1. COMPOSING COORDINATION SPACE

Originally conceived in the context of closed, parallel systems, coordination models and languages soon proved their effectiveness in the engineering of open, distributed systems [8]. Nowadays, space-based coordination models are developing to tackle with the issues of complex computational systems such as pervasive and knowledge-intensive systems [2, 6, 3]. There, in order to deal with strong dynamicity, multiple coordination flows, physical mobility, heterogeneous knowledge sources and the like, dynamic composition of expressive coordination abstractions is required, involving both the information contained in the shared spaces, and the laws of coordination embedded in the coordination media.

Space-based approaches to dynamic composition of coordination abstractions either focus on the composition of the information space or exploit the composition of the coordination space. The latter class of models provide engines with mechanisms for connecting and composing coordination media at runtime. The two most notable examples here are LGI [5] and TuCSoN [9]. LGI (Law-Governed Interaction) exploits composition of coordination abstractions in order to distribute coordination laws locally to individual agents—so, in a sense, composition is here implicit, and not explicitly usable by programmers of coordination policies. Instead, TuCSoN provide the means for explicit composition of coordination abstractions: linkability of tuple centers makes it possible to dynamically compose the coordinating behaviors of different coordination media [7]. On the other hand, linkability in TuCSoN represent an ad hoc composition mechanism, connecting specific coordination flows within the tuple centers, with no systematic integration of the coordination media—whereas, integration of global and individual coordination abstractions is systematic in LGI.

Here, the most interesting class of models exploits instead the integration of the items in the shared information space to compose coordination abstractions—most typically, tuples. Lime [10] was the first model to exploit dynamic composition of the information space to address the issues of agent mobility and context-awareness: there, in fact, mobile agents merge their private tuple spaces with the ones already in place when they move to a new location. The resulting coordination abstractions represent at any time the local contexts of interaction, including information from all the agents currently in place. In TOTA [3], tuples propagate through a tuple-based infrastructure, and merge with the local tuple spaces provided by the field-based coordination infrastructure. There, the laws of propagation (and decay) of tuples are embedded in the tuples themselves, and affect both the coordination infrastructure and the behavior of coordinated agents. LSA (Live Semantic Annotations) in SAPERE [11] on the one hand work as representatives of mobile agents, in the same way as tuples in Lime, to provide for context construction and awareness; on the other hand, they do propagate through different locations as in TOTA—however, based on chemical eco-laws rather than on self-contained laws of propagation.

Whatever the differences, however, all the above approaches share the fact that composition of the information space is dynamic yet structural, in that it depends on the architectural and topological properties of the infrastructure as well as on the mobility of the components. There is no mechanism, then, allowing for an ad hoc composition of the coordination abstractions based on more specific application needs, possibly arising at runtime in an unpredictable way.

This is where LogOp steps in: LogOp process, in fact, can compose coordination abstractions on an operation basis—that is, every operation accessing the coordination space can in principle define the scope where it will be executed. As in the Scopes model [4], a LogOp scope is a generalization of the context of tuple space as the target of a coordination primitive—unlike Scopes, however, it is not just a view over the tuples in tuple spaces. The simplest scope in LogOp is a single tuple space. More generally, scopes in LogOp are built out of tuple spaces by combining them dynamically by means of logical operators. So, each invocation of a coordination primitive can be associated in principle to a possibly different scope, built at runtime to address the current application needs. To the best of our knowledge, these features are unique to LogOp in the field of coordination models.

2. LogOp IN SHORT

The first fundamental LogOp feature is to allow multiple tuple spaces to be combined and accessed altogether within a single LogOp/LINDA coordination primitive. On the
one hand, this allows any programmer familiar with LINDA style to keep on relying on its understanding and practice of LINDA primitives: concurrency is introduced without new primitives, and with no changes to the original semantics of the basic LINDA primitives. On the other hand, an easy-to-handle level of abstraction is introduced, by generalising the notion of scope for a LINDA operation. So, instead of being limited to a single tuple space as in LINDA, the scope for a LogOp coordination primitive can be dynamically built as a composition of tuple spaces.

The second fundamental feature of LogOp is to adopt logical operators for dynamic tuple space composition. So, multiple tuple spaces can be combined and accessed within a single LogOp operation by means of LogOp operators AND, OR and XOR—for instance, the expression \( \text{ts1 AND ts2} \) (where ts1, ts2 are tuple space identifiers) denotes an admissible scope for a LINDA primitive in LogOp. This makes it possible to achieve a satisfactory level of expressiveness while keeping complexity of language low, and also allows for an easy intuition of the basic meaning of the operators.

**Scopes.** The core of LogOp clearly lays in the notion of scope expression—henceforth simply scope—as defined by the SCOPE definition (Figure 1). A scope expression determines how tuple spaces are dynamically combined in articulated contexts for basic LINDA operators. Scope expressions are obtained through the application of logical operators (LOP). A SCOPE could either be a single tsID (tuple space identifier), or result by applying a logical operator LOP to a list of SCOPEs—i.e., at least two SCOPEs. As a result, logical operators LOP can be applied recursively, and combined with each other. For instance, \( \text{in(or(ts1,ts2,ts3),[?string])} \) is an admissible LogOp expression. In the LogOp expression \( \text{out(AND(ts1,XOR(ts2,ts3)),["resFree"])} \), AND is applied to a single tuple space (ts1) and to a scope built by the XOR operator over tuple space identifiers ts2 and ts3.

A scope expression determines a number of possible admissible scopes: (i) an admissible scope is the collection of tuple spaces where the primitive can be executed, and (ii) more admissible scopes for the same scope expression are actually possible due to the intrinsic non-determinism of LogOp. For instance, scope \( \text{AND(ts1,ts2,ts3)} \) has just one admissible scope, where the three spaces ts1, ts2, ts3 are accessed—the tuple has to be inserted in all the three spaces (in the case of an \( \text{in or rd} \)). On the other hand, scope \( \text{XOR(ts1,ts2,ts3)} \) has three admissible scopes: the one where only ts1 is accessed, the one where ts2 and ts3 are accessed, and the one where only ts3 is accessed. Finally, scope \( \text{OR(ts1,ts2,ts3)} \) has three admissible scopes: ts1 alone can be accessed, ts2 alone can be accessed, or both ts1 and ts2 can be accessed.

**Logical operators.** AND is an n-ary operator \( (n > 1) \) composing n scopes so that they *all* can be accessed by a single operation. Within an \( \text{op (AND(ts1,...,tsn),tuple)} \) invocation, the AND operator conceptually builds a single admissible scope \( \text{ts1, ..., tsn} \) for the LINDA operation \( \text{op(tuple)} \). Operation \( \text{op (AND(ts1,...,tsn),tuple)} \) succeeds when one \( \text{op (tuple)} \) operation is successfully executed upon each of the n tuple spaces in the multiset \( \text{ts1, ..., tsn} \).

XOR is an n-ary operator \( (n > 1) \) composing n scopes so that *one and only one* among the n is accessed by a single operation. Within an \( \text{op (XOR(ts1,...,tsn),tuple)} \) invocation, the XOR operator conceptually builds n admissible scopes—ts1, ..., tsn—one of which will be non-deterministically selected as the target for the LINDA operation \( \text{op(tuple)} \). Operation \( \text{op (XOR(ts1,...,tsn),tuple)} \) succeeds when the \( \text{op (tuple)} \) operation succeeds upon *one and only one* of the tuple spaces in the multiset \( \text{ts1, ..., tsn} \).

If AND represents all spaces in a multiset, and XOR one space in a multiset, then OR stands for some. In short, the OR operator conceptually combines tuple spaces (more generally, scopes) so that processes can access tuples within some of the tuple spaces in a multiset—in a sense, it has a non-deterministic effect ranging from all and one of the tuple spaces in a multiset. In an invocation of the form \( \text{op (OR(ts1,...,tsn),tuple)} \), the OR operator conceptually builds \( 2^n - 1 \) admissible scopes—\( \text{ts1}, \ldots, \text{tsn} \)—one of which is then non-deterministically selected as the target for \( \text{op (tuple)} \). So, LogOp operation \( \text{op (OR(ts1,...,tsn),tuple)} \) succeeds when LINDA operation \( \text{op (tuple)} \) succeeds upon some of the tuple spaces in the multiset \( \text{ts1, ..., tsn} \)—that is, any non-void multiset contained there.

3. LINEAR LOGIC FOR LogOp

LogOp operators (AND, OR and XOR) can be accurately understood via the classical logical interpretation only in the simplest case—that is, when used as binary operators upon a pair of different tuple spaces. In the general case, instead, this interpretation no longer suffices—not for poor language choices in LogOp, but mostly for the fact that classical logic does not fit well the distributed computing scenario, where resources are limited, physically distributed, and their availability is indeed crucial. There, in fact, access to resources may have side-effects, so that for instance accessing a resource and accessing the same resource again is not the same as accessing it one time only—and in LogOp, we are typically interested in operations which access a tuple space more than once. This obviously cannot be captured directly by classical logic, where \( a = (a \ and \ a) \). A fitter logical framework for LogOp is indeed provided by linear logic, the logic of consumption of resources [1]. In linear logic, hypotheses are interpreted as resources: every hypothesis is consumed exactly once in a proof, resulting in a deduction system handling the concurrency features sought by LogOp.

In linear logic, the validity of a formula can be deduced from a multiset of premises—a valid formula is said to be syntactically entailed by the premises. An example of operator in linear logic is additive conjunction \( (\otimes) \), which resembles the and operator of propositional logic \( (\land) \). For instance, we have \( p, q \dashv \vdash p \otimes q \), that is, atomic premises \( p \) and \( q \) (syntactically) entail formula \( p \otimes q \), or, in other terms, the validity of \( p \otimes q \) can be deduced from the validity of premises \( p \) and \( q \). On the other hand, we have \( p /\not\vdash p \otimes q \)—both argu-
ments are required in the premises. A peculiar aspect of linear logic is that $p, q$ is the only (multi)set of atomic premises syntactically entailing $p \otimes q$; for instance, $p, q, r \not\vdash p \otimes q$. This contrasts with classical logics extending from propositional logic, where instead it holds that $p, q, r \vdash p \land q$, given that adding premises does not prevent a formula from being deduced. In fact, linear logic was introduced as the logic of consumption of resources: $p$ and $q$ are considered as resources, and $p \otimes q$ as the process that needs such resources; validity means that the process exactly consumes the resources expressed by premises. Hence, $p, q, r$ do not entail $p \otimes q$ since they provide too many resources.

The notion of resource consumption is a key aspect in the modelling of LogOp, where scopes are precisely used to describe which tuple spaces (resources) have to be accessed by a primitive. In particular, an admissible scope precisely describes which tuple spaces (resources) have to be accessed.

We have the following sound deduction system:

For LogOp, we consider a fragment of linear logic with atomic formulas $(p, q, \ldots)$ and only operators for additive disjunction ($\oplus$) and additive conjunction ($\otimes$), hence defined by the following syntax:

$$ A ::= p \mid A \oplus A \mid A \otimes A $$

Let $A$, $B$, $C$ stand for formulas, $\Delta$ for multiset of formulas, operator $\oplus$ for (associative and commutative) multiset union, and “$\Delta \vdash A$” for premises $\Delta$ syntactically entail $A$.

We have the following sound deduction system:

$$
\begin{align*}
\Delta \vdash & A \quad (i)
\Delta \vdash & B \quad (\Delta_1 \vdash B_1 \vdash C \quad (\oplus))
\Delta \vdash & B_2 \quad (\Delta_2 \vdash B \vdash A \quad (\otimes))
\Delta \vdash & B_1 \oplus B_2
\end{align*}
$$

Rule (i) states that an atomic formula $A$ is valid if premises are made by $A$ only; rule (\oplus) states that $B \otimes C$ is valid if its premises can be separated into those entailing $B$ ($\Delta_1$) and those entailing $C$ ($\Delta_2$); finally rules (\oplus1) and (\otimes2) state that $B_1 \otimes B_2$ is entailed by any premises $\Delta$ entailing either $B_1$ or $B_2$. For instance, it is easy to see that $p \vdash p \otimes q$ and $q \vdash p \otimes q$ hold, while $p, q \vdash p \otimes q$ does not—hence $\otimes$ resembles a $\otimes$. Instead, $p, q \vdash p \otimes q$ holds, whereas $p \vdash p \otimes q$ and $q \vdash p \otimes q$ do not—hence $\otimes$ resembles an $\otimes$.

By considering the right-hand side as an input, and the left-hand side as an output, a relation $\Delta \vdash A$ can be recast in terms of finding a multiset $\Delta$ of atomic premises making formula $A$ valid. In LogOp, then, we can define the meaning of a scope expression in terms of the multiset of identifiers that syntactically entail the corresponding linear logic formula. Such a multiset precisely describes where the LogOp operation has to be performed; if many such multisets exist, then the LogOp operation can be performed on any of them, non-deterministically. That is, a multiset of premises that entail a scope expression is precisely an admissible scope for that expression.

Accordingly, LogOp operators can be mapped onto linear logic operators as follows:

$$(s \text{ AND } r) \rightarrow (s \otimes r)$$
$$(s \text{ XOR } r) \rightarrow (s \oplus r)$$
$$(s \text{ OR } r) \rightarrow ((s \oplus r) \oplus (s \oplus r))$$

That is, AND corresponds to additive conjunction, XOR additive to disjunction, and OR can be defined in terms of AND and XOR—with two operands, it either takes both (conjunction) or one (disjunction) of them. Multi-arity of LogOp operators is identified in the usual way through associativity—namely, $\text{AND}(s1,s2,s3)$ is a shortcut for $(s1 \otimes s2) \otimes s3$, or $(s1,s2,s3)$ for $(s1 \oplus s2) \oplus s3$, and so on.

Accordingly, we have that $ts1, ts1 \vdash ts1 \oplus (ts1 \oplus ts2)$ and $ts1, ts2 \vdash ts1 \oplus (ts1 \oplus ts2)$ both multisets $ts1, ts1$ and $ts1, ts2$ make scope formula $ts1 \oplus (ts1 \oplus ts2)$ valid, so they are admissible scopes for scope expression $\text{AND}(ts1, \text{XOR}(ts1, ts2))$. Also, no other premises do the same, so that for instance $ts1, ts1, ts2 \not\vdash ts1 \oplus (ts1 \oplus ts2)$.

So, $ts1, ts1, ts2$ is not an admissible scope for expression $\text{AND}(ts1, \text{XOR}(ts1, ts2))$.

4. REFERENCES


