Towards Generative Software Composition

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

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

Abstract

Software systems are increasingly required to be open and distributed. Research on software composition *proposes to build such open and distributed systems by* "gluing together" *basic building blocks (*components*) using certain kinds of* connections *between them [16, 18]. Whereas work in this area primarily focuses on more static aspects of composition in order to ensure that given collections of components cooperate successfully, the aspect of dynamically changing compositions is less elaborated. In this paper, we pick up ideas from generative coordination for open systems [9] and develop the vocabulary of a composition language for modelling dynamic composition and re–composition for which the term* Generative Software Composition *has been coined. We provide a formal semantics for our composition language which can be orthogonally integrated with the semantics of the Objective Linda coordination model [7]. We illustrate the expressiveness of our approach on an example.*

1. Introduction

Software systems are increasingly required to be open and distributed. Such systems are open not only in terms of network connections and interoperability support for heterogeneous hardware and software platforms, but, above all, in terms of evolving and changing requirements. In open systems, new active entities (usually called "objects", "agents", or "actors") may dynamically join and later leave, which allows for evolving self–organizing systems of interacting (intelligent) agents [1, 4]. Commonly, open systems are modelled based on active objects that communicate using the mechanisms of some given formalism, preferably expressed in terms of a *coordination model* [3, 9, 12].

Coordination is typically defined as *"managing dependencies between activities"* [11] or *"constrained interaction"*[19]. Hence, coordination is normally understood as the modelling of the interactions and dependencies between multiple concurrent entities. Furthermore, the definition *"the process of building programs by gluing together active pieces"* [6] indicates a direct link between the notions of coordination and composition.

The contribution of this work is based on the observation that Objective Linda, our coordination model for open systems, inherently contains mechanisms for dynamic composition and re–composition. Even more, we claim that every coordination model must include some composition mechanisms in order to provide useful coordination abstractions. We see this claim in analogy to the observation that every programming language must contain some coordination constructs in order to produce useful programs – e.g. computation is absolutely useless unless some input or results can be exchanged with the computation's environment. Without composition aspects, coordination boils down to (sophisticated forms of) synchronization and data exchange. It is argued in [6] that coordination aspects should preferably be dealt with through a coordination language that is orthogonally separated from a computation language (the sequential programming language). Analogously, we propose in this paper to orthogonally separate composition aspects from coordination issues.

Software composition is typically defined as *"the construction of software applications from components that implement abstractions pertaining to a particular problem domain"* [15]. Work in this area is primarily concerned with the problem of how to assemble several components such that the assembly (the composition) realizes some desired behaviour. In the related field of *architecture–description languages* [18], composition is seen as a way to describe systems based on independent components and their connections. Here, *configurations* describe system structure, independently of the elements being structured, while dynamic reconfiguration is recognized as an important property.

Recently, the notion of *components* evolved from (active) objects to a more generic term. The definition of components as *"static abstractions with plugs"* [14] where composition is performed by assembling *"plug–compatible"* components nicely illustrates this view. This approach has also been investigated for con-

current compositions of active objects. The works in [10, 13] investigate such compositions based on a client–server communication style in which clients can be bound to given services. The former also extends this notion to dynamically changing configurations by allowing dynamic service bindings.

In a previous work [7], we investigated the problem of "plug–compatibility" of active objects, unlike other works based on generative communication mechanisms, using the Objective Linda coordination model. It is the aim of this paper to extend our previous work to the problem of dynamic aspects of composition, concerned with dynamically changing configurations. Therefore, we develop a successor version of the original Objective Linda model, as it had been presented in [9]. The new version adds a dynamic composition language to basic Objective Linda. It considers Objective Linda's active agents (its active objects) as its notion of components. Because Objective Linda's agents themselves may be concurrent compositions of several "sub–agents", the composition mechanisms can be hierarchically applied to several abstraction levels of a system.

The remainder of this paper is organized as follows. In Section 2, we briefly review basic properties of our coordination model Objective Linda as presented in [9]. In Section 3, we elaborate a notion of composition and an according composition language based on Objective Linda's basic generative features yielding an enriched model covering dynamic software composition. Section 4 presents a formal semantics for the elements of our composition language. Section 5 illustrates our approach on example before Section 6 concludes our work and indicates directions for future work.

2. Basic Objective Linda

Objective Linda [9] is a coordination model based on the principles of Linda [6] while seamlessly adding concepts of object orientation. Tuples and tuple spaces are replaced by objects and object spaces, respectively. Note that for the remainder of this paper "objects" denote passive objects unless explicitly mentioned otherwise, and active objects are often called *agents*.

Objective Linda's primary contribution to coordination of open systems is its object model. Objects to be stored in object spaces are self–contained entities; their encapsulated state can only be affected through their interface operations. Objects are instances of abstract types which are described in a language–independent notation, called *Object Interchange Language* (OIL). Actual programs may hence be written in conventional object–oriented languages to which a binding of OIL types (e.g. to language–level classes) can be declared. In OIL, all types form a type hierarchy having a common ancestor called OIL_Object which defines the basic operations needed by all types. The OIL allows subtyping according to the "principle of substitutability" such that an object of type S which is a subtype of T can be used whenever an object of type T is expected.

Object matching (the process of identifying objects to be retrieved from object spaces) is accordingly based on object *types* and *predicates* defined by type interfaces. A potential reader of an object has to provide a so–called *template object* to the in or rd operation. The template's type T denotes the type of object to be retrieved from an object space, whereas subtypes of T will also satisfy the operation. Further selection is performed by a predicate match (provided by the type OIL_Object) which takes an object of the same type as parameter and returns a boolean value determining whether a given object matches certain requirements based on predicates defined in T's interface. match does so (and can only do so because of object encapsulation) by evaluating predicates defined in T's interface. Hence, whenever an in or rd operation is performed using a template object of type T, objects of type T and its subtypes are presented to the template's match predicate until this predicate returns true. Several variants of matching objects of a given type can be selected by presetting the encapsulated state of the template object.

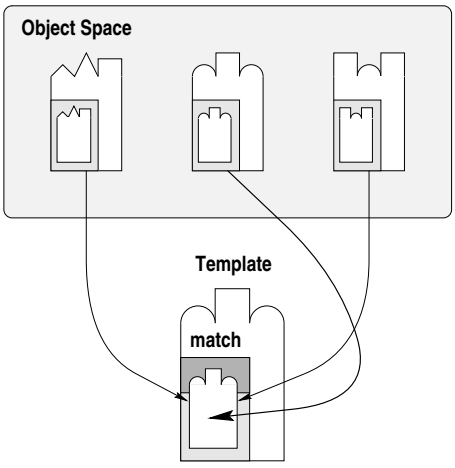

Figure 1: Object matching

This is illustrated in Figure 1. Objects of different types have different shapes, representing differ-

ent type interfaces. Every type implements a match predicate that has a socket into which only objects of the same type can be plugged in for evaluating the matching predicate. Hence, object matching is primarily performed on object types. Additionally, objects of a matching type may be further investigated by the match predicate of a template object via their interface operations. Objects belonging to subtypes of other types behave like "chameleons"; they may change their shape in order to fit into sockets of their supertypes, too.

Besides adapting the Linda model to object-orientation, Objective Linda also provides an improved set of operations on object spaces. 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, allowing to adapt communication behaviour to dynamically changing environments. Second, Objective Linda introduces *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 and max (which is possibly an infinite value) the maximal number of objects to be found in order to successfully complete the operation For reasons of simplicity and consistency, also the out operation takes multisets of objects to be stored in an object space. As has been shown in [9], Objective Linda's operations build a minimalistic set which covers Linda's operations as well as the needs of open systems.

Like their passive counterparts, active objects are also 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 in the moment of its evaluation. Hence, every active object has the behaviour implemented by the evaluate routine of its OIL type.

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 reflects the demands of open systems by treating active objects as encapsulated and reactive agents. Active objects simply disappear after termination. Analogous to Linda, active objects are invisible to operations in charge of retrieving passive objects from object spaces. Agents can only be observed by monitoring the passive objects they produce.

3. Modelling dynamic software composition

While coining the term *generative software composition*, we aim at describing a composition model in which generative features serve as the basis for the interaction between agents as well as for purposes of dynamically assembling and re–assembling configurations. In the sense of components being *"static abstractions with plugs"* we can identify agents as *static abstractions* and the object spaces attached to them as *plugs*. So far, we have the expressive power of any Linda–like system, enriched by Objective Linda's object model.

Configurations, as modelled with Objective Linda, consist of a number of agents, object spaces, and objects stored in the object spaces. Because agents can only be observed by their effects on object spaces (namely by the objects they out), we can not distinguish between a "simple" and a composite agent. Hence, with Objective Linda, we can uniformly deal with agents being defined on different abstraction levels, whether or not they are *"internally concurrent"* and whether or not they internally use "private" object spaces.

We distinguish between four different aspects of dynamic composition: creation and termination of agents, creation and deletion of object spaces, exposure and hiding of object spaces, and finally attaching to and detaching from object spaces.

3.1. Creation and termination of agents

Creation of Objective Linda agents is initiated by creating a (passive) object of the corresponding OIL type. This object may then be initialized for a certain task by calling operations from its type interface modifying the object's state. Finally, the object is passed as a parameter to the eval operation, performed on an object space. It is the responsibility of the eval operation to attach the object space in charge as the context object space of the new agent (as its default object space) and in turn to activate it by calling the object's evaluate routine.

Termination of agents simply happens whenever an agent's evaluate routine reaches its exit point. No further actions will be performed on behalf of agent termination, because Objective Linda's semantics define that termination has no effects on other agents or object spaces. Nevertheless, there may be the need for some *"garbage collection"* tasks on behalf of agent termination, but these are merely implementation–related and have no semantic meaning for Objective Linda configurations.

3.2. Creation and deletion of object spaces

New object spaces may be created at any time by any agent. This can easily be done as long as every newly created object space gets a unique identification assigned. This can be achieved without problems, even in open distributed systems, e.g. by using *Universal Unique ID's* as they are known from OSF's DCE [17].

In contrast, deletion of object spaces can not be performed explicitly. This is due to the nature of open systems in which the lack of a *"system closure"* prevents reasoning on the number of agents which may be attached to a given object space. Even worse, an object space could only be deleted when neither agents are attached to it nor agents may become attached to it in the future (cf. Section 3.4.). Nevertheless, implementations may in simple cases perform some garbage collection on object spaces.

3.3. Exposure and hiding of object spaces

In order to support flexible configuration topologies in which agents may use object spaces other than their context and the ones they created themselves, it is necessary to expose object spaces to other agents. As the core of our generative composition model, we propose to use generative communication for communicating object space information, namely by storing and retrieving objects to and from object spaces.

Objective Linda calls its key concept for exposing object spaces within object spaces *object space logical*. Such *logicals* combine an object of any subtype of OIL_Object with the identification of an object space. *Logicals* are, like regular passive objects, stored inside object spaces. They hence allow to identify object spaces based on properties or abstractions from the application's logic. Analogous to the operations in and rd, the selection of an object space to attach to is based on object matching. *Logicals* are created by a dedicated operation expose that takes as parameter an object with properties suitable for the application's needs in order to identify a given object space. Additionally, expose takes a (local) reference to the object space to be exposed and combines it with the object to a *logical* which is then stored inside another object space.

Reversely, it should also be possible to hide a certain object space from other agents, e.g. whenever some service will no longer be offered by agents "inside" this object space. As with other "destructive" composition operations, this is a subtle problem because of the latent danger of "hiding the wrong object space". In general, an expose operation can only be safely reverted by using (a clone of) exactly the *logical* object used for exposure. This is because hiding is performed by consuming a *logical* object *matching* a given template. In the case of an object space an agent "only" attached to, some information may get lost when the agent tries to construct a clone of the originally used *logical*. In fact, the agent can only reconstruct the template which it once used for attaching. In this case, the agent can not be sure to hide the intended *logical* object, because there may be other matching *logicals*, too.

To (partly) overcome this problem, the inverse operation to expose called hide takes as parameters a template object matching the *logical* and a reference to the exposed object space. hide removes the *logical* object from the object space if it matches the template object *and* if it refers to the same object space. This mechanism hence prevents agents from accidentally hiding the wrong object space. Unfortunately, unintendedly hiding another *logical* object to the same object space can not be avoided without an exact clone of the object originally used with the corresponding expose operation.

3.4. Attaching to and detaching from object spaces

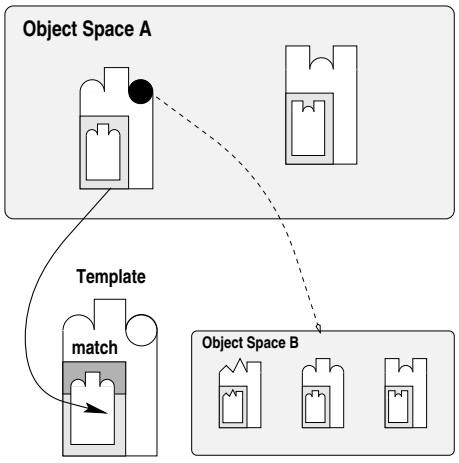

Figure 2: Generative attaching

Based on the *logical* concept, attachment to object spaces based on generative communication mechanisms becomes straightforward. The corresponding mechanism is illustrated in Figure 2. Inside object space A, the *logical* object on the left can be identified by the usual object–matching mechanism. Once the logical has been identified according to a template, the object space identification contained inside can be retrieved and used for attaching to object space B.

Objective Linda's corresponding operation called attach exactly performs these two tasks. It (non–destructively) retrieves a *logical* object matching a template and extracts the object space identification stored inside. It then returns a reference to the object space in charge. Note that expose uses references to object–space objects on the level of the underlying programming language, but stores *identifications* of object spaces inside the *logical* objects. Consequently, the reference returned by attach may be different from the one stored by expose, but it will always be equivalent to it. This mechanism not only solves the problem of passing references along distributed systems, it also allows the implementation of attach to perform further operations like tests whether the object space in charge can be attached to (is physically reachable, allows attachment, etc.)

A reverse detach operation is of debatable use since detaching from an object space is merely the agent's promise to no longer access the object space. It has no further effect on other parts of a configuration except the agent itself. The only benefit would be to use detach operations as hints for an implementation the object spaces of which might be garbage collected. Because of this limited usefulness, we exclude a detach operation from our composition model.

Alternatively to Objective Linda's attach operation which rd's the *logical*, it might also be discussed whether an analogous operation based on the destructive in would be desirable. We have rejected such a concept because of two reasons. First, such an operation would mix different concepts, namely dynamic composition and mutual exclusion. Hence, it would contradict our goal of presenting a composition model which is orthogonally separated from the coordination model. Second, the reverse operation of such a destructive variant of attach would suffer from the problems already outlined concerning the hiding of object spaces.

3.5. A dynamic composition language

We can now summarize our model for dynamic composition of open systems by presenting informal descriptions of the operations of a corresponding composition language. The notation is based on an OIL binding to the C++ language, and thus the interface of a class Object_Space is shown. In addition to the operations listed below, object spaces also provide the operations out, in, and rd for the composition–unrelated aspects of Objective Linda, as presented in the previous section.

Object_Space (void)
: Creates a new object space. (also called new)

bool eval (OIL_Object *o ,
double timeout = infinite_time)
: Tries to move the object o into the object space and starts its evaluate routine. 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 from the object space which matches o and to which s had been assigned. Returns true if such an object could be found within timeout seconds. Returns false otherwise.

Object_Space *attach (OIL_Object *o ,
double timeout = infinite_time)
: Tries to get attached to an object space for which a *logical* matching o can be found in the object space. Returns a valid reference to the newly attached object space if a matching *logical* could be found within timeout seconds; otherwise the result has a NULL value.

This set of composition operations covers all aspects of dynamic composition with Objective Linda. We achieve a minimalistic but powerful set of operations for expressing all aspects of dynamic software composition which is still orthogonally separated from Objective Linda's coordination operations.

4. Formal semantics

We now sketch a formal semantics for our composition language. As the above–defined composition operations complement Objective Linda's basic operations, we present their semantics using the PNSOL formalism (*Petri Net Semantics of Objective Linda*) as presented in [7]. Below, we only briefly review PNSOL's features in order to present the semantics of our composition operations.

The basic idea behind PNSOL is to model every agent by a separate net. In such an *agent net*, particular transitions represent operations on object spaces. In the PNSOL formalism, this is reflected by the fact that these transitions are adjacent to *places representing object spaces*. This way, every kind of agent may

be specified separately. In order to reason on the behaviour of a configuration consisting of several agents and several object spaces, one has to transform all necessary agent nets into one *"overall net"* by merging corresponding object–space places with each other.

4.1. The basics of PNSOL

PNSOL is based on features of Coloured Petri Nets (CPNs) [8] enhanced by timeout annotations. PNSOL primarily relies on CPNs, because their truly concurrent semantics allow to model concurrent behaviour in open systems without centralized control instances. Furthermore, CPNs already deal with typed data items and allow to have multisets of tokens to be moved between transitions and places. In order to fit Objective Linda's needs, we define $\{T\}_{min}^{max}$ as the syntax denoting a multiset of objects of type T with at least *min* elements, and at most *max* elements. Additionally, $\{t_1, t_2, \ldots, t_n\}$ denotes an explicitly enumerated multiset.

In order to model Objective Linda's operations, a timeout mechanism is introduced. This mechanism provides a kind of *"macro definition"*. Because these macro definitions can completely be transformed into plain CPNs, PNSOL nets remain applicable to formal analysis.

For modelling timeout behaviour, we add a new kind of input arcs to transitions representing Objective Linda operations. These so–called *non–deterministic input arcs* are drawn with white arrow heads. Enabledness of a *"macro"* transition is directly determined by its traditional input arcs. In the case when objects are also available from a transition's non–deterministic input places, the transition will occur *"normally"*, producing tokens at its traditional output arcs. But whenever a *"macro"* transition has been enabled (by its traditional input places) for at most *timeout* time without occurring (e.g. due to objects not being available at its non–deterministic input places), it is forced to occur *"abnormally"*, producing tokens only at its timeout output arcs (also drawn with white arrow heads.)

For modeling dynamically changing configurations, we have to present a second *"macro"* feature of PNSOL, called *named places*. It is intended for modelling object space places that are shown with small named boxes (their *"name places"*). These *name places* represent identifications of the object spaces actually in use. In order to avoid dynamically changing nets in the *overall* CPN, agent nets are transformed to plain CPNs by merging all object spaces into a single place. Then, the individual object spaces are distinguished by changing all annotations from objects o to tuples $< i, o >$ where i denotes the respective object–space id.

4.2. Semantics of the dynamic composition operations

The semantics of our dynamic composition operations in PNSOL notation are shown in Figure 3. Figure 3.a shows the eval operation, which takes the id of the object space in charge and assigns it to the agent, before the latter is stored into the *AgentSpace* place, which is a special place containing all active agents. The *AgentSpace* place is also merged to a single, global place whenever an overall net is constructed.

Figure 3.b shows the new operation which is responsible for creating new object spaces. This is a perfectly deterministic operation, because the creation of object spaces can be performed independently by every agent. new stores a new id for the newly created object space into its name place. In the translation to plain CPNs, this id token is withdrawn from another global place, containing "sufficiently many" unique id tokens.

Figure 3.c shows the attach operation which takes (a copy of) a single matching *logical* object, retrieve's its object–space id, and stores this id in objsp2's name place.

The expose operation, presented in Figure 3.d, stores a single *logical* object o which has been assigned the id of the object space to be exposed (objsp1).

Finally, Figure 3.e shows the hide operation. This operation removes a single *logical* object matching o, to which the id of the object space to hide (objsp1) had been assigned.

5. Example: an electronic meeting system

The example system that we use to illustrate our composition language is borrowed from the field of groupware applications. Groupware systems aim to support collaborative work of multiple users in a computerized environment. The application is a simple version of a system which supports electronic meetings (such as video conferencing and multi–user talk systems).

Users can propose meetings with their colleagues. The users which match the characterization of a meeting proposal can participate in the meeting. Participants perform their discussion by exchanging data and information, such as lines of text, pictures, documents, etc.

Similarly to what is seen in real world meetings, participants may decide to start side discussions for a number of reasons: e.g. because the meeting agenda

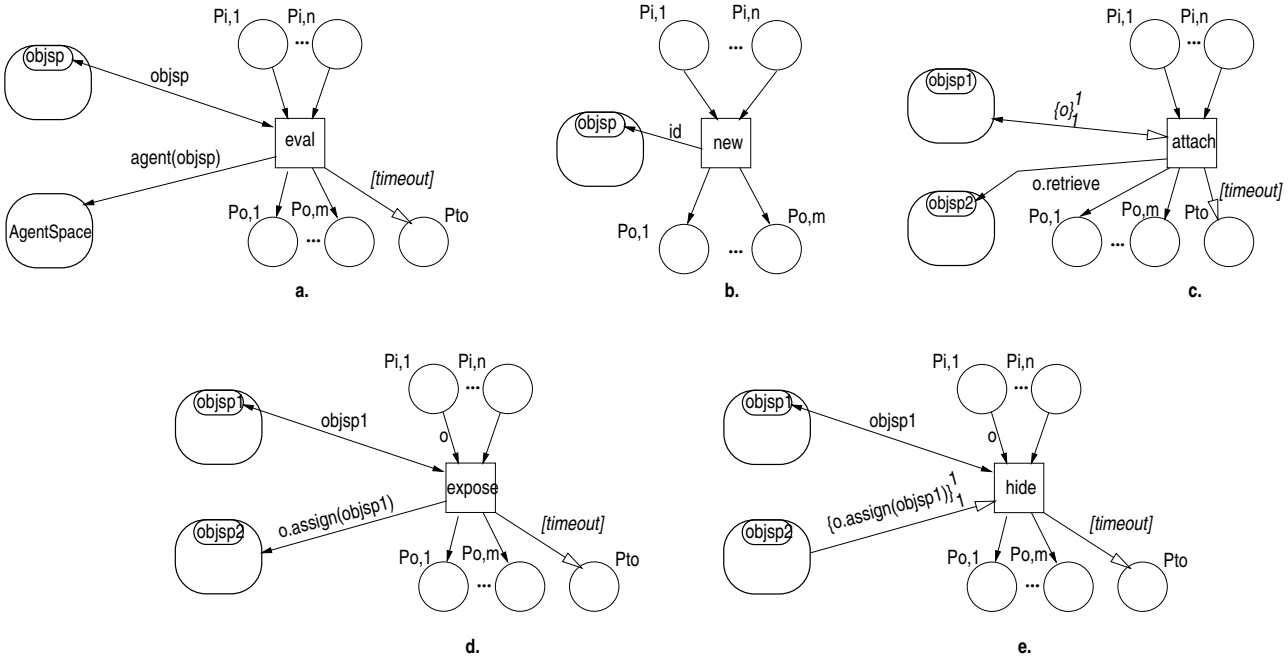

Figure 3: Modelling composition operations

may be too ambitious, forcing to divide the work into multiple smaller meetings (e.g. work groups), in order to talk about an interesting soccer game, etc. As a result, the configuration of a particular meeting can be changed dynamically. A participant can propose another "meeting place", while others can join this new session, and hence recompose themselves to another group of participants.

5.1. Meeting proposers and participants

The initial configuration of the entire meeting system consists of one object space. Here we propose two classes of agents through which people can start using the system: a Proposer and a Participant class. The agents of these classes can be looked upon as two user interface programs.

A user can construct a meeting proposal by creating a Proposer agent. A meeting proposal can be characterized by its subject, the date and time at which the meeting should start, the attendants' profiles, etc. The attendants' profiles may be as simple as a list of user names of expected participants, or information about the expected interests, the required responsibilities for joining the meeting, etc. Later, this agent may decide to withdraw the proposal, e.g. if the subject is outdated, or if no further participants to that meeting are allowed any more.

A Participant agent can be instantiated by a user to allow him/her-self to attend meetings. This agent should have the knowledge about the profile of the user it represents. As soon as the agent retrieves a meeting proposal matching its user's characteristics and/or interests, the agent transfers the user to the proposed meeting room. A meeting room is an object space which is shared by all participants of a particular meeting. Participants can interact through the meeting room. They can also create new meeting rooms and provide the others with the necessary information such that they can join a side discussion in a newly created meeting room. When participants leave a meeting, they return to their initial state, i.e. looking for suitable meetings to join.

A possible scenario is depicted in Figure 4. We assume that at the start of the scenario, the system configuration (shown in Figure 4.I) consists of five Participant agents and two Proposer agents, the latter each having put the relevant information and OS_Logical objects for respective meeting rooms into the shared object space called MeetingProposalObjectSpace. After some time, four participants have matched the meeting information of one of the meeting proposals, and they have attached to the appropriate meeting room, i.e. MeetingRoom1, shown in Figure 4.II. Here, one of the participants in this meeting has already created a new meeting room, called MeetingRoom3, which will be used for a particular side discussion. Two of the four participants then recompose themselves towards this new meeting room (Figure 4.III). When they finish their discussion, they re–establish

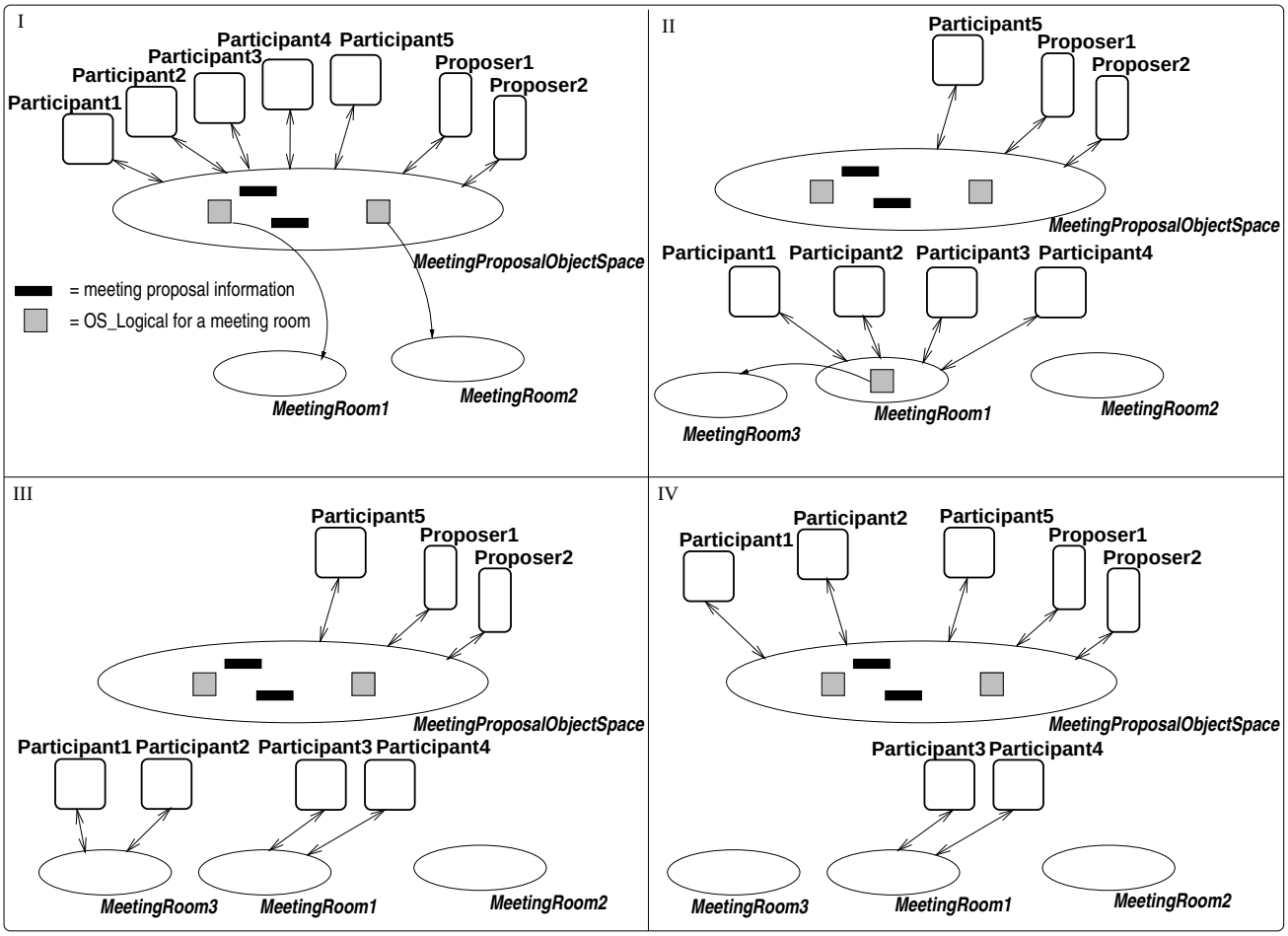

Figure 4: Dynamic Meeting Configurations

their initial state, while the others may continue their own discussion (Figure 4.IV).

5.2. Agent net specifications

According to the semantics presented in Section 4., Figures 5 and 6 depict the net descriptions for Proposer and Participant agents, respectively. The agent nets reflect mostly coordination and composition behaviour. Most internal computations, such as the construction of user profiles and objects to share in a meeting are left open for further specification. The net specifications are by no means intended to be complete. They merely serve to give an intuitive understanding of how such descriptions look like, and how they can be used to model agent behaviour.

The Proposer agent net contains two object space places. On creation, the context object space becomes the object space in which the agent is eval-ed, which is an object space such as MeetingProposalObjectSpace in Figure 4. The transition MakeProposalInfo represents the section in which the user who initiated the proposer agent can build the prerequisites for a forthcoming meeting. The result is a passive object of class MeetingInfo. This information can then be used to construct a suitable *logical* object for the meeting room object space. This object space is created when the Proposer agent is constructed, and it will be used as the meeting room object space for the intended meeting. This *logical* is then exposed into the context object space, which will allow participants to join the meeting. After some time, the proposer may decide to not allow any new participants to this meeting, and the agent can withdraw (hide) the *logical* object it had exposed before.

The Participant agent net is modelled around three object spaces. The context object space is the object space which is supposed to contain useful information on scheduled meetings (like e.g. MeetingProposalOb-

jectSpace.) If the Participant can find particular meeting information which matches its user's profile, it can construct an appropriate *logical* template which will match the corresponding *logical* object in the context object space. Once the attach operation is completed, the participant has joined the meeting. In this meeting, a participant can (1) follow the meeting by inspecting (rd'ing) the objects which have been inserted into the meetingroom object space; (2) build ExchangeableObjects itself and share (out) them with the other participants; (3) recompose itself to a side discussion that another participant is setting up by providing a suitable *logical* to a new meeting room; and (4) create a new discussion room by itself and exposing an appropriate *logical* to which other participants can recompose themselves. A participant can also decide to quit the meeting, in which case its initial state is re–established.

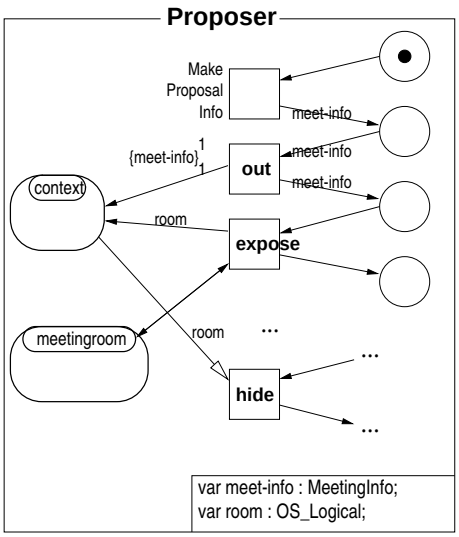

Figure 5: Net specification for Proposer agents

5.3. Discussion

The "Electronic Meeting System" as described above is an oversimplified version of what a real world system could be. For example, a real application should specify a type (according to the *agent types* as introduced in [7]) for the agents in the system. Types should be used not only to detect erroneous agents willing to use the system, they also serve to statically check particular security aspects: one could check whether the Participant agent does not join meetings it is not supposed to, does not withdraw e.g. meeting information delivered by another Proposer, and so on. However, the example is illustrative enough to offer

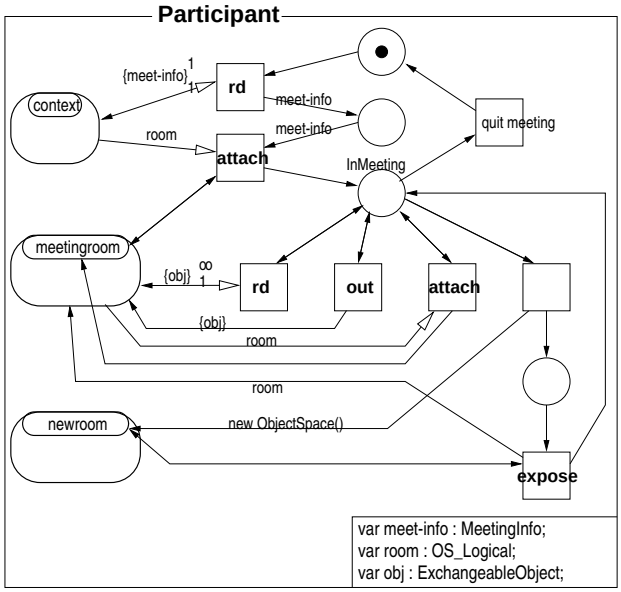

Figure 6: Net specification for Participant agents

an intuitive idea of how flexible configurations can be achieved through dynamic composition.

The application scenario as it is described above is sufficiently *generic* as to be reused for other similar groupware systems: a *chat* system, in which e.g. chatters can change to a channel which another chatter has pointed out for them, a *game hall*, in which players can join ongoing multi–user games, etc. It is also rather straightforward to combine and integrate all these groupware systems. A possible scenario would be that during a meeting, someone says that he knows a chat channel (an object space) in which interesting things are discussed. Meeting participants can attach to this object space and start chatting. During this chat, someone points to a nice card game. By attaching to the appropriate object space, one can now find some distraction by playing a little game before going back to the meeting. This gives a clue towards the high degree of flexibility that can be achieved with the presented approach of dynamic composition.

6. Conclusions

In this paper, we have investigated the notions of *coordination* and *software composition*, their close relation, and their still orthogonal contributions to the design of concurrent software applications. We have focussed our investigations on the increasingly important field of open system design with its special requirements for dealing with dynamically changing configurations.

Based on the observation that coordination models inherently contain composition mechanisms, we have identified the respective mechanisms in Objective Linda, our coordination model for open system design. We have developed a model for *generative software composition* in which Objective Linda's generative mechanisms are employed for modeling dynamic composition and re–composition of configurations in open systems.

According to this model, we presented the vocabulary of a dynamic composition language. This language provides a minimalistic but expressive set of operations for dynamically changing software configurations. This language nicely complements Objective Linda's composition–unrelated operations. We have presented a formal semantics for our composition language based on the PNSOL formalism [7] which is based on Coloured Petri Nets. By integrating composition aspects into the PNSOL framework, we are able to benefit from automated analysis, as it is e.g. performed by the Design/CPN tool [8].

Finally, we have illustrated usefulness and adequacy of our approach by modelling a highly flexible groupware architecture in which Objective Linda and the generative composition language nicely complement each other.

Dynamic composition is, however, not the only aspect of software composition. Of at least equal importance is the more *"static"* assembly of given components in order to build composite applications. Here, the topic of interest is the notion of *plug compatibility* which enables to decide which kinds of components may be successfully combined in order to achieve a desired behaviour. Our notion of components as being agents or their concurrent compositions already proofed to be suited as a basic abstraction for this form of "static" composition [7]. Future work in this area will hence concentrate on the definition of a complementary *static composition language* which allows to specify components and their interconnections. Recent results [7] show that our uniform formal model for agents, components and types allows us to reason upon the behaviour of generic components (components of which some elements are represented only by their type), an ability which is so far unmatched by other approaches dealing with types or components.

References

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

[2] G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object–Oriented Programming.* MIT Press, Cambridge, Mass., 1993.

[3] C. J. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1994.

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

[5] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, LNCS 1061, 1996. Springer. Proc. COORDINATION'96.

[6] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Commun. ACM*, 35(2):96–107, 1992.

[7] T. Holvoet and T. Kielmann. Behaviour Specification of Active Objects in Open Generative Communication Environments. In *Proc. of the Thirtieth Annual Hawaii International Conference on System Sciences*, vol. 1, pp. 349–358, Wailea, Hawai'i, USA, 1997. IEEE.

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

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

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architecture. In *Software Engineering – ESEC'95*, LNCS 989, pp. 137–153, 1995. Springer.

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

[12] O. Nierstrasz. Composing Active Objects. In Agha et al. [2], chap. 5, pp. 151–171.

[13] O. Nierstrasz. Regular Types for Active Objects. In Nierstrasz and Tsichritzis [16], chap. 4, pp. 99–121.

[14] O. Nierstrasz and L. Dami. Component–Oriented Software Technology. In Nierstrasz and Tsichritzis [16], chap. 1, pp. 3–28.

[15] O. Nierstrasz and T. D. Meijler. Research Directions in Software Composition. *ACM Comput. Surv.*, 27(2):262–264, 1995.

[16] O. Nierstrasz and D. Tsichritzis, editors. *Object–Oriented Software Composition.* Prentice Hall, 1995.

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

[18] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[19] P. Wegner. Coordination as Constrained Interaction. In Ciancarini and Hankin [5], pp. 28–33. Proc. COORDINATION'96.