri net semantics of Objective Linda operations.

Having a formal representation of agents allows us to provide a formal definition for typing agent behaviour. Agent types are based on the *observable behaviour* of agents which corresponds to the set of objects the agent produces, which other agents can retrieve from object spaces. We define types for agents as sets of observable behaviours, and a subtyping relation based on subset inclusion, correspondingly. We illustrate adequacy and expressiveness of these definitions on a typical example.

The formalism we use for modelling agents does not only allow us to define observable behaviour in a generative environment, it also offers a means to perform type-checking operations based on a elementary property of Petri nets, namely testing of *liveness*.

The remainder of this paper is organized as follows. In Sect. 2, we introduce our coordination model Objective Linda. In Sect. 3, we present the high–level PN formalism that is used to model Objective Linda's operations on object spaces. We illustrate it on an example from open system design in Sect. 4. Then, we introduce our notions of types and subtyping in Sect. 5 which we illustrate on an enhanced version of our example in Sect. 6. Finally, we consider related work and conclude in Sect. 7 and 8, respectively.

2. Objective Linda

We briefly introduce the coordination model Objective Linda which we use as the basis of our work. An in–depth description of features and design decisions can be found in [7].

Objective Linda is based on the principles of Linda [4] and has been designed in order to meet the requirements of open systems. In Objective Linda, objects to be stored in *object spaces* are self–contained entities; their interface operations only affect their encapsulated object state. Objects are instances of abstract data types which are described in a language–independent notation, called *Object Interchange Language* (OIL). Actual programs may hence be written in conventional object–oriented languages to which a binding of OIL types (e.g. to language–level classes) can be declared. In OIL, all types form a type hierarchy having a common ancestor called *OIL_object* which defines the basic operations needed by all types. OIL allows subtyping according to the "principle of substitutability" [16] such that an object of type S which is a subtype of T can be used whenever an object of type T is expected.

Object matching (the process of identifying objects to be retrieved from object spaces) is based on object *types* and *predicates* defined by type interfaces. A potential reader has to specify the type of objects it wishes to obtain from an object space and additionally a predicate from the type interface that further selects the objects of a given type matching a specific request. Subtype relations are of course also exploited by object matching.

The matching predicates are directly integrated into the types on which they operate. Therefore, the type *OIL_object* provides a predicate *match* which takes an object of the same type as parameter and returns a boolean value deciding whether a given object matches certain requirements. Several variants of matching a type can be selected by presetting the encapsulated state of the object provided to a matching operation, which we call a *template object*. The type of objects to be matched is denoted by the template–object's type.

In Linda, active tuples are treated as functions and are converted into passive tuples after termination, yielding their results. In contrast with this functional view, Objective Linda treats active objects as encapsulated and reactive agents. Active objects simply disappear after termination. Agents can only be observed by monitoring the passive objects they produce.

Configurations in Objective Linda consist of active as well as passive objects, and object spaces. Active objects have, from the moment of their activation on, access to two particular object spaces: (1) their *context* which is the object space on which the corresponding *eval* operation has been performed, and (2) a newly created object space called *self* which is directly associated to the object. With this basic mechanism, hierarchies of nested object spaces can be built.

The *context* and *self* object spaces do not suffice for expressing all coordination problems. Therefore, we provide a mechanism for allowing agents to attach to other, already existing object spaces. This mechanism should reflect that object spaces are not part of agents but are accessed by references, and hence are shared between agents. In order to avoid problems with direct (low level) references as well as with global naming schemes, it is necessary to introduce a construct (using generative communication) that allows agents to attach to existing object spaces. Objective Linda therefore introduces a special subtype of *OIL_object* which is called *object space logical*. *Logicals* combine a reference to an object space with a logical identification such that object spaces can be found by matching properties of *logical* objects. These properties can of course be customized to application needs via subtyping. Agents willing to let others attach to object spaces they are already attached to simply create a *logical* object including the reference to the object space to be made available and a convenient logical identification for that object space. This *logical* is then *out*'ed to an object space. An agent A willing to attach to object space N must call a special operation called *attach* on the object space O in which the corresponding *logical* object for N is stored. This operation has two effects: (1) O verifies that N can be attached to (is reachable, allows attachment, etc.), and (2) returns a reference to N which is locally useful to A.

Besides adapting the Linda model to object-orientation, Objective Linda also provides an improved set of operations on object spaces in order to reflect open systems. First, Objective Linda introduces a *timeout* parameter to its operations that determines how long an operation should block before a failure is reported. It can vary from zero to a value indicating an infinite delay. Second, Linda's ability to retrieve only one object at a time from an object space is too restrictive. It is e.g. impossible to non–destructively iterate over all objects of a particular kind [7]. Additionally, synchronization problems can be dealt with more adequately when multiple objects may be consumed atomically from object spaces. These observations lead to the introduction of *multisets of objects* as parameters and results of operations on object spaces. *in* and *rd* specify multisets of objects to be retrieved by two parameters, namely *min* and *max*. *min* gives the minimal number of objects to be found in order to successfully complete the operation whereas *max* denotes an upper bound allowing to retrieve (small) portions of all objects of a kind. An infinite value for *max* allows to retrieve all currently available objects of a kind. For consistency and simplicity reasons, we also use multisets of objects for *out* and *eval*. We can now informally present Objective Linda's operations on object spaces. We use a binding to the Eiffel language as notation.

out (m : Multiset; timeout : Real) : Boolean
Tries to move the objects contained in *m* into the object space. Returns *true* if the operation completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

in (o : OIL_OBJECT; min, max : Integer; timeout : Real) : Multiset
Tries to remove multiple objects $o'_1 \dots o'_n$ matching the template *o* from the object space and returns a multiset containing them as soon as at least *min* matching objects could be found within *timeout* seconds. Then, the multiset contains at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, *Result.Void* is true.

rd (o : OIL_OBJECT; min, max : Integer; timeout : Real) : Multiset
Analogous to *in* except that objects are not removed from the object space but clones of them are returned instead.

eval (m : Multiset; timeout : Real) : Boolean
Tries to move the objects contained in *m* into the object space and starts their activities. Returns *true* if the operation could be completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

attach (o : OS_Logical; timeout : Real) : Object_Space
Tries to get attached to an object space for which an *OS_Logical* matching *o* can be found in the current object space. Returns a valid reference to the newly attached object space if a matching object space logical could be found within *timeout* seconds; otherwise *Result.Void* is true.

3. A high-level Petri net formalism for agent behaviour specification

We propose to use Petri nets for specifying *behaviour* because they make a simple and clear formalism with a visual representation, they are powerful in expressing concurrency, causality and nondeterminism, and they can rely on a substantial theoretical background.

In our approach, systems are modelled as dynamic sets of cooperating agents. Instead of modelling an entire system by a single Petri net, we model each agent by a separate Petri net, called an *agent net*. Besides improving modularity, this design is vital for open systems, where agents can join and leave systems at run time.

The agent net, as an encapsulated part of the agent description, is a Petri net specifying the agent's behaviour. The object spaces, through which agents interact, and the *agent space* (representing a virtual space containing all agents in a system) are modelled as **places** in the agent net – we call these the object space places and the agent space place, respectively. An agent can perform two kinds of actions, which are each represented by transitions: actions that correspond to Objective Linda's operations, and internal actions, representing internal computations.

The fact that, due to the nature of open systems, it is impossible to construct one Petri net for the entire system does **not** imply that one is not interested in reasoning on the behaviour of the system at a particular moment in time or in a particular system configuration. For that purpose, one can take a snapshot of the system, consisting of a a set of agent definitions and object types, and a particular configuration. An outline of an algorithm for constructing a "snapshot overall net" is presented below. The resulting net is a high-level Petri net to which known analysis techniques can be applied.

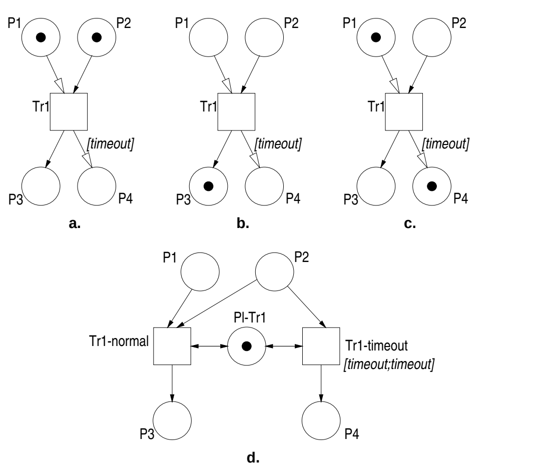

Figure 1: A timeout transition and its expansion into TPN components.

3.1. A high-level PN formalism

The formalism we propose resembles Coloured Petri Nets (CPNs) [5], and we adopt ideas from Time Petri nets (TPNs) [9] for dealing with timeouts of Objective Linda's operations. One of the strengths of our formalism is that it can be translated into CPNs and TPNs. Hence, all the features we introduce are "syntactic sugar"; they all translate to features of commonly known Petri Nets.

Timeout transitions. First, we introduce a timeout mechanism for transitions for timeout exception handling. It allows us to model the timeouts of Objective Linda's operations. A timeout transition has a new kind of input arcs, called *non-deterministic input arcs*, and special output arcs, called *timeout output arcs*.

Semantically, a timeout transition is **enabled** if tokens are available from all but the "non-deterministic input places", and it **can fire** (normally) when it has been enabled for less than *timeout* time and if tokens are available from the "non-deterministic input places". In this case, no tokens are shifted towards the output places which are connected through timeout output arcs. A timeout transition **must** fire if it has been enabled for *timeout* time without having fired. During this timeout exception firing, tokens are withdrawn from all but the "non-deterministic input places" and tokens are shifted only through the timeout output arcs. Graphically, non-deterministic input arcs are represented by an arc with an open arrow head; timeout output arcs are additionally annotated with a timeout value in square brackets.

Consider the example in Fig. 1.a. It shows a timeout transition with one normal ($P2$) and one non-deterministic input place ($P1$), and one normal ($P3$) and one timeout output place ($P4$). The transition is enabled, since a token is available in P2. It is also firable because there is a token in P1, too. Firing normally for this transition means retrieving tokens from both of its input places, and shifting a token towards the normal output arc. This results in the marking as shown in Fig. 1.b. However, if the transition has been enabled for *timeout* time without having fired, it is forced to fire. In this case, tokens are retrieved from $P2$ only, and a token is shifted towards $P4$, the output-arc with the timeout annotation. This results in the marking as shown in Fig. 1.c.

A Petri net with timeout transitions can easily be translated into a TPN. TPNs associate with each transition t_i two times $t_{i,1}$ and $t_{i,2}$. A transition t_i can fire only if it has been enabled for at least $t_{i,1}$ time and it must fire before $t_{i,2}$ once it is enabled.

The translation of the transition from Fig. 1.a into TPN components is shown in Fig. 1.d. *Tr1* is replaced by two TPN transitions, *Tr1-normal* and *Tr1-timeout*. *Tr1-normal* has the same input places as *Tr1*; *Tr1-timeout* is only adjacent to *Tr1*'s classical input places. Both share an additional, initially marked place *Pl-Tr1*. The time annotation of *Tr1-normal* is $[0, \infty]$, and is therefore left out. If the timeout transition *Tr1* is enabled, *Tr1-timeout* is enabled. If there is also a token in *P1*, *Tr1-normal* is enabled, too. When *Tr1-normal* fires, tokens are shifted towards its output places, *P3* in the example. If *Tr1* has been enabled for *timeout* time without firing, *Tr1-timeout* has been too. Therefore, *Tr1-timeout* will fire. The additional place *Pl-Tr1* ensures that if the transitions are multiply enabled (i.e. could fire more than once consecutively), the timer associated with the *Tr1-timeout* transition is reset, such that it cannot fire more than once consecutively.

Tokens are objects. A characteristic of high-level nets is that tokens are not anonymous entities merely indicating state by their presence at a place, but they are structured; they contain information. E.g. in CPNs, tokens can have colours, i.e. values.

In our approach, tokens are Objective Linda objects, as described in Sect. 2. Contrary to CPNs and Predicate/Transition nets [6], where the values of tokens are used directly in transition predicates or in transition actions, tokens in our net formalism are approached through their interfaces.

Multiset annotations. Some high-level nets allow arcs to be annotated such that transition firings consume or produce multisets of tokens. CPNs and Interval Timed CPNs [15] allow arc expressions that yield

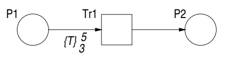

Figure 2: A multiset arc annotation.

multisets of tokens. How these expressions should look like is not further specified for these formalisms. The arc expressions that we require allow a non-deterministic choice between several multisets by only demanding a minimal and a maximal number of tokens. Arc annotations look like $\{E : T\}_{min}^{max}$, where E is an expression yielding an object of type T, indicating that the enabledness of the corresponding transition depends on the availability at the corresponding place of at least *min* tokens of type T that match the expression E, and when the transition fires, at least *min* and at most *max* tokens are withdrawn from the place. The expression can be as short as a variable, denoting that any object of type T matches, or it can be a piece of code returning an object of type T. Multiset arc annotations allow us to provide an adequate model for the Objective Linda operations which use multisets of objects. For simplicity reasons, we allow variable names as arc annotations in cases where single objects are moved along an arc.

An example is sketched in Fig. 2. Transition *Tr1* consumes either three, four or five tokens of type T from place *P1*. Because suitable multiset annotations are already part of the CPN definition, we need no explicit translation to a lower–level formalism, here.

Named places. The last feature we introduce is *named places*. The idea is the following. A named place in a net has a variable associated with it. The content of this variable is the *identifier* of the *actual place* that the named place represents. As a result of transition firings, the content of these variables may be changed (e.g. with an assignment operation), which allows another actual place to be represented by the named place. The intention is to model object space places as named places. Since within one agent an object space is represented by variables containing a "reference" to an actual object space that can be changed at run time (e.g. by an *attach* operation), this kind of flexibility is necessary.

Named places allow a hidden form of dynamicity in the net structure. Changing the content of a named place variable means changing the actual input or output place for transitions that are adjacent to the named place. Hence, corresponding arcs are no longer a relation between places and transitions, but rather a relation between place variables and transitions.

Named places may seem a subversive feature, yet again there is a simple high-level net equivalent. We present an algorithm outline for the translation of this feature, illustrated through an example (see Fig. 3). First, join all named places into one place (*NamedPlaces*), and provide a separate place per used named place variable (here: *Pl1*, *Pl2* and *Pl3*). The type of the tokens of these new places is a special type, *PlaceId*. The purpose of the introduction of these new places is two-fold. First, a token in such a place represents the variable and actually contains the reference to the named place as its value. Second, it is used to protect an agent from inconsistencies due to concurrent (re)assignments to the named place variables.

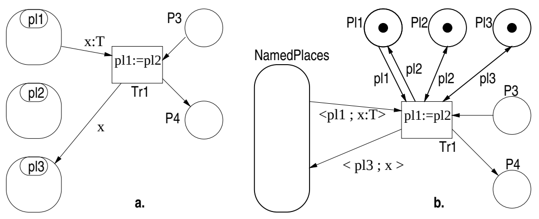

Figure 3: Named places.

Then, each annotation of an arc that is connected to a named place is altered such that each expression denoting a token, e.g. x, is replaced by an expression denoting a token <*place-id;x*>, where *place-id* is an expression yielding an identifier of a place, e.g. the variable *pl1*. Finally, each transition that is adjacent to a named place and/or manipulates one or more place variables is connected through a bidirectional arc with the new places representing the place variables (such as *Pl1*, *Pl2* and *Pl3* in the example).

3.2. Modelling Objective Linda operations

Objective Linda's operations are modelled as transitions in the agent net. The extended features proposed above make modelling these operations fairly easy. Here, we present a formalism for Petri net semantics of Objective Linda's operations called PNSOL. Transition templates for each of the operations are presented in Fig. 4, each showing a fragment of an agent net.

Fig. 4.a shows the PNSOL representation of *in*. It is a *timeout transition* with a *non-deterministic arc* with *multiset annotation*, and with a *named* input *place*. If the state of the agent is such that the transition is enabled, it can fire normally if appropriate objects are available in *objsp* within *timeout* time. In any case, if the transition did not fire within the time bound, it is forced to fire, retrieving tokens from the input places $P_{i,1} \ldots P_{i,n}$, and putting a token into the timeout output place P_{to}. Note that if appropriate objects are available in *obj-sp* within *timeout* time, the transition

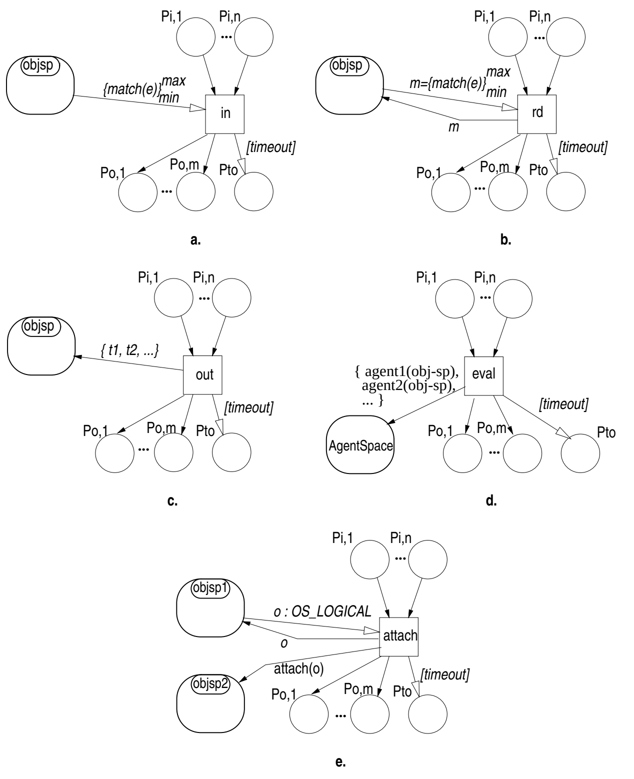

Figure 4: Modelling Objective Linda operations.

can, but is not forced to fire.

Similarly, templates for *rd* and *out* are presented in Fig. 4.b and 4.c. The *rd* transition is a *timeout transition* with a *non-deterministic arc* with *multiset annotation*, and with a *named* input *place*. If the transition is enabled, it can fire if appropriate objects can be retrieved from the object space. If it fires, such objects are withdrawn from the object space place and replaced immediately.

The *out* transition is a *timeout transition* with an output arc with *multiset annotation*, and with a *named* output *place*. Since there is no non-deterministic input arc, firability coincides with enabledness for this operation: if the transition is enabled, it either fires normally, within *timeout* time, or the timeout exception mechanism makes it fire if it has not fired after being enabled for *timeout* time.

The template for *eval* is presented in Fig. 4.d. If the transition fires within the time bound, a multiset m of agents is shifted towards the *AgentSpace* place, representing the activation of new agents. In this case, every new agent $A(o) \in m$ will be assigned o as its initial *context* object space.

The *attach* operation (Fig. 4.e) is a *rd* operation with a multiset containing a single object o of a specific type, *OS_Logical*. Additionally, there is an output arc

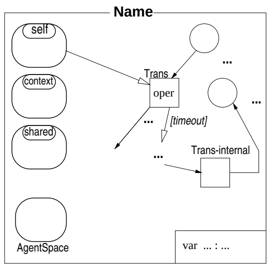

Figure 5: Generic model of an agent class.

to the name variable of the object space to attach to where the value of *attach(o)* is stored which is defined to return a reference to an object space matching o.

3.3. Agents with Petri net specifications

We now introduce the notion of *agent class*, as opposed to *agent type*, as to be defined in Sect. 5. An *agent class* is defined as an abstract implementation of a set of similar agents. In contrast, *agent types* are specified based only on the observable behaviour of agents. Analogous to sequential object–oriented programming, it is possible to implement a certain agent type by several, different agent classes. Fig. 5 shows a generic model of an agent class which consists of:

- **class name**, identifying the name of a collection of identical agents;
- **object space places**, the net representation of the object spaces;
- the **agent space place**, the net representation of the agent space;
- **variable declarations**, to be used in arc and transition annotations;
- and the **agent net**, the high-level net representing the (autonomous) behaviour of the agent.

3.4. Constructing an overall net

We now provide an outline of an algorithm that takes a description of a system in terms of Objective Linda agents with net specifications, and yields one overall net, modelling a snapshot of the open system.

The algorithm is divided into two phases. The first phase translates each agent definition separately into a high-level net. The second phase merges these nets into one overall net. The global idea is that all agents of a class are represented by one Petri net, where the state of each individual agent is identified by tokens identifying the agent of which they are part of the agent net. Separate nets then communicate through shared places (modelling the object spaces).

Phase 1: Translation of agent definitions. The goal of the first phase is to have a representation of all agent instances by one net per class of agents. This net does no longer contain any of the new features we introduced in Sect. 3.1. and 3.2. The first phase consists of four steps:

1. The first step is to expand the transitions representing Objective Linda operations, as explained in Sect. 3.1. and 3.2. At this time, per class of agents **Agent**, a new object type is defined, *CreationRequest***Agent**. The output tokens of the *eval* transition are replaced by pairs <*CreationRequest***Agent** *; context*> where *context* refers to the object space the new agent is to be created in.
2. The second step is to translate the named places, i.e. the object space places of the agent nets. All object space places are replaced by one place (per agent class). Arcs that were adjacent to an object space place are replaced by an arc that is adjacent to the joint object space place. The arc annotation is changed and a new place per object space variable (of type *PlaceId*) is introduced, as explained in Sect. 3.1.
3. All arcs that are not adjacent to object space places change annotations: each annotation T denoting an object, is replaced by an annotation <*agent_id;T*>. Arcs having implicit annotations, denoting anonymous tokens, are replaced by an annotation representing tokens as <*agent_id*>.
4. Finally, the initial marking of the agent is withdrawn. A new transition, called *InitNewAgent***Agent**, is added. Its set of output places equals the set of places that are marked initially. One firing of this transition creates a new agent identifier, and then forwards tokens, which must correspond to the establishment of the initial marking of a new agent.

Phase 2: Merge agent definitions. When all the agent definitions have been expanded separately, they can be joined as to constitute one overall net. This is achieved correctly by merging each object space place of all different classes of agents into one overall object space place, and by merging all AgentSpace places into one overall AgentSpace place. Furthermore, each transition *InitNewAgent***Agent** gets one input arc, connecting it with the AgentSpace place, and is annotated by <*CreationRequest***Agent***;context*>. This ensures that the initialization transition of a new agent can fire if (and only if) a creation request for an agent of class **Agent** has been issued.

4. Restaurant of dining philosophers

We now illustrate Petri net descriptions of Objective Linda agents on example. We partly present a solution to the problem of *The Restaurant of Dining Philosophers* taken from [2], which is an extension of Dijkstra's classical problem to open systems. In the restaurant, there is the table with n seats. Between each two seats, there is exactly one chopstick on the table. Philosophers sitting down need two chopsticks (the ones left and right to their seat) in order to eat rice from the bowl on the table. So far, this is the classical synchronization problem. Additionally, philosophers can enter the restaurant, wait for a free seat, and leave the table after eating.

We extend the problem a bit. There are two kinds of agents in the system: a waiter who runs the restaurant with the name *The Philo and the Fork* and philosophers who come in and eat. Arriving philosophers try to enter the restaurant (*attach* to its object space), wait for a free seat and then try to atomically grab the two chopsticks next to their seat. After eating, they put the (now dirty) chopsticks back on the table, stand up, and leave the restaurant. The waiter opens the door (by putting a *logical* for his restaurant which is of a special subtype of *OS_Logical* denoting restaurants into his *context* space). Then he sets up the table by putting chairs and chopsticks into his *self* object space. Until it is time to close the restaurant, the waiter looks for dirty chopsticks which he cleans and puts back on the table. A complete solution can be found in [7].

We present a *Philosopher* agent description using our Petri net formalism in Fig. 6. The agent net is rather simple, modelling that a philosopher attaches to the shared "restaurant" object space first. And if he cannot do so within 10 minutes, the agent terminates. After successfully entering the restaurant, the philosopher takes a seat and two chopsticks. While he is eating, the sticks get dirty. Finally, he puts the seat and the sticks back, and reaches the *Terminated* state. This net illustrates how our PN description can be used to specify agent behaviour that can easily be understood due to its visual representation.

5. Agent types

We introduce a notion of types and subtypes for active objects based on the *observable behaviour* of agents. The observable behaviour of an agent is defined by the effects of its actions. In an Objective Linda environment, agents can only be observed by other agents through objects they store in object spaces.

Before we can define *agent types* and *subtypes*, we define the notions of *computations*, *observers*, and *experiments*. Therefore, we rely on the work in [3].

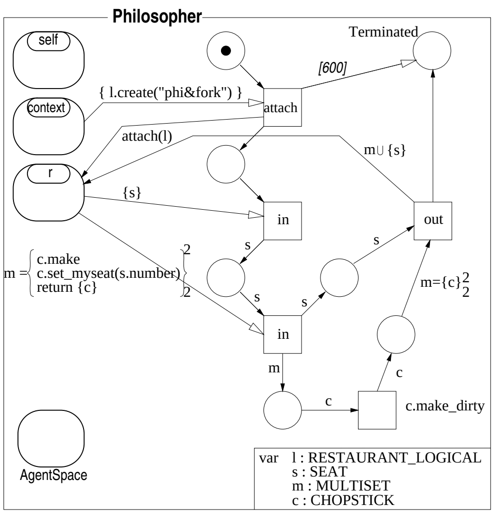

Figure 6: The description of *Philosopher* agents.

Definition: *Computation*
Consider a snapshot of a system S, for which a Petri net N_S has been constructed (as proposed in Sect. 3.). Let M_0 be the initial marking of N_S. A *computation* of the system S is a (finite) sequence of net markings $C_S = (M_0, M_1, \ldots, M_n)$, $n > 0$, such that M_{i+1} is a reachable marking starting from M_i (in PN terms: $[M_i > M_{i+1})$), for $i = 0 \ldots n-1$.

A computation C_S of S is called *maximal* iff there is no computation C''_S from M_0 such that $C''_S = C_S, C'_S$.

PNSOL has been introduced for describing agent behaviour, especially the agents' interactions with object spaces. Hence, we can use an observer as a "third-party agent" for investigating observable behaviour. We pose the following two restrictions on observer agents in order to make them useful in experiments, as defined below: (a) a test (the evaluation of an observer agent) takes finite time and observers are described by finite nets, and (b) an observer does not stop if it can proceed, which can be formalized by the notion of maximal computations.

Definition: *Observer*
An *observer O* is an Objective Linda agent that can output (but not input) an object of a distinguished type *OK* into its *context* object space.

The idea is the following: An observer agent is used for investigating the observable behaviour of another agent. A test setting is a system configuration consisting of two agents, the agent whose behaviour is being investigated, and the observer. An observer is intended to test the agent, and depending on how the agent reacts to this test, the observer may finally decide that the agent *passed* the test, and it consequently *out*'s an object of type *OK* into its *context* object space. This idea is embodied by the concept of *experiments*.

Definition: *Experiment*
Given an Objective Linda agent A and an observer O, an *experiment* E for A and O in the semantics PNSOL is a maximal computation (M_0, M_1, ..., M_n) of the system consisting of nets defining the classes of agents A and O and the *StartUp* agent S which is shown in Fig. 7.

An experiment $(M_0, M_1, \ldots M_n)$ is said to be successful if at marking M_n, the place *ObjectSpace* contains an object of type *OK*. Otherwise, the experiment is called unsuccessful.

The result of applying an observer O to an agent A with respect to semantics PNSOL is $result(O, A) \subseteq \{\mathsf{true}, \mathsf{false}\}$ defined by: $\mathsf{true} \in result(O, A)$ if there is a successful experiment for A and O. $\mathsf{false} \in result(O, A)$ if there is an unsuccessful experiment for A and O.

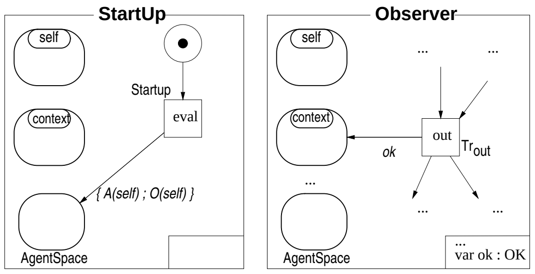

Figure 7: The startup net S of the experiments and a generic description of observer agents.

Now we can formally introduce *agent types* in terms of their observable behaviour:

Definition: *Agent Type*
An *agent type* is a set of observer agents O_T. Given an agent A and a set O_T of observer agents in semantics PNSOL, A has the agent type T (notation: $A :: T$) iff $\forall o \in O_T : \{\mathsf{true}\} = result(o, A)$

An agent A is of a type T, if it successfully passes the experiments with all observers $o \in O_T$, constituting the agent type T. A definition of subtyping relations for agent types is now straightforward.

Definition: *Agent Subtype*
An agent type S is called a *subtype* of agent type T (notation: $S :\sqsubseteq T$) iff $O_T \subseteq O_S$.

Intuitively, a type S is a subtype of T, if it is more demanding. Being of type S implies that, besides all experiments of type T, an agent has to pass some additional experiments.

Notice that $O_{\text{OIL\_Object}} = \emptyset$, because OIL_Object is the root type of the entire type system and every object in an Objective Linda configuration is of this type by definition.

One of the questions that is raised immediately here concerns *type–checking*: how can one check whether an agent A, specified by its agent net, is of a particular type T? This question can be dealt with quite easily using properties of Petri nets: Checking whether agent A is of type T means that we need to check whether for each experiment performed with the observers $o \in O_T$ an object of type *OK* is stored into the *context* object space of o. Rephrasing this in terms of the Petri net representation of the "observation system" (the overall net), results in checking whether (for each o) a token of type *OK* is put into the *ObjectSpace* place. This can be rephrased in terms of a well–known property of Petri nets: *liveness*. A transition t is called live iff for each marking M reachable from the initial marking M_0, there is a marking that is reachable from M, such that t is enabled. Consider the observer agent in Fig. 7. Its agent net contains by definition a transition Tr_{out} representing an *out* operation of an object of type *OK* on its *context* object space.

We can conclude that an agent A is of type T iff for each observer $o \in O_T$, the transition Tr_{out} in the Petri net representation of an experiment consisting of A, o and a *StartUp* agent is *live*.

6. The restaurant revisited

Let us now illustrate the definition of agent types by a concrete example, namely the type of a philosopher from the restaurant example in Sect. 4. For describing a type, one needs to provide a set of observers defining the requirements of the agent's environment, to which an agent must comply in order to be of that type. In our example, the corresponding type can be described by a single observer agent of class *PhiloObserver*, presented in Fig. 8. A *PhiloObserver* agent first establishes the setting in which a philosopher agent can be introduced, by presenting chopsticks, a seat, and the restaurant (an OS_LOGICAL). If some time later, the observer agent finds that the agent in the experimental setting produced two dirty chopsticks in their common object space, and it can remove (*in*) the agent's seat, a token of type *OK* is produced in its *context* object space. In this case, the agent is of type *PhiloType*.

Let us now consider the *Philosopher* agents pre-

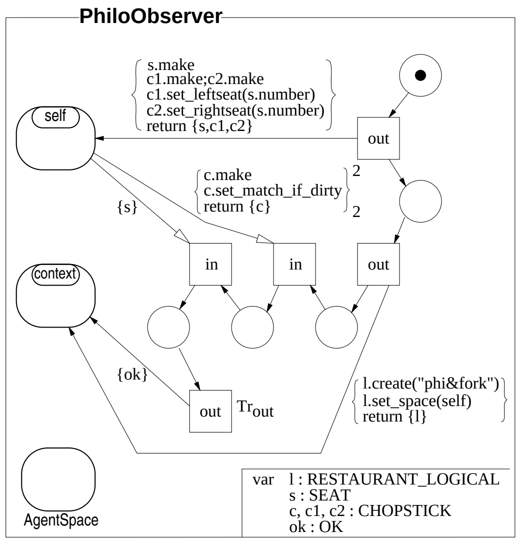

Figure 8: The observer agent for type *PhiloType*.

sented in Fig. 6. In order to check whether this kind of agents is of type *PhiloType*, an experimental setting is constructed. This setting is then translated into one high–level Petri net, as explained in Sect. 3.

The definition of types now tells us that a *Philosopher* is of type *PhiloType*, if the transition Tr_{out} (of *PhiloObserver*) is live, i.e. if there is no sequence of actions that results in a state in which Tr_{out} will never be able to fire. We can use automated tools like the one included with Design/CPN [5] in order to check this. By now, we rely on the reader's intuition for noticing that a *Philosopher* agent actually is of type *PhiloType*, since it retrieves a seat and sticks from its object space places and, after having eaten, puts two dirty chopsticks and its seat back into the restaurant object space.

Let us now think of an agent description of a particular kind of philosopher which is **not** of type *PhiloType*, called *MaliciousPhilosopher*. Agents of this kind resemble *Philosopher* agents, except that they do not put the chopsticks back after having acquired them. It is clear that in the Petri net representation of the experimental setting with an agent of this class and a *PhiloObserver* agent, the transition Tr_{out} is not live. It can not fire after the *MaliciousPhilosopher* agent acquired the chopsticks, since it nevers returns any chopsticks to the corresponding object space.

7. Related work

A large family of formalisms for modelling concurrency is based on process calculi like e.g. the π–calculus [11]. However, these formalisms lack adequate abstractions for modelling shared data spaces.

Petri net–based models are clearly superior in this respect.

Type systems (describing relations in type hierarchies) are well understood for passive objects [13]. Types specify an object's interface presuming an encapsulated object state, ensuring only *type safety* such that no "*message not understood*" errors will occur. However, this definition is too weak for types of passive objects in a concurrent environment. This observation led to more behaviour oriented type definitions and correspondingly a *"behavioural notion of subtyping"* [8]. As a natural continuation, behaviour and types of active objects have been investigated. The starting point of these research activities are active objects that communicate either by message-passing, as in Actor systems [1], or in a client/server style [12]. In the sequential case, invoking an unavailable operation indicates a program error. In a concurrent environment, however, this is an expression of synchronization conditions in which the calling object has to wait until the operation is available [10]. The work in [12] introduces a type system for active objects that is based on this principle for which the term *non–uniform service availability* was coined. In this respect, our type definition is at least as powerful.

8. Conclusion

In this work, we have introduced a formal model called PNSOL, which is based on high-level Petri nets. This formalism allows to specify behaviour of active agent objects communicating via Objective Linda's shared object spaces. The simplicity of our visual formalism has been illustrated in Sect. 4. Furthermore, because PNSOL transitions can be completely expressed in terms of Coloured Petri Nets and Time Petri Nets, theoretical results on these net types can be applied, too.

Based on PNSOL, we introduced a notion of types and a corresponding subtyping relation. Type checking of agents can be performed through liveness checking of a transition of a Petri net. Because Couloured Petri Nets, onto which we map the building blocks of our formalism, are computationally equivalent to Turing machines, liveness is in general undecidable. But with suitable restrictions to finiteness of nets and object domains, liveness can be at least automatically checked, as it is e.g. done by the Design/CPN tool [5]. Because the work in [14] reports that the introduction of time annotations like the ones we use does not prohibit feasibility of liveness checking, it sounds feasible to construct automated type checking tools for agents.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, Cambridge, Massachusetts, 1986.

[2] P. Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.

[3] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the Operational Semantics of a Coordination Language. In *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pp. 77-106. Springer, 1995.

[4] D. Gelernter. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80-112, 1985.

[5] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Springer, 1992.

[6] K. Jensen and G. Rozenberg, Eds. *High-level Petri Nets.* Springer, 1991.

[7] T. Kielmann. Designing a Coordination Model for Open Systems. In *Coordination Languages and Models*, LNCS 1061, pp. 267-284, Springer, 1996.

[8] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811-1841, 1994.

[9] P. Merlin. *A Study of Recoverability of Computing Systems.* PhD dissertation, Dept. of Information and Computer Science, University of California, Irvine, California, 1974.

[10] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Commun. ACM*, 36(9):56-80, 1993.

[11] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. LFCS Report Series ECS-LFCS-89-85 and ECS-LFCS-89-86, LFCS, Dept. of Computer Science, University of Edinburgh, Edinburgh, UK, 1989.

[12] O. Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, Eds., *Object-Oriented Software Composition*, chap. 4, pp. 99-121. Prentice Hall, 1995.

[13] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley, 1994.

[14] L. Popova. On Time Petri Nets. *Journal of Information Processing and Cybernetics*, 27(4):227-244, 1991.

[15] W. M. P. van der Aalst, K. M. van Hee, and P. A. C. Verkoulen. Interval Timed Coloured Petri Nets and their Analysis. In *Advances in Petri Nets '93*, LNCS 691, pp. 453-472. Springer, 1993.

[16] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proc. ECOOP'88*, LNCS 322, pp. 55-77. Springer, 1988.