# MiNEMA Exchange Grant: Scientific Report

Kurt Schelfthout, Grant ID 771

## 1 Purpose of the visit

The purpose of the visit was two-fold.

The **main purpose** of the visit was to collaborate on a specific problem that we encountered while applying a middleware for mobile networks [3]. This middleware enables the automatic activation of *roles* at certain network nodes, see Fig. 1. A role is the behavior of one partner in an interaction; as such, an interaction is completely specified by the behavior of all of its roles. The middleware supports protocol-based interaction in mobile networks by activating roles, and thus interactions, based on the current context in the network. For example, a collision avoider role is activated whenever two vehicles are entering collision range. A consistent activation of roles is maintained in the network as network nodes move, i.e. the middleware makes protocol-based coordination easier by dealing with node mobility.

The specific problem on which we collaborated is the following. Any one node can be involved in multiple protocols. For example, a node can be involved in a mutual exclusion protocol and deadlock detection at the same time. These protocols can be dependent on each other; for example, a deadlock detection protocol may need information from mutual exclusion, such as which other nodes a node is currently waiting for. Since the protocols are embodied in their respective roles, these dependencies need to be handled between the roles active on a specific node. We call this *role composition*, reflecting the fact that independently developed roles need to be composed into a working and coherent system for every network node.

The existing solution was to use a tuplespace to allow roles on one node to exchange data, and to coordinate [3]. This means that roles need to take the presence of other roles in the system into account explicitly, e.g. one role needs to produce data in a form the other roles expect, if they are to use the data. This has a detrimental effect on the reusability of the roles, and thus on the reusability of the protocols, see Fig. 2. Also, it does not offer much support to the application developer, for example it cannot guarantee any properties of the composition (except by testing and debugging). A better form of role composition is thus desirable.

The purpose was to collaboratively come to an appropriate solution for this problem, based on the host institution's experience with role-based coordination and coordination in general.

The **second purpose** was to lay the fundamentals for a joint paper for the Journal of Autonomous Agents and Multi-Agent Systems. At the E4MAS [1] workshop at the

[1] Environments for Multi-Agent Systems, `http://www.cs.kuleuven.ac.be/~distrinet/`

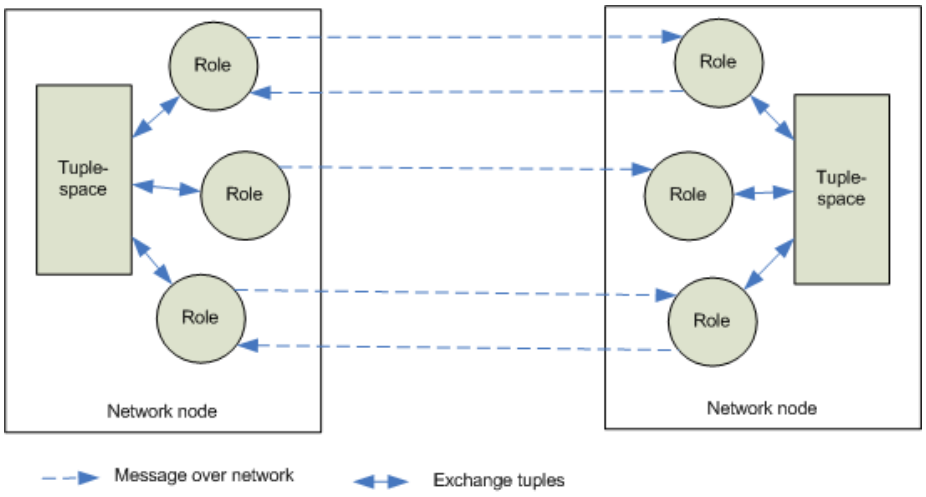


Figure 1: Two network nodes, with roles interacting directly in between network nodes, and indirectly through a shared tuplespace on one node.

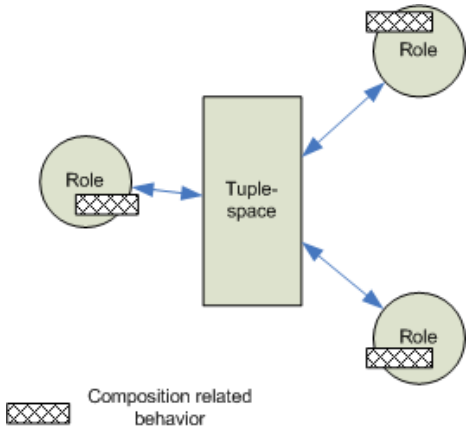


Figure 2: Schematic representation of the behavior necessary for composition of roles using a tuplespace: it is scattered over the roles.

AAMAS'05 [2] conference, five groups were formed to produce five papers in a special issue of the Journal of Autonomous Agents and Multi-Agent Systems [3]. We are co-authoring a paper with Mirko Viroli and Alessandro Riccion, both working at the host institute in Italy, on the topic of "Engineering Environments". This visit was thus an excellent opportunity to collaborate on the paper.

# 2 Work carried out & results obtained

The main idea that emerged during the visit to tackle the problem presented above is the following. We propose to consider roles as black boxes, which are described by stating the expectations they have from the other roles, and the provisions they make towards the other roles. This description can be used in two ways:

1. As a basis for the static checking of the composition of a set of roles.
2. As a basis for the designer, to specify the necessary rules for the composition of the various roles. These rules determine the actions of a composition infrastructure at runtime, which can provide the necessary "glue" to tie together roles.

First a proposal for behavioral description of a role is given, then possible composition support using this behavior description is described, both statically and at runtime.

## 2.1 Behavioral description

The behavior of a role is described in terms of its actions on a shared tuplespace. In other words, a role behavioral description describes the data that a role needs, and that it produces and consumes.

As a fairly simple, yet expressive description of a role we use a state chart model, where each transition consists of preconditions, an operation name, and postconditions. Preconditions express that certain tuples must be available for the transition to fire. When the precondition is fulfilled, the transition fires, and the effect on the shared tuplespace is expressed in the postconditions of the transition.

A precondition consists of a set of constraints on the contents of the tuplespace, which can be expressed as *contains(DataType* $>=$*Number*, indicating that the tuplespace should contain at least *Number* tuples of type *DataType*.

An operation is the name of the operation that is executed on the tuplespace when the transition fires. This is added for clarity reasons, and has no semantic meaning per se. This operation may be any operation that the tuplespace supports.

A postcondition consists of a set of *consumes*, *produces* and *reads* statements, indicating the tuples that are removed, added or read from the tuplespace.

So, the operational meaning of the behavior description `start->pre/someop/post -> end` is that the role waits in state `start` until `pre` is true (until the tuplespace contains a

`events/e4mas/2005/`

[2] Autonomous Agents and Multi-Agent Systems, `http://www.aamas2005.nl`

[3] `http://www.springer.com/sgw/cda/frontpage/0,,5-40100-70-35744137-0,00.html`

certain number of tuples of a certain type); then executes the operation `someop`, which has as effect that atomically the contents of the tuplespace are changed according to the production and consumption rules in `post`.

For example,

```
start -> contains(Position >= 1) / updatePosition /
consumes(Position= 1) produces(Position = 1) -> end
```

is a role that, in state `start` executes the `updatePosition` operation, which waits until the tuplespace contains at least one Position tuple, then atomically consumes a Position tuple and also produces a Position tuple.

Multiple `contains` clauses may be given, and are composed with "and" semantics; "or" semantics can achieved by introducing multiple transitions from the same state. `consumes` and `produces` clauses are not commutative, so the ordering is important, but the "execution" is always atomic.

## 2.2 Support for composition

Using the above behavioral description, two kinds of support can be envisioned: static checking of properties and dynamic enforcement or support of composition properties.

### 2.2.1 Static checking of the composition

By using the information contained in the roles' behavior description, it is possible to guarantee (some) properties of the composition statically. Examples of static checking:

- checking whether the tuple types required in preconditions are produced (in enough quantity)
- checking whether there are tuple types that are produced but not consumed by any role (indicates superfluous data production)
- worst case deadlock analysis

During the visit, a proof-of-concept implementation of a static check for the two first items was done in Java. This would enable an application developer to find errors in the composition quickly.

A trade-off encountered here is typically that, in principle, the better the behavioral description, the better the guarantee that can be given, but also the more computationally expensive the checking becomes. Advanced guarantees can only be obtained by "simulating" the behavior descriptions of all roles, which suffers from the state explosion problem, and so is computationally infeasible.

### 2.2.2 Dynamic guiding of the composition

Given the behavioral description of the roles, a run time infrastructure can "guide" the composition, keeping it in a safe state and detecting and correcting problems at runtime. In particular, an infrastructure can:

- delay execution of an operation
- transform an operation into another operation
- transform the data produced or consumed by the operation, or present in the tuplespace.

Some examples of dynamic guiding are deadlock avoidance by delaying certain operations, given the current state of each role and the expected behavior of each role as given by the behavior description; or data transformation rules (duplicating tuples, transforming tuples,...) to extract information from a tuple that is needed by a role.

The place where such transformation should be done is in the shared tuplespace to which all roles are connected. For this reason, we use a *programmable tuplespace*, which is a tuplespace that can be programmed with additional behavior for the base operations (Linda [1] like, i.e. *in, out, rd, inp, rdp*), or with entirely new operations. Such a tuplespace can be programmed to include the composition rules, or "glue", for roles. Rules are triggered upon certain operations on the tuplespace. For example, a data transformation rule can be straightforwardly added to a programmable tuplespace.

The basic programmable tuplespace can be augmented with support for interpreting a set of role behavioral descriptions and support for runtime observing of a set of roles executing on a tuplespace. This would simplify the work of the application developer when adding the application-specific composition rules.

Putting the composition rules in the programmable tuplespace has two distinct advantages vs. programming the composition rules in the roles themselves, mainly because concerns are better separated. First, roles and the protocols they constitute are more reusable, since the dependencies between roles are handled outside of the roles themselves. Individual roles no longer need to be changed in order to work together with other roles, see Fig. 3. Secondly, due to the static checking, and support at runtime, more guarantees can be given regarding the correctness of the composition.

# 3 Future collaboration & Projected publications/articles

While the fundamental ideas have been outlined, we aim to implement a prototype infrastructure using TuCSoN tuple centres [2], which are tuplespaces that are programmable in a first-order logic language. Also, a case study using the real world problem of AGV control will be realized using this prototype [4]. Furthermore, we are working on a joint publication on this work, with Alessandro Ricci and Andrea Omicini.

Further collaboration will also ensue with Mirko Viroli and Alessandro Ricci on the JAAMAS paper (see above).

# References

[1] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986.

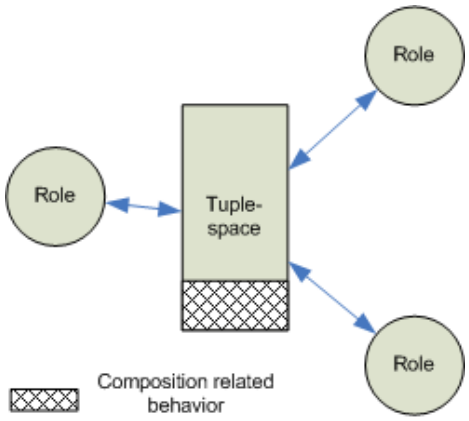


Figure 3: Roles interacting in a programmable tuplespace. The composition rules can be conveniently programmed in the programmable tuplespace. The roles can be considered black boxes, and are more easily reusable.

[2] A. Omicini and F. Zambonelli. The tucson coordination model for mobile information agents. In *Proc. of the 1st Workshop on Innovative Internet Information Systems*, 1998.

[3] K. Schelfthout, D. Weyns, and T. Holvoet. Middleware for protocol-based coordination in dynamic networks. In *Proc. of 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC05)*, 2005.

[4] D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of E'GV transportation systems. In *Proceedings of AAMAS 2005 - Industry Track*, Utrecht, The Netherlands, 2005.