# Coordinating Active Agents in Open Systems

**Thilo Kielmann**
University of Siegen,
Dept. of Electrical Engineering and Computer Science
Hölderlinstr. 3, D–57068 Siegen, Germany
kielmann@informatik.uni-siegen.de

**Abstract**

The field of agent oriented programming (AOP) recently emerged from its object–oriented roots. Whereas the benefits of object–orientation nowadays are widely used in AOP, modeling of communication in multi–agent systems (MAS) still lacks adequate abstractions. Currently considered approaches like RM–ODP, CORBA, or languages like Java introduce object–orientation to inter–agent communication while unfortunately providing only rather low–level communication abstractions.

This work introduces Objective Linda, a coordination model especially designed for the needs of communication between active agents in open systems. We present how Objective Linda can be used as a suitable platform for MAS and illustrate this on an example of a traffic scenario.

## 1 Introduction

The notion of *Agent Oriented Programming* (AOP) [Sho93] has been coined as a specializing evolvement of actor systems which today manifest the de–facto model for systems of communicating active objects [Agh86]. In AOP, objects are specialized to agents while object state is treated as the agent's *mental state* and where communication between agents is typically considered in terms of *speech act theory* [Sea69] like *information, request, offer* etc.

In [Bur95], AOP has been shown to be especially beneficial for use in open systems. In that work, open systems are characterized by the following requirements on agents: *continuous availability* (persisting individual operations), *extensibility* (coping with dynamic configurations), *decentral control*, *asynchrony*, *inconsistent information* (lack of globally consistent states), and *arms–length relationships* (coping with only local and hence incomplete system knowledge).

In this work, we will outline our notion of open systems which comes close to the above–mentioned definitions. We then argue that mere object–orientation is not sufficient for adequately modeling the interoperation of active agents. Instead, so–called *coordination models* should be used to express and constrain object interactions. Then we present our coordination model Objective Linda and illustrate its usefulness for the implementation of multi–agent systems (MAS). Hence, our contribution is a coordination platform on top of which intelligent agents in the sense of AOP can be implemented.

The paper is structured as follows: In Sect. 2, we will clarify our notions of coordination and of open systems and we will briefly evaluate commonly used approaches. In Sect. 3, we present our coordination model Objective Linda. Its usefulness for implementing active agents will be shown by an example in Sect. 4.

# 2 Communication Platforms for Open Systems

*Open systems* are systems in which new active entities ("objects", "agents", or "actors") may dynamically join and leave, i.e. evolving, self–organizing systems of interacting agents [Agh86, Cia90]. It is widely accepted that open systems are composed of software components which are *encapsulated* and *reactive* [Weg93]. Components are called encapsulated if they have an interface that hides their implementation from clients; they are called reactive if their lifetime is longer than the processing of their atomic interactions (e.g. messages). This definition of components directly leads to object–based design because objects are by their very nature encapsulated and reactive entities.

A fundamental property of open systems is their ability to cope with incremental adaptability. In this perspective, encapsulation captures spatial incrementality by controlled propagation of local state changes and reactiveness enables temporal evolution by incrementally executing interactions. Another fundamental property of open systems is their inherent heterogeneity. The openness for new components implies openness for so–far unknown kinds of components yielding systems which are composed of various kinds of hard- and software.

Programming open systems is primarily concerned with the *coordination* of concurrently operating active entities. Coordination involves the management of the communication between these entities. Coordination models based on *generative communication* are considered the most prospective approaches to this research domain. Generative communication, as initially introduced in [Gel85], is based on a shared data space, sometimes also called a *blackboard*, in which data items can be stored ("generated") and retrieved later on.

This kind of communication inherently uncouples communicating agents: a potential reader of some 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 storing. The latter point leads to the other major advantage of generative communication: 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: uncoupled communication enables to cope with dynamically changing configurations in which agents move or temporarily disappear. Anonymous communication allows to communicate with unknown agents. Hence it allows communication with incomplete knowledge about the system configuration which is a crucial demand of open systems. 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.

A related important notion is the one of *open distributed systems*. It is defined in the upcoming ISO reference model of open distributed processing (RM–ODP) [ISO95]. In the RM–ODP definition, *distributed systems* have to cope with *remoteness* of components, with *concurrency*, the *lack of a global state*, and *asynchrony* of state changes. In addition, *open distributed systems* are characterized by *heterogeneity* in all parts of the involved systems, *autonomy* of various management or control authorities and organizational entities, *evolution* of the system configuration, and *mobility* of programs and data.

The RM–ODP model which conceptually provides the basis for commercially available systems uses object–based modeling too; also because of the principal object properties of

encapsulation and reactiveness. RM–ODP focuses on interaction between objects based on the client/server architecture: "They (objects) embody ideas of services offered by an object to its environments, that is, to other objects." [ISO95] In RM–ODP, coordination between objects takes place via centralized instances, so called *traders* [ISO94], which are repositories of service type definitions, used to identify offered and requested services.

Presumably the most prominent commercial system for open, object–based systems is the Common Object Request Broker Architecture (CORBA) [Obj93]. Its central component, the Object Request Broker (ORB) acts as a trader in the sense of RM–ODP. Like other traders, the ORB provides references to server objects which in case of dynamically changing configurations may quickly turn into void ("dangling") references causing problems in open configurations. Today, client/server architectures are seen as the current intermediate step on the way from mainframe–oriented to collaborative (peer–to–peer) computing [Lew95]. Nevertheless, service–oriented communication is an important paradigm for open distributed systems [Adl95] and must hence be captured by coordination models. But because client/server communication is restricted to the exchange of request/reply pairs, other communication forms like e.g. for group communication can not be modeled adequately. Hence, coordination models for open systems need to be more general in their applicability.

As an alternative approach to object interoperability by trader–based schemes, the programming language *Java* [AG96] recently attracted broad attention. Java is a fully–featured object–oriented programming language with concurrency abstractions based on a thread concept. It's major benefit is a tight coupling to the World Wide Web (WWW) for which a mechanism for dynamic software loading across physically distributed and potentially heterogeneous systems has been developed. This mechanism, together with Java's interpreted code execution, enables the development of mobile code which is a crucial feature for the vision of autonomous software agents roaming around the Internet.

Unfortunately, the communication abstractions provided by the Java runtime system are rather low–level, like datagrams, sockets, and a wrapper to access documents in the Web. There are no suitable abstractions for expressing behaviour and interactions of active agents. This is where generative coordination models come into play, as we already outlined above.

# 3 Objective Linda

We will now introduce the coordination model Objective Linda which we use as the basis of this work. A complete description can be found in [Kie96a]. Objective Linda is based on the foundations of Linda and has been designed in order to meet the requirements of open systems. We start with its language–independent object model, then outline how multiple object spaces can be handled cleanly in open systems, and complete by presenting the set of operations on object spaces.

## 3.1 Objective Linda's Object Model

Since the goal is to model open systems, a language–independent object model is necessary. In Objective Linda, objects to be stored in *object spaces* are self–contained entities;

their interface operations only affect their encapsulated object state. The 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 the 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" [WZ88] such that an object of type $S$ which is a subtype of $T$ can be used whenever an object of type $T$ is expected.

### 3.1.1 Object Matching in Objective Linda.

Objective Linda's object model treats objects as encapsulated entities which can only be accessed via their interface routines defined by the corresponding type. Consequently, object matching (the process of identifying objects to be retrieved from object spaces) in Objective Linda is based on object *types* and the *predicates* defined by type interfaces. A potential reader has to specify the type of object it wishes to obtain from an object space and additionally a predicate from the type interface which selects the objects of a given type matching a specific request. Because OIL's subtype relations provide types which can be used as replacements for their supertypes, object matching will also consider objects of subtypes of the requested type.

Denoting the type of objects a reader tries to obtain can be achieved by passing an object as a parameter to the operation. The type of this object can be easily deduced. Passing a predicate is a little bit more difficult. In Objective Linda, 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* in the following.

### 3.1.2 Evaluating Active Objects.

According to Linda's *eval* operation, we will call the activity of an agent the *evaluation* of an active object. In favour of a homogeneous model, passive as well as active objects are characterized by their OIL type. The mechanism used to specify this activity is similar to object matching: the type *OIL_object* provides an operation called *evaluate* whose behaviour is redefined by every type of objects that will become active. Similar to the *match* operation, the behaviour of this operation may depend on the object's state before its evaluation.

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. Hence, the *eval* operation activates objects which simply disappear after termination. Analogous to Linda, active objects are invisible to operations in charge of retrieving passive objects from object spaces. Hence, the behaviour of agents can only be observed by monitoring the passive objects they produce and consume.

## 3.2 Multiple Object Spaces in Objective Linda

Configurations in Objective Linda consist of two kinds of objects: (active as well as passive) OIL objects, and object spaces. Active objects have, from the moment of their activation on, access to two 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 providing hierarchical abstractions for sub–configurations.

The restriction to exactly the *context* and *self* object spaces is not powerful enough in order to generally express coordination problems. Therefore, we need a mechanism 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. This is necessary because object spaces must by their very nature be 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 (based on the generative communication mechanism) which 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 an object space can be found by matching properties of *logical* objects. These properties can of course be customized to application needs by 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 which also contains 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 $a$.

## 3.3 Operations on Object Spaces

Besides the adaptation of the Linda model to object-orientation, Objective Linda also provides an improved set of operations on object spaces. Improvements concern on one hand the blocking semantics of operations which can be customized by a timeout parameter. On the other hand, operations take multisets of objects instead of single tuples as in Linda.

### 3.3.1 Operation Blocking.

The operations in the original Linda model have been designed without consideration of openness. As a consequence, the blocking operations for putting an object into an object space (*out*), for consuming an object (*in*), and reading an object (*rd*) assume unrestricted access to the data space and may hence block infinitely in case of open systems where access to an object space may fail due to transient problems.

Furthermore, semantics of the non–blocking versions of *in* and *rd* (*inp* and *rdp*) imply access to a data space as a whole: these operations are defined to immediately return, in-

dicating a failure when there is no object matching a given request. Their semantics must be slightly modified for open systems: operation failure of *inp* and *rdp* should indicate "no such object could be found (in the moment)", reflecting the fact that synchronization based on the absence of a certain object is impossible in open systems.

In order to allow customization of agent behaviour between immediately failing and infinitely blocking, Objective Linda introduces a *timeout* parameter to all of its operations that determines how long an operation should block before a failure will be reported. It can vary between zero and a value indicating an infinite delay.

### 3.3.2 Multisets of Objects.

Linda's ability to retrieve only one object at a time from an object space is simple and elegant, but unfortunately too restrictive. It is e.g. impossible to non–destructively iterate over all objects of a certain kind [BWA94]. 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.

While multisets of objects are necessary for *in* and *rd*, they have no substantial benefits for *out* or *eval*. But for consistency and simplicity reasons, we use multisets of objects for all operations on object spaces.

### 3.3.3 Operation Specification.

We can now specify Objective Linda's operations on object spaces. We use a binding to the C++ language as notation.

**bool out ( MULTISET *m , double timeout )**

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, int max, double timeout )**

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 at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, the result has a *NULL* value.

**MULTISET *rd ( OIL_OBJECT *o, int min, int max, double timeout )**

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 at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, the result has a *NULL* value.

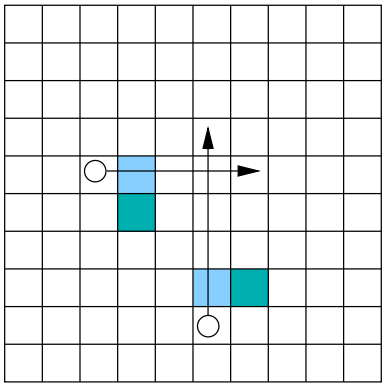

Figure 1: Two cars with intersecting paths.

**bool eval ( MULTISET *m, double timeout )**
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.

**OBJECT_SPACE *attach ( OS_LOGICAL o, double timeout)**
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 the result has a *NULL* value.

**int infinite_matches**
Returns a constant value which will be 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 will be interpreted as infinite delay when provided as *timeout* parameter to *out*, *in*, *rd*, and *eval*.

# 4 An Example: Collision Avoidance

We will now illustrate the suitability of Objective Linda as a platform for implementing multi–agent systems by an example. The scenario described below models the problem of collision avoidance in the traffic domain and has been inspired by the work in [vM92]. Our example is of course overly simplified because our intention is to present Objective Linda as a platform for MAS, rather than intelligent behaviour of the agents themselves.

In our example, agents are concerned with steering cars. Cars drive in a (cyclic) grid which is shown in Fig. 1. Because cars may drive in the four directions *up, down, left,* and *right*, the driveways occasionally intersect. It is the agents' task to avoid collisions in such cases.

We propose a solution in which the agents communicate via an object space. Every agent puts an object of type *Position* into the object space which carries the agent's

```
class Position : public OIL_OBJECT{
private:    bool match_position;   // switching the matching mode
public:     int x,y;               // the grid position
            int car;               // the car's id
bool match(OIL_OBJECT* obj){
    if (match_position)
        return ( (((Position*)obj)->x == x) &&
                 (((Position*)obj)->y == y)    );
    else
        return ((Position*)obj)->car == car;
};
set_match_position() { match_position = true; }
set_match_car()      { match_position = false; }
};
```

Figure 2: Source code of a C++ class *Position*

identification as well as its position on the grid. When changing its position, an agent consumes (using *in*) the *Position* object with its own identification and replaces it by a new one with the updated position.

Whenever an agent wants to make the next step in its desired direction, it has to check first whether it can do so safely. For this purpose, our agents follow the "right–goes–first" priority rule as it is well–known from street traffic. For this purpose, an agent first checks whether it can *rd* a *Position* object for the grid position directly in front of it, in Fig. 1 shown in light–grey colour. If there is such an object, the agent's car will not move but wait. If there is no such object, there is still the possibility of a collision in case a second car will approach on an intersecting path, as it is shown in Fig. 1. For this case, the agent checks for a *Position* object for the grid position in its right–front, in the figure shown in dark–grey colour. Again, if there is such an object, the car stops. Otherwise it moves on. This behaviour is shown as a C++ class *Car* in Fig. 3.

We claim this solution to be adequate for modeling active agents in open systems, because there is no centralized control instance and because agents operate autonomously and asynchronously, and without global knowledge about the number or kinds of cars running in the system. Consequently, agents may join or leave the system at any point of time completely on their own behalf.

Figure 2 sketches the source code of a C++ class *Position* and demonstrates how Objective Linda provides communication abstractions on an adequate level. The focal point of an Objective Linda type for passive objects is its *match* routine. *Position* objects are matched in two different ways: By car id (for updating) and by grid position (for collision avoidance). One can easily see how this can be performed: An agent creates a template *Position* object and sets (as desired) corresponding values either for the car id, or for the position. Finally, it either calls *set_match_position* or *set_match_car* in order to preselect the matching mechanism. Then, the template object can be used as a parameter for *in* or *rd* operations.

Whereas the scenario outlined so far shows the simplest of the possible cases, one can think of several extensions that can be easily supported by Objective Linda:

```
class Car : public OIL_OBJECT{
private:    int x,y;               // the grid position
            int car;               // the car's id
            direction dir;         // the direction to move
void wait(){};   // wait for an arbitrary (random) interval
void evaluate(){
     MULTISET m = new MULTISET;
     Position *p; int nx,ny,px,py;
     while (true) {
         m->put(new Position(id,x,y));
         (void)context->out(m,context->infinite_time);
         wait();
         // store next position to move to in nx and ny
         nx = ... ; ny = ... ;
         p = new Position(id,nx,ny); p->set_match_position();
         m = context->rd(p,1,1,0);
         if ( m ) { // there is a car in front of us
              delete m; delete p;
         }
         else {
              delete p;
              // store position with priority in px and py
              px = ... ; py = ... ;
              p = new Position(id,px,py); p->set_match_position();
              m = context->rd(p,1,1,0);
              if ( m ) { // there is a car with priority
                delete m; delete p;
              }
              else {     // move!
                x = nx; y = ny;
                delete p;
              }
         }
         p = new Position; p->car = id; p->set_match_car();
         m = context->in(p,1,1,context->infinite->time);
         p = m->get(); delete p;
     }
};
};
```

Figure 3: Source code of a C++ class *Car*

- A first improvement might be to enlarge the agents' *range of vision*. In order to control an area instead of single grid points, one might easily extend *Position*'s *match* routine to match positions in given intervals. Hence, an agent might retrieve information on all other cars in a certain area within one multiset returned by the *rd* operation.

- One might also consider scenarios with multiple (grid) areas, each represented by a separate object space. These areas might be connected by gates, represented by object–space logicals. Hence, car agents might dynamically change their *context* object space by using Objective Linda's *attach* operation.

- Finally, one might think of systems with different kinds of vehicles that could be represented by objects of different subtypes of *Position*. For purposes of collision avoidance, car agents would still try to *rd* objects of type *Position*. Because Objective Linda's matching considers subtyping, agents could get objects of several subtypes of *Position* in one multiset for a given range.

  For different purposes, agents might look directly for a subtype e.g. in order to answer the question "is there a truck available?"

# 5 Conclusion

Multi–agent systems need more than the simple (low–level) communication abstractions as they are provided by RM–ODP, CORBA, or languages like Java. Coordination models, esp. Objective Linda which has been designed to meet the requirements of active agents on open systems, provide such abstractions on a higher and hence better–suited level. With the example of collision avoidance given in the previous section, we have illustrated the benefits of our approach.

We are currently experimenting with a prototype implementation of Objective Linda for the C++ programming language based on PVM [GBD+94] as communication platform. First results are encouraging and we have also shown that interoperability between heterogeneous platforms is generally feasible for languages like C++ when communication is based on Objective Linda [Kie96b]. In order to improve interoperability between heterogeneous platforms, we plan to provide the Objective Linda model to Java programs as our next step.

# References


[Adl95] Richard M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22, 1995.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M.I.T. Press, Cambridge, Massachusetts, 1986.

[Bur95] Hans-Dieter Burkhard. Agent–Oriented Programming for Open Systems. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents, ECAI–94 Workshop on Agent Theories, Architectures, and Languages*, number 890 in Lecture Notes in Artificial Intelligence, pages 291–306, Amsterdam, The Netherlands, 1995. Springer.

[BWA94] Paul Butcher, Alan Wood, and Martin Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[Cia90] Paolo Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.

[GBD+94] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[ISO94] ISO/IEC JTC1/SC21/WG7. Information Technology – Open Distributed Processing – ODP Trading Function. Draft ISO/IEC Standard 13235, Draft ITU–T Recommendation X.9tr, July 1994.

[ISO95] ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. Draft International Standard ISO/IEC 10746–1 to 10746–4, Draft ITU–T Recommendation X.901 to X.904, May 1995.

[Kie96a] Thilo Kielmann. Designing a Coordination Model for Open Systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, pages 267 – 284, Cesena, Italy, 1996. Springer. Proc. COORDINATION'96.

[Kie96b] Thilo 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).

[Lew95] Ted G. Lewis. Where is Client/Server Software Headed? *IEEE Computer*, 28(4):49–55, 1995.

[Obj93] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.

[Sea69] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, England, 1969.

[Sho93] Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[vM92] F. von Martial. *Coordinating Plans of Autonomous Agents.* Number 610 in Lecture Notes in Artificial Intelligence. Springer, 1992.

[Weg93] Peter Wegner. Tradeoffs between Reasoning and Modeling. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object–Oriented Programming*, pages 22–41. MIT Press, Cambridge, Mass., 1993.

[WZ88] Peter Wegner and Stanley 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.