Behaviour Specification of Parallel Active Objects

Tom Holvoet

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

Thilo Kielmann (Correspondent Author)

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

Abstract

The development of parallel programs is primarily concerned with application speed. This has led to the development of parallel applications in which software engineering aspects play only subordinate roles. In order to increase software quality in parallel applications, we motivate the construction of parallel programs by composing *active objects* which interact by means of an object–oriented coordination model.

This paper presents a formalism for specifying the behaviour of parallel active objects and a corresponding notion of *behavioural types* which can be used for verifying whether certain active objects conform to a specified behaviour. Our approach is based on high–level Petri nets which enable (besides other benefits) automated analysis, in particular for automated type checking of active objects. We illustrate the usefulness of our approach by presenting reusable active objects for a manager/worker architecture. Their correct interaction is shown by automated checking of behavioural types.

Keywords: coordination, Objective Linda, behaviour specification, high-level Petri nets, reusable components

1 Introduction

The rapid growth of interconnected high–performance computing resources has initiated various efforts for developing software methods aimed at utilizing the available computational power in an effective manner. However, most current methods are merely ad hoc approaches based on low-level abstractions which map directly to operating system equivalents. In general, the starting point of these methods is to have different processes run on different processors where the processes use particular libraries of communication primitives. Substantially, three communication paradigms are considered: (1) message passing, used with libraries such as PVM [10] and MPI [11], which provide the basic capability to send and receive messages in different flavours in an architecture-independent manner, (2) remote procedure calls (or remote object method invocation), a communication mechanism tailored for realizing client–server applications in distributed computing environments [25,26]; and (3) distributed shared memory [29], an abstraction which supports the concept of shared memory in a multi-computer environment (networked computers without physically shared memory).

Present parallel computing environments in general do not rely on advanced software engineering principles, yet only focus on performance issues. Although performance is one of the main issues in using parallel computers, other criteria, related to *software quality* (such as structured design, software extendibility, reusability, maintainability, and so on), should not be neglected, especially since these do not necessarily contradict performance issues.

A lot can be learned from the domain of software engineering as it is applied to sequential computing. In this domain, *object-oriented* techniques are by now well established for designing and implementing large application systems of high software quality. One of their most important strengths is that these software models consist of abstractions (objects) of *real world entities*. Since this narrows the gap between software models and the systems to be modelled, more complex systems can be dealt with more thoroughly, and software models become more understandable and hence adaptable. Moreover, features of object–orientation such as inheritance and polymorphism offer support to make software systems reusable.

Basically two motivations for integrating parallelism and objects can be distinguished. First, new abstractions should allow software models to have an even closer relationship to the real world, since the real world consists of intrinsically parallel entities. And second, one tries to move parallel computing from a pure performance–oriented number crunching business to a software engineering approach to parallel programming.

Active objects seem to be the most promising approach in both of these views. Active objects unify the notions of (passive) objects and processes by attributing each active object with its own thread of control.

Although object–orientation has become a key technology for the development of large–scale applications, (sequential) object–oriented programming languages lack appropriate abstractions for modelling the communication of these parallel entities, i.e. the active objects. This leads to a severe semantic gap between object–oriented programming and low–level communication.

Coordination models [21] aim at closing this gap by providing suitable abstractions for modelling the interactions of concurrent processes. Ideally, a coordination model allows programmers of parallel systems to abstract from the underlying communication systems, and to focus on aspects of the parallel computations instead. Using a suitable, object–oriented coordination model is hence a highly promising approach for building parallel applications from concurrent active objects.

Coordination models based on generative communication (such as Linda [3] and its relatives) are the state of the art in this field. Implementation performance of Linda–like systems has been shown to be directly comparable to hand–crafted code based on lower level abstractions [4,18,20], indicating the successful applicability of coordination technology to the field of parallel computing.

Generative communication inherently *uncouples* communicating processes. A sender of a data item does not directly contact another party, which is supposed to be the receiver, but instead it produces and inserts (it "generates") an item into a data space which is shared with other active entities. On the other hand, a potential reader of a particular data item does not have to take care about it (e.g. as with rendezvous mechanisms) until it actually needs it. The reader does not even have to exist at the time of generation. The latter point leads to the other major advantage of generative communication: 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 parallel applications: uncoupled communication allows to abstract from details of underlying process configurations, especially in the case of dynamically changing configurations (like e.g. with workstation clusters [17]) in which processes may move or temporarily disappear. Due to this fact, coordination models based on generative communication are superior to message passing or trader–based schemes because these both rely on knowledge about a receiver's or server's identification.

Despite these benefits, generative coordination models are unfit for expressing the *behaviour* of parallel processes, since they focus on shared data spaces only, and not on the processes manipulating them. Hence, generative coordination models can easily model the state of shared data spaces and *single* operations on them, but it is hard to model, and in consequence to reason about *behaviour* of processes in terms of sequences of data space operations.

The intent of this work is to fill this gap by introducing plausible definitions (based on formal specifications) for process *behaviour*, characterized by sequences of operations on shared data spaces. Therefore, we rely on our coordination model Objective Linda which introduces object-orientation to the Linda paradigm. With Objective Linda, several active objects (called *agents*) can interact in a generative manner by producing and consuming objects of well–defined data types. As has been shown in a precursor of this work [12], high–level Petri nets [15] are an attractive formalism for behaviour specifications of Objective Linda processes, because they combine an easily understandable visual representation and expressive power with respect to concurrency, causality and non–determinism. A combination of ideas from Coloured Petri nets [14] and Time Petri nets [22] leads to a well–suited formal model for behaviour specification of Objective Linda's active objects. This formalism, called PNSOL, is hence also used for Petri net semantics of Objective Linda operations.

The formal representation of Objective Linda agents allows to provide a formal definition for *typing* agent behaviour. *Agent types* are based on the *observable behaviour* of agents. We define types for agents as sets of observable behaviours, and a subtyping relation based on subset inclusion, correspondingly. 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 an elementary property of Petri nets, namely transition *liveness*.

The remainder of the paper is organized as follows. In Section 2, we briefly introduce our coordination model Objective Linda, before we present the high–level PN formalism that is used to model Objective Linda's operations on object spaces in Section 3. In Section 4 we introduce the notions of *behavioural types*, *subtypes*, and corresponding *type checking* for active objects. In Section 5 we illustrate the usefulness of our approach by a thorough discussion of a typical configuration found in parallel applications, namely a manager/worker architecture for which we show how well–formed behaviour of active objects can be verified by applying type checking. Section 6 summarizes and concludes this work.

2 Objective Linda

In this section, we briefly introduce the coordination model Objective Linda which forms the basis of our work. An in–depth description of its features and design decisions can be found in [16,18]. Objective Linda is based on the principles of Linda [3] and seamlessly adds concepts of object orientation. Tuples and tuple spaces are replaced by objects and object spaces. Note that for the remainder of this paper "objects" denote passive objects unless explicitly mentioned otherwise, and active objects are often called *agents*.

Objects to be stored in object spaces are self–contained entities; their encapsulated state can only be affected through their interface operations. In Objective Linda, 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 the OIL, all types form a type hierarchy having a common ancestor called OIL_Object which defines the basic operations needed by all types. The OIL allows subtyping according to the "principle of substitutability" [32] such that an object of type S which is a subtype of T can be used whenever an object of type T is expected. We use the notation $S \preceq T$ for expressing that S is a subtype of T, for given OIL types S and T.

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 of an object 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 also respected 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 determining 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.

Passive as well as active objects are characterized by an OIL type. The type OIL_Object provides an operation called evaluate whose behaviour is redefined for each subtype denoting active objects. Similar to the match operation, the behaviour of this operation may depend on the object's state before its evaluation. In the case of an active object, its OIL type is only relevant as long as the object has not yet been activated (by invoking its evaluate operation). Once activated, an object behaves according to its implementation of evaluate and is no longer accessible via its OIL type interface.

In Linda, active tuples are treated as functions and are converted into passive tuples after termination, yielding their results. In contrast to this functional view, Objective Linda treats active objects as encapsulated and reactive agents. Active objects disappear after termination. Analogous to Linda, active objects are invisible to operations in charge of retrieving passive objects from object spaces. Processes 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 a particular object space, called context. This is the object space on which the corresponding eval operation has been performed. Additionally, every agent can dynamically create new object spaces which are initially private to their creator.

The context and additional private object spaces do not suffice for expressing all coordination problems. Therefore, Objective Linda provides a mechanism (based on generative communication) for allowing agents to attach to other, already existing object spaces. Objective Linda therefore introduces a concept which is called *object space logical*. Such *logicals* combine a "description" of an object space in terms of the application logic (namely an object of a suitable subtype of OIL_Object) with the identification of the object space to be made attachable. *Logicals* are (like other passive objects) stored in object spaces, but they are "invisible" to in and rd operations.

Logicals are implicitly created by the expose operation which combines an object and the identification of an object space and stores the combination inside an object space, hence exposing one object space inside another one. The reverse operation to expose is called hide. It removes a given *logical* object.

An agent may then attach to an additional object space by presenting a template object t to the attach operation for which a *logical* with an object matching t can be found.

Besides adapting the Linda model to object orientation, Objective Linda also provides an improved set of operations on object spaces in order to satisfy the needs of *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. A timeout mechanism is considered necessary for applications that run in a distributed computing environment, in which workstations or the communication medium is not unlikely to drop performance or even fail. Second, Linda's ability to retrieve only one object at a time from an object space is too restrictive. For example, it is impossible to non–destructively iterate over all objects of a particular kind [30]. 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. The operations 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. In the following, Objective Linda's operations on object spaces are summarized. The notation is based on a binding to the C++ language, and thus the interface of the class Object_Space is shown. In order to simplify code, default values are assigned to the min, max, and timeout parameters causing Objective Linda's operations to behave in the default case analogous to the corresponding Linda operations.

bool out (Multiset *m , double timeout = infinite_time)
: 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.

Multiset *in (OIL_Object *o , int min = 1 , int max = 1 , double timeout = infinite_time)
: Tries to remove multiple objects $o_1 \ldots o_n$ matching the template object o from the object space and returns a multiset containing them if at least min matching objects could be found within timeout seconds. In this case, the multiset contains between min and max matching objects. If min matching objects could not be found within timeout seconds, the result has a NULL value.

Multiset *rd (OIL_Object *o , int min = 1 , int max = 1 , double timeout = infinite_time)
: Tries to return clones of multiple objects $o_1 \ldots o_n$ matching the template object o and returns a multiset containing them if at least min matching objects could be found within timeout seconds. In this case, the multiset contains between min and max matching objects. If min matching objects could not be found within timeout seconds, the result has a NULL value.

bool eval (OIL_Object *o , double timeout = infinite_time)
: Tries to move the object o into the object space and starts its activity. Returns true if the operation could be completed successfully; returns false if the operation could not be completed within timeout seconds.

bool expose (OIL_Object *o , Object_Space *s , double timeout = infinite_time)
: Tries to move the object o into the object space. If this could be performed successfully, o will expose the object space s. Returns true if the operation could be completed successfully; returns false if the operation could not be completed within timeout seconds.

bool hide (OIL_Object *o , Object_Space *s , double timeout = infinite_time)
: Tries to remove an object which matches o and to which s had been assigned. Returns true if the operation could be completed successfully; returns false if the operation could not be completed within timeout seconds.

Object_Space *attach (OIL_Object *o , double timeout = infinite_time)
: Tries to get attached to an object space for which a *logical* with an object matching o can be found in the current object space. Returns a valid refer-

ence to the newly attached object space if a matching object space logical could be found within timeout seconds; otherwise the result has a NULL value.

int infinite_matches

Returns a constant value which is interpreted as infinite number of matching objects when provided as min or max parameter to in and rd.

double infinite_time

Returns a constant value which is interpreted as infinite delay when provided as timeout parameter to out, in, rd, eval, expose, hide, and attach.

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 non-determinism, 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.

In the case of open systems it is in general impossible to construct one Petri net for a dynamically changing configuration. But even in this case, it is still possible to reason 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 set of agent definitions and object types, and a particular configuration. An algorithm can be used to produce one Petri net which is behaviourally equivalent to the snapshot system. An outline of such an algorithm is presented below. The resulting net is a high-level Petri net to which known analysis techniques can be applied.

3.1 A high-level PN formalism

The formalism we propose is called PNSOL (Petri net semantics of Objective Linda). It is based on Coloured Petri Nets (CPNs) as defined in [13]. We additionally adopt ideas from Time Petri nets (TPNs) [22] for dealing with time-outs of Objective Linda's operations. One of the strengths of our formalism is that it can be completely translated into CPNs and thus can be submitted to automated analysis by Petri net analysis tools, such as Design/CPN [14]. Hence, all features we introduce are merely "syntactic sugar"; they all translate to features of commonly known Petri nets. A complete formal description of how to translate PNSOL nets into CPNs with TPN time annotations can be found in [18]. For reasons of simplicity, this paper will only present the most important features of PNSOL and their translation to well–known Petri nets in order to allow the reader to understand the syntax and dynamic behaviour of these Petri nets. It can also be shown that TPN–like time annotations can be translated to time features as they were introduced for CPNs. However, such a (rather complex) translation will not be presented here in order to keep the presentation concise and simple.

We begin our description by a brief recapitulation of basic CPN features. We will specify further details while presenting the PNSOL formalism. In general, a CPN consists of three different parts: the *net structure*, the *declarations*, and the *net inscriptions*. The net structure is a directed graph with two kinds of nodes, the set of *places* P and the set of *transitions* T, interconnected by the set of *arcs* A in such a way that each arc $a \in A$ connects two nodes of different kinds. The declarations define *types* (the originally called *colour sets*) which are used for typing *variables* and other *expressions* used in the net inscriptions. We are interested in three kinds of inscriptions, namely (a) node *names* which are of no semantic meaning, (b) *arc expressions*, and (c) *initialization expressions* for places. Both arc expressions and initialization expressions must evaluate to multisets over types which are either predefined or introduced in the declarations part of the net.

The dynamic behaviour of CPNs is defined over sequences of so–called *markings* of the net. Places may contain *tokens* which are of certain types and may hence carry arbitrarily complex information. A marking is a distribution of tokens over all places of a net. The initial marking of a net is defined by the initialization expressions of all its places.

Transitions may change markings of a net. Therefore, we have to further classify places: every arc can be described as a pair (s, d) where s is its source (the node from which the arc starts) and d is its destination (the node where the arc ends). For every transition t we call $I_t = \{p \in P : \exists(p, t) \in A\}$ the set of *input places* of t and $O_t = \{p \in P : \exists(t, p) \in A\}$ the set of *output places* of t. Correspondingly, we call the arcs connecting a transition with its input places or it output places the *input arcs* or *output arcs*, respectively.

A transition t is called *enabled* whenever the net has reached a marking M_i in which each input place $p \in I_t$ contains at least the tokens denoted by the arc expression of the arc (p, t). When t is enabled, it may *occur* in which case it removes the tokens denoted by its input arc expressions from the corresponding input places and puts tokens into its output places, corresponding to its output arc expressions, yielding a new marking M_{i+1}, which is hence called *reachable* from M_i.

Several transitions may be enabled in a marking M. In this case, the corresponding input places may either contain enough tokens such that all enabled transitions may retrieve a disjunctive subset of all available tokens, or there are too few tokens. In the first case, the transitions are called to be *concurrently enabled* and may simultaneously occur, each removing its own input tokens and producing its own output tokens. In the second case, the transitions are said to be in *conflict* with each other and cannot occur simultaneously. Instead, at most a non–conflicting subset (which results from a non–deterministic choice) of the conflicting transitions may occur. As a special case, a transition may also be concurrently enabled with itself.

Timeout transitions.

A first "extension" to CPNs is a timeout mechanism. For illustrating it, we first introduce the most simple type of tokens, called *anonymous tokens*. Anonymous tokens carry no information besides their presence or absence. We indicate anonymous tokens in places by $\bullet$ signs; empty arc expressions denote the multiset containing exactly one anonymous token.

We introduce a timeout mechanism for transitions in order to model 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 enough tokens (according to the corresponding arc expressions) are available from all but the "non-deterministic input places", and it **can occur** (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 is **forced to occur** if it has been enabled for *timeout* time without having occurred: 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.

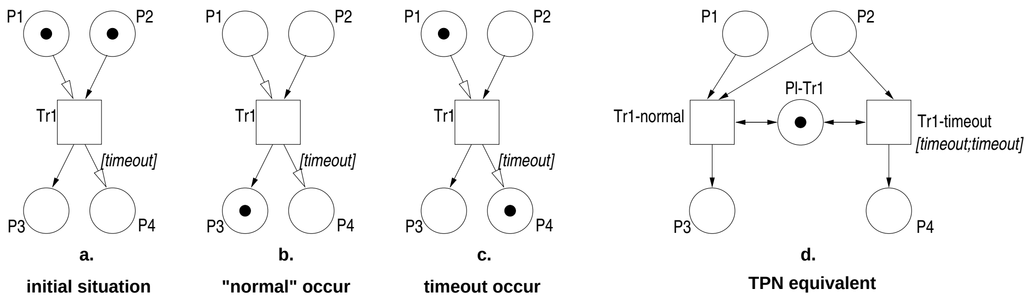

Fig. 1. A timeout transition and its expansion into TPN components.

Consider the example in Fig. 1.a. It shows a timeout transition Tr_1 with one normal ($P2$) and one non-deterministic input place ($P1$), and one normal ($P3$) and one timeout output place ($P4$). The marking M of this net has exactly one anonymous token in $P1$ and one in $P2$. The transition is enabled, since a token is available in P2. It can also occur because there is a token in P1, too. Occurring normally for this transition means retrieving tokens from all 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 occurred, it is forced to occur. In this case, the token is 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. This marking would also have been reached after *timeout* time if M would only have had a token in $P2$ but not in $P1$, which alone also had enabled Tr_1.

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 occur only if it has been enabled for at least $t_{i,1}$ time and it must occur before $t_{i,2}$ once it is enabled.

The translation of the transition from Fig. 1.a into TPN components (without the markings of $P1$ and $P2$) 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*. (We use bi–directional arcs as a graphical shorthand for two separate arcs with opposite directions and identical arc expressions.) 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* occurs, tokens are shifted towards its output places, *P3* in the example. If *Tr1* has been enabled for *timeout* time without occurring, *Tr1-timeout* has been enabled too. Therefore, *Tr1-timeout* will occur. The additional place *Pl-Tr1* ensures that if the transitions are multiply enabled (i.e. could occur more than once consecutively), the timer associated with the *Tr1-timeout* transition is reset, such that it cannot occur more than once consecutively. To illustrate this more clearly, consider the case in which *P1* initially contains one, and *P2* two tokens, and assume *timeout* to have a value indicating five seconds. Assume that after four seconds, the *Tr1-normal* occurs. If the place *Pl-Tr1* would not be considered, the transition *Tr1-timeout* would still be enabled for four seconds, and could fire at the fifth second. The place *Pl-Tr1* prohibits this by disabling *Tr1-timeout* for a very short time, and thus ensures that *Tr1-timeout* can only occur after being enabled for another five seconds. Because transitions may occur concurrently with each other, every timeout transition occurs independently from each other without any need for synchronized times.

Tokens are objects.

A characteristic of high-level Petri nets is that tokens are not anonymous entities merely indicating state by their presence at a place, but they are structured; they contain information. In our approach, tokens are primarily Objective Linda objects, as described in Sect. 2. Therefore, we add the type OIL_Object together with all its subtypes to the set of valid types for tokens to be stored in places.

Multiset annotations.

CPNs (among other formalisms) allow arcs to be annotated such that transition occurrences consume or produce multisets of tokens. We hence only need to develop a syntax for multiset expressions suitable for our needs. The arc expressions 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 $\{E\}_{min}^{max}$, where E is an expression yielding an object of a type $T \preceq$ OIL_Object (written $E : T$), 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 occurs, 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* (consisting of operations defined on objects declared in the *declarations* section of the net) that returns 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.

Formally speaking, a multiset m over a non–empty set S, is a function $m \in [S \rightarrow \mathbb{N}]$. The non–negative integer $m(s) \in \mathbb{N}$ denotes the number of appearances of the element s in the multiset m. By S_{MS} we denote the set of all multisets over S. The non–negative integers $\{m(s) : s \in S\}$ are called the coefficients of the multiset m, and $m(s)$ is called the coefficient of s. Hence, $\{E\}_{min}^{max}$ with $E : T \preceq$ OIL_Object denotes a multiset $m \in T_{MS}$ where $m(\mathrm{match}(E)) \in [\mathrm{min}, \mathrm{max}]$ is the coefficient of objects matching E. The predicate $\mathrm{match}(E)$ is defined as the match operation of the corresponding OIL type T.

For simplicity reasons, we also allow an alternative syntax for denoting explicitly enumerated multisets, namely $\{e_1, e_2, \ldots, e_n\}$, where $m(s) = 1$, if $s \in \{e_1, e_2, \ldots, e_n\}$, and $m(s) = 0$, otherwise. As special cases, $\{e\}$ denotes a multiset containing exactly one object e, and $\{e\}_1^1$ denotes a multiset containing exactly one object *matching* e.

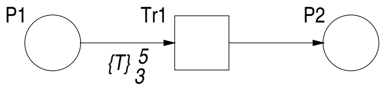

Fig. 2. A multiset arc annotation.

An example is sketched in Fig. 2. Transition *Tr1* consumes either three, four or five tokens of type T from place *P1*. In particular, if place *P1* contains at least five tokens of type T, the choice whether three, four or five tokens are withdrawn is non-deterministic. 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 to be introduced are *named places*. The idea is the following. A named place in a net is a place which has a variable representing an associated name. The content of this variable is the *identifier* of the *actual place* that the named place represents. As a result of transition occurrings, the content of these variables may be changed, which allows another actual place to be represented by the named place. Graphically, transitions reading or modifying name variables are drawn with arcs adjacent to them, as it can be seen in Figure 3.a.

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.

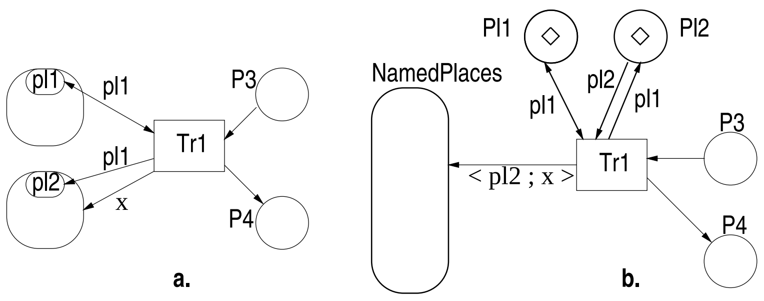

Fig. 3. Named places.

Named places may seem to break the soundness of the proposed nets. Yet again there is a simple high–level net equivalent. The translation of named places to high–level net features can be performed by the following algorithm. This is illustrated by an example in Fig. 3.

(1) First, join all named places into one place (*NamedPlaces*), and provide a separate place per used named place variable (here: *Pl1* and *Pl2*). The type of the tokens of these new places is a special type, *PlaceId*. Graphically, *PlaceId*'s are represented by the ◇ sign. 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.
(2) Then, each annotation of an arc that is connected to a named place is altered such that each expression denoting a token 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 *pl2*. 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* and *Pl2* in the example).

3.2 Modelling Objective Linda operations

In this section, we provide a semantics for Objective Linda operations using fragments of PNSOL nets. Objective Linda's operations are modelled as transitions in the agent net. The extended features proposed above make modelling these operations fairly easy. Petri net fragments for each of the operations are presented in Fig. 4, each showing a fragment of an agent net.

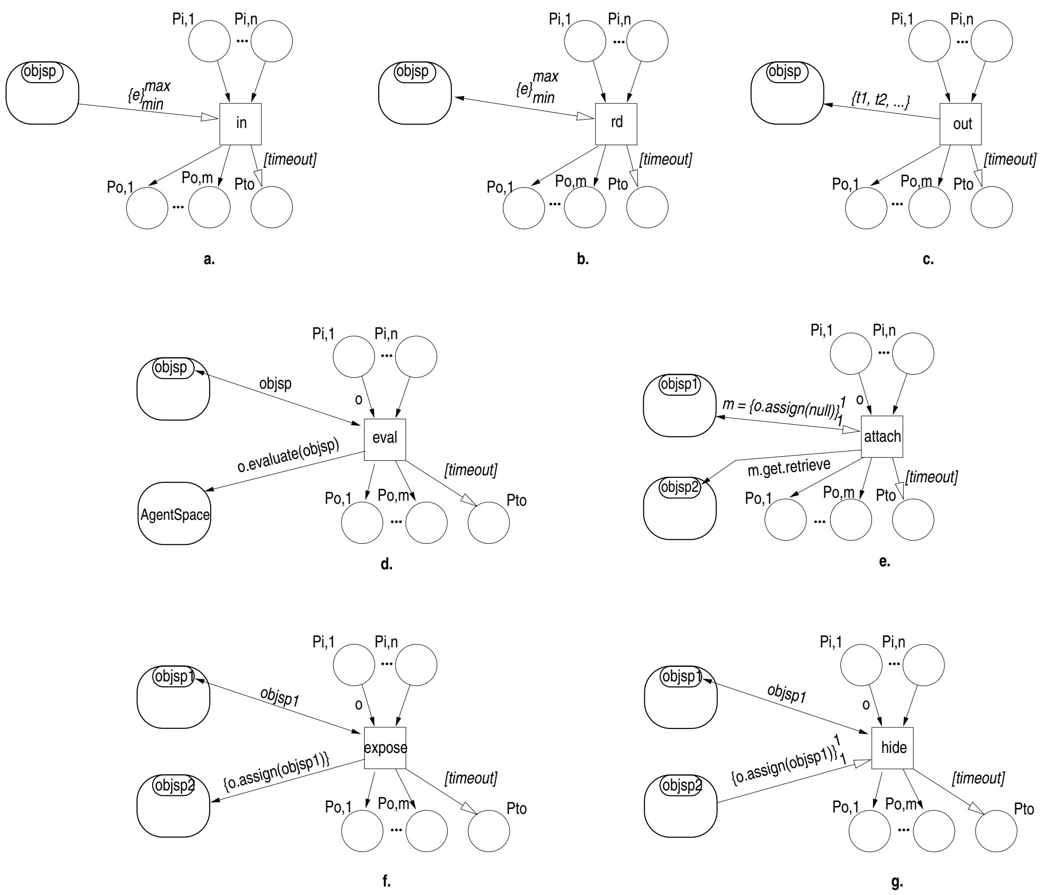

Fig. 4. Modelling Objective Linda operations.

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 (depending on $P_{i,1} \ldots P_{i,n}$ only), it can occur normally if appropriate objects are available in *objsp* within *timeout* time. In any case, if the transition did not occur within the time bound, it is forced to occur, retrieving tokens from the input places $P_{i,1} \ldots P_{i,n}$, and putting a token into the timeout output place P_{to}. This new state should represent the state in which the agent starts handling the timeout exception, i.e. similar to the timeout exception handling if the in operation would yield an empty multiset. Note that if appropriate objects are available in *objsp* within *timeout* time, the transition may, but is not forced to occur.

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 occur if appropriate objects can be retrieved from the object space. If it occurs, such objects are withdrawn from the object space place and replaced immediately (atomically with their withdrawal). As mentioned earlier, the bi–directional arc used for this operation is a graphical shortcut for two separate arcs with opposite directions and identical arc expressions.

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, the possibility to occur coincides with enabledness for this operation: if the transition is enabled, it either occurs normally, within *timeout* time, or the timeout exception mechanism makes it occur if it has not occurred after being enabled for *timeout* time.

The template for *eval* is presented in Fig. 4.d. If the transition occurs within the time bound, an agent *o* is shifted towards the *AgentSpace* place, representing its activation. In this case, the new agent is assigned objsp as its initial *context* object space. The bidirectional arc between the transition and the name variable of *objsp* denotes that the *PlaceId* token of the object space is used (withdrawn and replaced) for assigning a *context* object space to the newly created agent.

The *attach* operation (Fig. 4.e) is a *rd* operation with a multiset containing a single object *o*. Additionally, there is an output arc to the name variable of the object space to attach to where the respective PlaceId is stored. Here, the function *assign* $\in$ [OIL_Object $\times$ PlaceId $\rightarrow$ Logical] assigns a PlaceId to an object, yielding a *logical*. Analogously, *retrieve* $\in$ [Logical $\rightarrow$ PlaceId] retrieves a PlaceId from a *logical*. Finally, *get* $\in$ [Multiset $\rightarrow$ OIL_Object] retrieves an object from a multiset. Fig. 5 illustrates the expansion of the *attach* operation to lower level CPN/TPN components, analogous to Fig. 3.

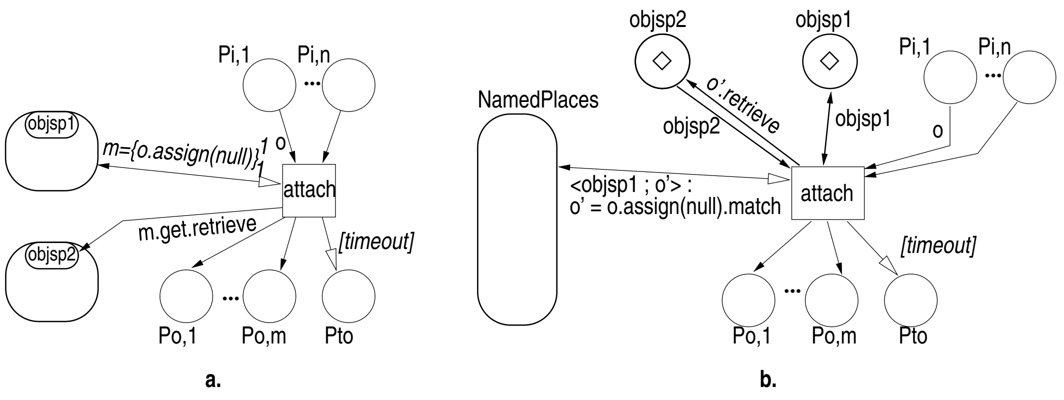

Fig. 5. The expansion of attach to CPN/TPN components.

Fig. 4.f shows the template for expose which comes closest to out except that it stores a *logical* combined from the object *o* and the PlaceId objsp1 in objsp2. Analogously, Fig. 4.g shows the template for hide which comes closest to in except that a *logical* matching the combination of the object *o* and the PlaceId objsp1 is withdrawn from objsp2.

We now introduce the notion of *agent class*, as opposed to *agent type*, as to be defined in Sect. 4. 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. 6 shows a generic model of an agent class which consists of:

- **class name**, which identifies the name of a collection of identical agents.
- **object space places**, a particular set of named places which correspond to the object spaces.
- the **agent space place**, one particular place in the agent net which represents the agent space.
- the **agent net**, the high-level net representing the (autonomous) behaviour of the agent. Similar to CPNs, agent net transitions can be annotated with *transition code segments*. Code segments are portions of sequential code for transforming the transition input tokens into the respective output tokens. Basically, code segments are functions of the transition input tokens, which do not allow side effects on objects (tokens) other than the ones that submitted to the transition occurrence.
- **declarations**, an enumeration of variable and function declarations that are used within the agent net. It should be emphasized that these declarations are not data representations for the agents of this class. Rather, they are only used in arc and transition annotations of the agent net.

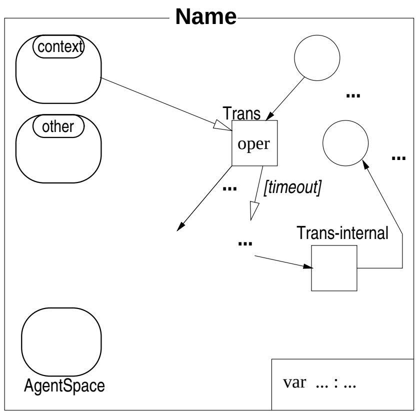

Fig. 6. Generic model of an agent class.

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 PNSOL net specifications, and yields one overall net, modelling a snapshot of a configuration.

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 one class are represented by one Petri net, where the state of each individual agent is identified within this Petri net by tokens identifying the agent of which they are part of. 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 <*Creation-Request***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, namely 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 occurrence 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 occur if (and only if) a creation request for an agent of class **Agent** has been issued.

4 Agent types

Type systems (describing relations in type hierarchies) are well understood for passive objects [27]. 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"* [19]. As a natural continuation, behaviour and types of active objects are being 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 [24]. 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 [23]. The work in [24] 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 section, we will 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 [6] which itself is based on the notion of *testing equivalence* [7].

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 $(M_0,\ M_1,\ \ldots,\ M_n)$ 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 $\mathit{result}(O,A) \subseteq \{\mathsf{true}, \mathsf{false}\}$ defined by: $\mathsf{true} \in \mathit{result}(O,A)$ if there is a successful experiment for A and O. $\mathsf{false} \in \mathit{result}(O,A)$ if there is an unsuccessful experiment for A and O.

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

Definition: *Agent Type*

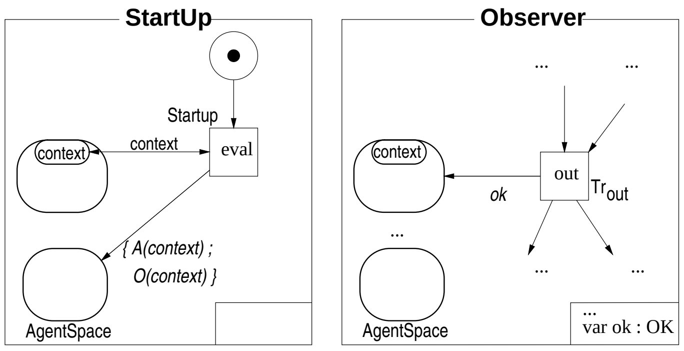

Fig. 7. The startup net S of the experiments and a generic description of observer agents.

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 $\forall o \in O_T : \{\mathsf{true}\} = \mathit{result}(o, A)$

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.

The above definitions not only provide us with a plausible definition of types and subtypes of agents, they also allow to talk about types and subtypes of *compositions* of agents. The agent A as well as the observer O might be a *concurrent composition* of agents $A_1, A_2, \ldots, A_n$ resp. observers $O_1, O_2, \ldots O_m$. A concurrent composition of agents is defined by the overall net composed out of the agent nets of the concurrently operating agents. In the case of a composed agent, experiment results concern the type of the composition as a whole. In the case of a composed observer, experiment results provide information on how an agent A behaves in the presence of multiple observers. This comes close to the notion of a *concurrent client* as introduced in [24].

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*.

Because Coloured 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 [14]. Additionally, the work in [28] reports that the introduction of time annotations like the ones we use does not prohibit feasibility of liveness checking. In the following section, we will illustrate the usefulness of PNSOL by an example in which the type–conformance of given active objects can be shown by Design/CPN.

5 Example: A Manager/Worker Architecture

The goal of this section is to illustrate the applicability and usefulness of our approach. We consider a common structure of parallel programs, namely a manager/worker architecture. For this architecture we identify generic components for manager and workers which can be instantiated and (re–)used later on by parallel applications.

This example aims to show that successful cooperation between manager and workers can be specified by *Agent Types*. By re–using given components which have been shown to cooperate successfully (by type checking of their agent types), application–specific instantiations of these components will also cooperate successfully. Hence, application programmers may rely on correct coordination–level components and are able to focus their efforts on application–specific problems. We give a succinct description of a parallel ray-tracing application that reuses the presented manager/worker architecture.

It is a common situation in parallel programming to have a specific manager process divide a given problem into smaller tasks and distribute these tasks among available worker processes. While workers repeatedly process such tasks and return corresponding results to the manager process, the managerial task is much more complex. The manager not only has to operate on the application level by providing task units and later combining the received results to the overall result of the application. It also has to perform coordination–level tasks like assigning tasks to workers and terminating workers.

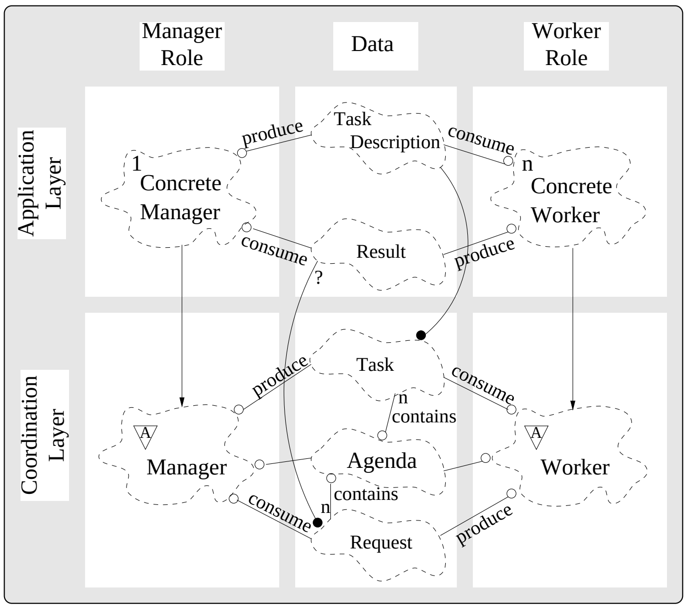

Fig. 8. Components in the Manager/Worker Architecture.

Although both levels of managerial tasks are independent of each other, they are typically intermixed in existing applications. They are hardly made explicit, but instead implicitly performed by the communication operations of manager and workers. Therefore, we follow the approach presented in [9] to provide clearly defined abstractions for both levels in the form of *reusable coordination components*.

The intent of these components is to decouple coordination–level issues such as task assignment strategies and worker termination from application–level issues such as task creation, task computation, and result combination. The Manager component is responsible for providing and assigning task units and for collecting results. The Worker component is responsible for acquiring and executing task units and for transmitting computed results to the manager.

The intent of the abstract Manager and Worker components is to provide all necessary functionality for the coordination level. Concrete and hence application–specific instantiations of managers and workers can then be built from our abstract components by means of inheritance. Using PNSOL, it can be shown that our Manager and Worker components cooperate successfully. Hence, correctness of a parallel application based on our components only depends on the correct implementation of the worker function $w \in [\text{task} \rightarrow \text{result}]$ and the manager function $m \in [\text{result}_{MS} \rightarrow \text{global\_result}]$ for computing the results of single tasks and for their combination to the overall result of the application, respectively. Both w and m are independent from coordination issues and are hence much easier to implement than equivalent functionalities directly based on operations of a coordination model.

The interactions between the Manager and Worker components are illustrated by the Booch diagram [2] shown in Figure 8. In these diagrams, dashed clouds indicate classes; solid triangles marked with an "*A*" denote abstract classes; a solid undirected edge with a hollow circle at one end indicates a "*uses*" relation between two classes. Furthermore, directed edges indicate inheritance relationships between classes; and a solid circle illustrates a composition relationship between two classes. Figure 8 is generally divided into two layers, one for coordination aspects and one for application aspects. Furthermore, the diagram is orthogonally divided into the manager role, the worker role, and the data exchanged between both, namely objects and object spaces.

The Agenda is the central shared data structure (the object space) through which the abstract components of the coordination layer (namely Manager and Workers) communicate. The Task and Request objects exchanged in this layer are primarily used as containers for their application–specific counterparts, Task Description and Result. By being subtypes of OIL_Object, objects of both types can be "transported" as parts of Task and Request objects via the coordination layer in a transparent way, namely without knowledge about these application–specific types in that layer. Finally, Concrete Manager and Concrete Worker are instantiated by inheriting from Manager and Worker, respectively, while providing suitable implementations for their application–specific methods like the above mentioned manager function m and the worker function w which we now can refine to be defined as $w \in [\text{task\_description} \rightarrow \text{result}]$.

5.2 Interactions between Manager and Worker

Although a manager/worker architecture seems to be trivial at first glance, there is a broad spectrum of possible interaction protocols between Manager and Workers [8]. In the simplest case, the Manager just out's all Task objects while the Workers in them as long as there are still Task objects available. They then out Request objects containing the corresponding results back to the Agenda. This type of collaboration is only useful in situations in which

(a) workers can be safely set up *after* the agenda has been filled (because their reasoning is based on the absence of objects and hence they cannot distinguish between “no more tasks” and “tasks not yet available”), (b) the entire set of tasks is known at program startup, e.g. there are no iterations with processing steps of the manager in between (also due to “reasoning on the absence of objects”), (c) the assignment of tasks to particular workers is not important (e.g. it will have no significant impact on performance), and (d) the entire set of tasks can be stored in the agenda without violating memory restrictions.

In order to deal with problems (a) and (b), Workers can also join a “*worker group*” maintained by the Manager at program startup and leave it after having received a “stop task”. In order to deal with problems (c) and (d), workers can *request* tasks being assigned to them from the manager which e.g. allows to perform load–balancing issues.

Further refinements of manager/worker interactions may stem from the need to decouple the number of Result objects being created by the workers from the number of tasks created by the manager, which might become necessary in order to overlap processing between manager and workers or for implementing adaptive schemes in which workers may act as “sub–managers” for parts of the tasks assigned to them.

Finally, creation and termination of Workers may also be withdrawn from the Manager’s functionality and may be performed by a specific *scheduler* or *resource–manager* component in order to perform computations in dynamically changing configuration like workstation clusters [17].

Because of diversity and complexity of the interaction protocols between Manager and Workers, it is a highly desirable goal to provide generic components which provide all necessary functionality in the coordination layer to be re–used by parallel applications. In the following, we will show how *Agent Types* as introduced in Section 4 and corresponding type checking can contribute to implementing such components with reliable behaviour.

5.3 Constructing Reliable Components

We now illustrate how such reliable components can be developed using PNSOL specifications and checking of *Agent Types*. In order to keep our presentation simple, we restrict ourselves to the case in which the manager out’s n task objects and in turn in’s n result objects. The corresponding workers repeatedly in a task object, compute the corresponding result, and out a result object, until the agenda object space contains no more task objects. Also, we assume that workers are eval’uated separately and will not stop processing tasks until all tasks have been computed. Agent classes for manager and workers based on the PNSOL formalism are shown in Figure 9. The Manager agent class specification includes a simple agent net. It contains one object space, called agenda, which represents the object space that Manager agents will share with Worker agents. From the initial state of a Manager agent, the transition labelled out is enabled. If this transition occurs, the agent constructs n Task objects, $t_1, \ldots, t_n$, and stores these into the agenda object space. Thereafter, the agent is in a state in which the transition labelled in can occur when n Request objects can be retrieved from the agenda object space. When this transition has occurred, the Manager agent has completed its activities.

The Worker agent net models the autonomous behaviour of Worker agents as follows. It also relies on one object space, called agenda, through which it will be composed with a Manager agent. From the initial state of the Worker agent net, the transition labelled in can occur if it can retrieve a Task object from the agenda object space. Thereafter, the agent may execute this task by performing particular computations based on information from the Task object. The results from these computations are encapsulated in a Request object, which is finally restored in the agenda object space through the transition labelled out. This action re-establishes the initial state of the agent, which can repeat its behaviour.

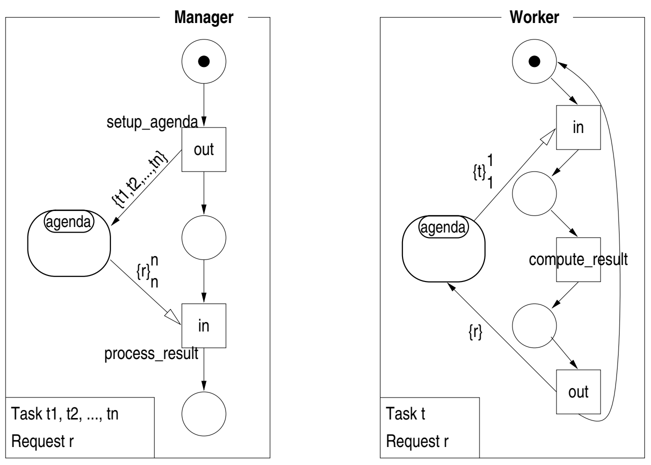

Fig. 9. Agent Classes for Manager and Workers.

Again for simplicity reasons, we restrict ourselves to further investigating the worker's behaviour. Analogous investigations for the manager component are straightforward and therefore left out in this presentation. For verifying the behaviour of Worker agents, we construct an observer agent that simulates the manager by storing n Task objects and consuming n Request objects afterwards. After having consumed n Request objects, the observer agent verifies whether it may consume further Requests or even (unprocessed) Task objects. In this case, the transition labelled Red. (indicating "redundant" objects) will occur, invalidating the liveness property of the OK transition. Figure 10 shows an agent class WorkerObserver.

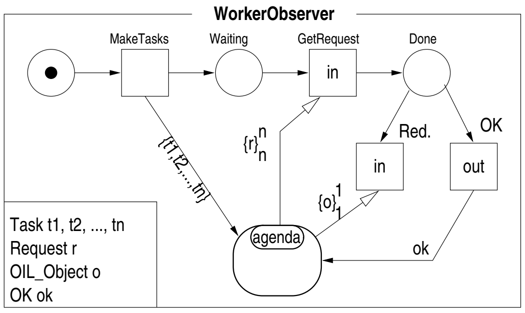

Fig. 10. The Agent Class WorkerObserver.

We can now specify an experiment E_w, in which a Worker agent, and a Worker-Observer are initialized by a StartUp net, analogous to the one shown in Figure 7. Hence, we can define the type for Worker agents: WorkerType = {WorkerObserver}. Because *Agent Types* are defined on simple agents as well as on concurrent compositions of agents, it is straightforward to verify whether several concurrently operating Worker agents, seen as an inherently concurrent agent, comply to WorkerType, too.

According to the transformation algorithm outlined in Section 3.4, the agent class descriptions for Worker and WorkerObserver can be translated to "plain" Coloured Petri Nets. The resulting nets are merged into one CPN, represented in Figure 11. In this figure, one can identify the CPN representations of the different agent nets. They are indicated by dashed boxes (which do not have any semantic significance in this Petri net). One recognizes the StartUp part (left upper corner), Worker part (left lower corner) and WorkerObserver part (right lower corner). The two central places in this CPN represent the global object space (annotated with "Agenda") and the virtual AgentSpace. Transitions *InitNewAgent***Worker** and *InitNewAgent***WorkerObserver** represent the creation of the respective agents. Figure 12 contains the corresponding CPN colour definitions and declarations: definitions of colour type representations of anonymous, Request, and Task objects, tuple and "CreationRequest" colour definitions which result from the algorithm to construct the overall net, and the OIL_Object colour which corresponds to the union of all sorts of passive objects, and a number of variable declarations required in this CPN. The initial marking of this CPN consists of one anonymous token in the Init-StartUp place. This marking enables the transition CreateExperimentActors, which represents the eval operation of a Worker and a WorkerObserver agent, and hence puts two appropriate tokens in the AgentSpace place. At that time, the respective *InitNewAgent* transitions can occur and the experiment begins.

The net in Figure 11 has directly been fed into the Design/CPN analysis tool in order to perform automated type checking. This can be achieved by querying whether certain properties concerning places or transitions hold. In our case, the liveness property of the transition labelled OK has been queried

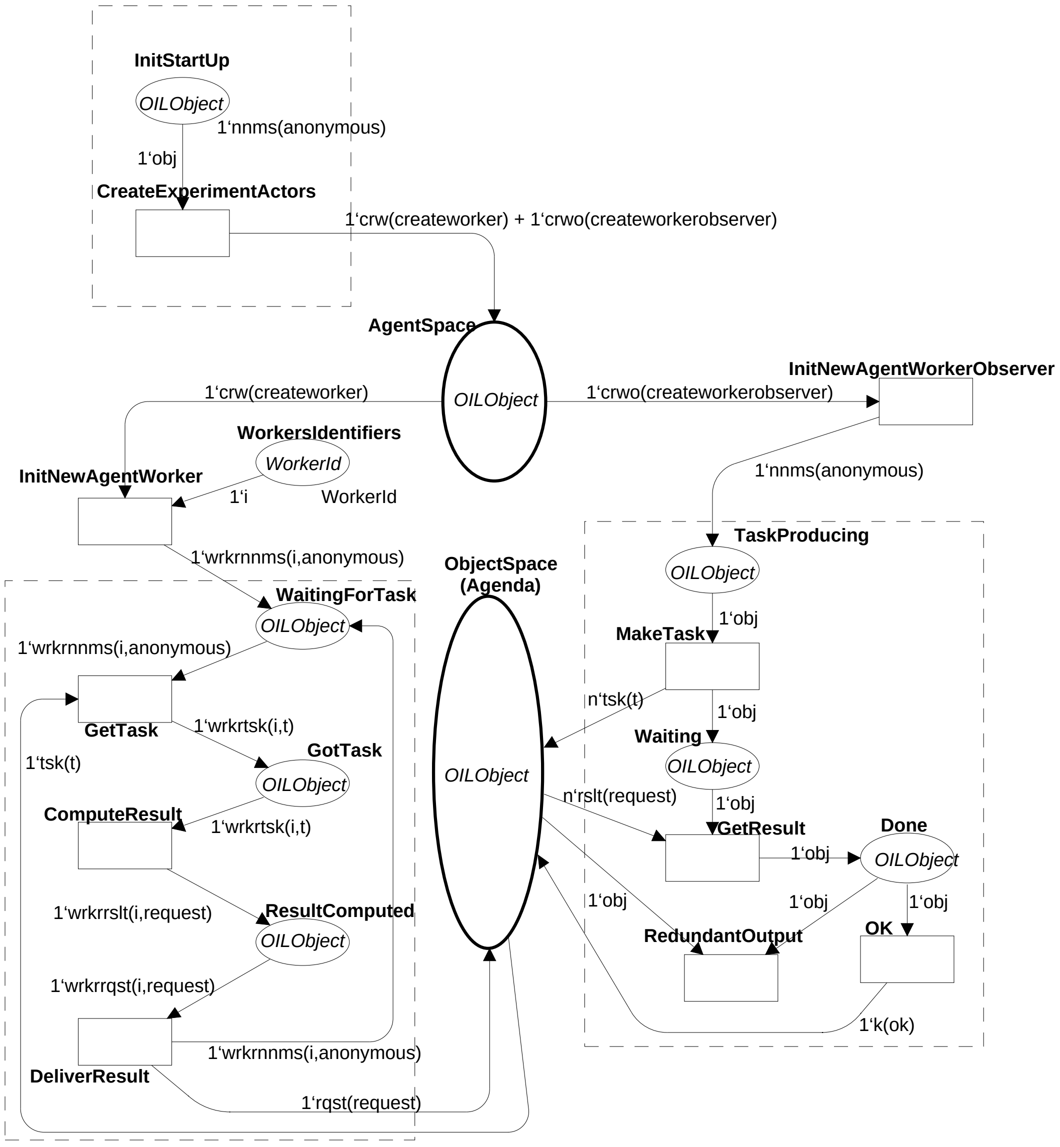

Fig. 11. Overall net for analyzing Worker behaviour: the net structure.

which has been answered by Design/CPN by `true`, indicating that the OK transition in fact is live.

5.4 Consequences

As has been shown above, the OK transition in the WorkerObserver agent class description has been (automatically) shown to be live by Design/CPN. Based thereupon, we can conclude that Worker agents successfully pass the experiment with WorkerObserver and hence comply to the agent type WorkerType.

By analogously defining agent types for Manager agents and correspondingly for Workers and Managers with more sophisticated interaction protocols, a

```
color Anonymous = with anonymous;
color Request = with request;
color Task = with task;
color WorkerId = int;

color WorkerAnonymous = product WorkerId * Anonymous;
color WorkerTask = product WorkerId * Task;
color WorkerRequest = product WorkerId * Request;

color CreationRequestWorker = with createworker;
color CreationRequestWorkerObserver = with createworkerobserver;

color OK = with ok;

color PassiveObject = union nnms:Anonymous + wrkrnnms:WorkerAnonymous + tsk:Task + rqst:
Request + wrkrd:WorkerId + wrkrtsk:WorkerTask + wrkrrqst:WorkerRequest + crw:
CreationRequestWorker + crwo:CreationRequestWorkerObserver + k:OK declare ms;

var t : Task;
var i : WorkerId;
var obj : PassiveObject;
```

Fig. 12. Overall net for analyzing Worker behaviour: declarations.

complete set of reliably communicating components can be built like it has been shown above for the simplest possible case.

Automated type checking for agent types is not the only benefit of specifying agent behaviour by PNSOL, the corresponding agent types, and their CPN equivalents. Additionally, tools like Design/CPN are also capable to perform more empirical analyses like simulations of given systems. Such simulations are quite useful for understanding runtime behaviour of given agent nets, especially in cases in which erroneous behaviour has to be detected and to be understood in order to correct it.

5.5 Parallel Raytracing

In this section, we illustrate the use of the abstract Manager and Worker architecture by a parallel raytracing program [8]. Raytracing is a widely used method for generating realistically looking images on a computer. The input to a raytracing algorithm is a *scene* – the description of the geometry of 3D objects and the definition of the objects' materials, the lights, and the imaginary camera. The output is the image of the scene, i.e. the colour of each pixel of the image (also called a *frame*), as seen by the defined camera.

A straightforward way to perform raytracing in parallel is to distribute the pixels of the image to different processors and colour them independently. In this approach, each processor has the complete scene description available in order to compute the colour of any frame pixel. This approach is typically realized by a manager/worker scheme.

In the following, we present a simple scheme using our manager/worker architecture for parallel raytracing, illustrating the reusability of our components.

We provide application–specific classes for manager and worker agents through inheritance of the corresponding abstract classes. These subclasses employ the same agent nets as their abstract superclasses; only transition code annotations are redefined. We use a C++ notation for describing the implementation of the transition code.

A straightforward approach is that the manager divides the image into lines and puts each line as a task to be processed into the agenda. The workers retrieve tasks from the agenda, process them and put the computed results back into the agenda. Worker agents repeat this behaviour until all tasks have been processed.

We assume two subclasses of OIL_Object, range and image, denoting the range of lines in the frame to be computed and the corresponding image lines. The classes Simple_Manager and Simple_Worker inherit from the abstract classes Manager and Worker, and implement code segments of the respective superclass agent nets in order to correspond to the application–specific behaviour of managers and workers in the raytracing program.

A C++ implementation of Simple_Manager and Simple_Worker is shown in Figures 13 and 14, respectively. The operation setup_agenda of the Simple_Manager class represents the code segment that is associated with the setup_agenda transition of the agent net. The operation parameters reflect the transition input tokens, the return value corresponds to the output tokens. The setup_agenda operation constructs a multiset of tasks by separating the image in disjoint ranges. The operation process_result represents the code segment of the process_result transition, which takes a multiset of results and computes the final result of the application.

The Simple_Worker class mainly defines an implementation for the code segment that corresponds to the compute_result transition of the abstract Worker agent net. The operation takes one parallel raytracing task as its input parameter, and yields a computed result.

As can be seen from the code fragments, both application–specific classes only have to implement the operations specific to their own problem domain whereas the coordination–related code for the interaction between manager and workers has been separated into dedicated reusable components while their successful interaction can be shown by (automated) analysis of Petri nets.

```
class Simple_Manager : public Manager{
  private:
    int lines, lines_received;
  protected:
    virtual Multiset *setup_agenda(void) {
      Multiset *m;
      for (int i = 0 ; i < lines ; i++ )
        m->put(*new range(i,i));
      return m;
    }
    virtual void process_result(Multiset *m) {
      // store image lines to file
    }
  public:
    Simple_Manager (int size_of_image){
      lines = size_of_image; lines_received = 0;
    }
};
```

Fig. 13. A C++ class Simple_Manager.

```
class Simple_Worker : public Worker{
  protected:
    virtual OIL_Object *compute_result(OIL_Object* my_range){
      return image lines(my_range); // compute image of line range
}
};
```

Fig. 14. A C++ class Simple_Worker.

6 Conclusions

In this work, we motivated the construction of parallel applications by composing active objects that communicate via Objective Linda, an object–oriented coordination model. In order to enable reasoning about the behaviour of active objects, we briefly outlined Objective Linda itself and introduced a formal semantics for it (PNSOL), which is based on high–level Petri nets. We have chosen to use high–level Petri nets because they provide an intuitively understandable, visual formalism that is highly expressive with respect to concurrency, causality, and non–determinism. Furthermore, high–level Petri nets can rely on a solid theoretical foundation and hence allow formal reasoning, as we employ it for verifying behaviour of active objects.

We showed how PNSOL building blocks translate to features of Coloured Petri Nets (CPNs) and Time Petri Nets (TPNs). Besides providing a formal semantics for Objective Linda by means of PNSOL, we also introduced the notion of *agent nets* which represent the behaviour of a single active object, called *agent* in Objective Linda terms. In order to enable reasoning on the behaviour of several interacting agents, we outlined an algorithm for transforming all related agent nets into a single overall net on which reasoning can be performed.

Based on PNSOL, we introduced a notion of *agent types* where so–called *observer* agents verify whether or not a given agent successfully passes a given *experiment*. We define agent types by sets of successfully passed experiments and a subtype relation on subset inclusion, accordingly. Type checking can be performed by testing the *liveness* of certain transitions inside the observer agent nets.

We illustrated the usefulness of PNSOL and the corresponding agent types by the example of a manager/worker software architecture. We presented generic manager and worker components which we have shown to cooperate successfully by an automated liveness analysis using the Design/CPN tool. As we outlined, these generic manager and worker components can be easily instantiated in order to form concrete managers and workers for use in parallel applications. Hence, every application which is built by reusing our generic components can rely on an interaction mechanism between manager and workers which has been shown to operate correctly.

The properties shown for the manager/worker example are quite encouraging. It is hence an important goal for future work to build a toolkit of coordination abstractions consisting of reliable, reusable components with already shown correct interaction.

References

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

[2] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[3] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, Cambridge, Massachusetts, 1990.

[4] N. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message–passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[5] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, Cesena, Italy, 1996. Springer. Proc. COORDINATION'96.

[6] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the Operational Semantics of a Coordination Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science, pages 77–106. Springer, 1995.

[7] R. De Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[8] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters. In Sprague Jr. [31], pages 596–605.

[9] B. Freisleben and T. Kielmann. Coordination Patterns for Parallel Computing. In D. Garlan and D. L. Métayer, editors, *Coordination Languages and Models*, number 1282 in Lecture Notes in Computer Science, pages 414–417, Berlin, Germany, 1997. Springer. Proc. COORDINATION'97.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[12] T. Holvoet and T. Kielmann. Behaviour Specification of Active Objects in Open Generative Communication Environments. In Sprague Jr. [31], pages 349 – 358.

[13] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets*, number 483 in Lecture Notes in Computer Science, pages 342–416. Springer, 1990.

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

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

[16] T. Kielmann. Designing a Coordination Model for Open Systems. In Ciancarini and Hankin [5], pages 267 – 284. Proc. COORDINATION'96.

[17] T. Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. In *Proc. ICCI'96, 8th International Conference of Computing and Information*, Waterloo, Ontario, Canada, June 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).

[18] T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming.* PhD dissertation, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.

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

[20] S. Maffeis. System Support for Distributed Computing. In W. Gentzsch and U. Harms, editors, *High–Performance Computing and Networking, Proc. HPCN Europe 1994*, number 797 in Lecture Notes in Computer Science, pages 293–301, Munich, Germany, 1994. Springer.

[21] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994.

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

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

[24] O. Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object–Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[25] Object Management Group. The Common Object Request Broker: Architecture and Specification. (draft) edition 2.0, 1995.

[26] Open Software Foundation. Introduction to OSF DCE. Open Software Foundation, Cambridge, USA, 1992.

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

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

[29] J. Protić, M. Tomašević, and V. Milutinović. A Survey of Distributed Shared Memory Systems. In *Proc. 28th Hawaii International Conference on System Sciences*, pages 74–84, Maui, HA, 1995.

[30] A. Rowstron and A. Wood. Solving the Linda Multiple rd Problem. In Ciancarini and Hankin [5], pages 357–367. Proc. COORDINATION'96.

[31] R. H. Sprague Jr., editor. *Proc. of the Thirtieth Annual Hawaii International Conference on System Sciences*, Wailea, Hawai'i, USA, 1997. IEEE.

[32] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *Proc. ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 55–77, Oslo, Norway, 1988. Springer.