Università di Pisa

Dipartimento di Informatica

Technical Report : TR-98-12

Composing specifications for coordination

Montangero C. ; Semini L.

November 26, 1998

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Composing specifications for coordination

Carlo Montangero[1] and Laura Semini[2]

[1] Dipartimento di Informatica, Università di Pisa, e-mail: monta@di.unipi.it
[2] Dip. di Sistemi e Informatica, Università di Firenze, e-mail: semini@dsi.unifi.it

Abstract. We introduce Oikos_adtl, a specification language for distributed systems based on asynchronous communication via remote writings. The language is designed to support the composition of specifications. It allows expressing the global properties of a system in terms of the local properties of the components and of *coordination patterns*.
Oikos_adtl is based on an asynchronous, distributed, temporal logic, which extends Unity to deal with components and events.
We present the specification language and its semantics, introduce a number of compositionality theorems, and discuss some coordination patterns. A fragment of a standard case study is used to validate pragmatically the approach, with respect to expressiveness and work-ability.

1 Introduction

The design of quality software for distributed systems is becoming more and more critical, due to the current impact of software on every technical accomplishment, and the fact that networks pervade any current application. The problem has two facets: the complexity of the systems under development, and the need of continuous update to keep the pace with moving requirements. To avoid old errors and face complexity, we need a design process that helps in getting the *right* system architecture *right* from the outset, rather than rely *only* on afterwards checks. Besides, to cope with change better, the process should naturally support the recording of the complete justification of the development, for future reference.

A design process based on formal refinements, centered on the system architecture, when applied in the early phases of development, would mitigate the problems related to both complexity and change. So, a long term goal of ours is to extend the refinement calculi, traditionally applied to programming in the small [2], to programming in the large. The approach can be effective only if it comes with simple, workable concepts and notations. A key strategy to cope with complexity, also in design, is "divide and conquer": express the global properties of a system in terms of the properties that are local to its components and of *coordination patterns* that make it easier for the designer to *compose* specifications. The other way round, once valuable composition patterns are available, they can be exploited to devise *refinement patterns*, to be used top-down in the design.

The work reported here is a first step towards a distributed refinement calculus that integrates in a natural way local refinements (i.e. within a single component) and coordination patterns. Our previous work [21] can readily be adapted to work locally. The compositionality results of this paper represent the main ingredient to develop useful coordination, and hence refinement, patterns.

To exploit our past experience, we circumscribe our field of application and consider distributed systems that consist of loosely coupled asynchronous long running services, like work-flow systems. These systems are conveniently modeled as multiple data-spaces interacting via remote writings. The multiple data-spaces paradigm is widely regarded as interesting for coordination applications [4, 8, 10]. Brogi and Jacquet [5] argue convincingly for the relevance of asynchronous communications. Still, there is a lack of satisfactory design and analysis techniques for these systems.

We introduce Oikos_adtl, a specification language for distributed systems based on asynchronous communication via remote writings. The language is designed to support the composition of specifications. It allows expressing the global properties of a system in terms of local properties and of *coordination patterns*. The former are properties exposed by a single component, the latter describe the approach taken to control the interactions of the components. Oikos_adtl is based on an asynchronous, distributed, temporal logic, which extends Unity [6] to deal with components and events.

The structure of the paper is the following. Section 2 presents the specification language and its semantics. Section 3 introduces a number of composition theorems, and discusses some coordination patterns. A fragment of a standard case study from the COORDINA working group is used in Section 4 to show the expressiveness of the language and to validate the selected theorems and patterns, pragmatically. Before some concluding remarks, Section 5 discusses related work.

2 Oikos_adtl

The extensions to Unity that allows Oikos_adtl to deal with *events*, are taken from Oikos_tl [21]. This paper works out the extension to distributed components first proposed in [20, 17].

A component has a name, and it is specified by liveness (something good will happen) and safety (nothing bad will happen) properties of the evolution of its state. An event occurs when a given condition is established in a component. Operators CAUSES, NEEDS, and BECAUSE permit to deal with events. The former expresses liveness conditions, the latter express safety conditions: CAUSES relates an event and a condition, and specifies that the occurrence of the event is sufficient to arrive in a state in which the condition holds; NEEDS forces a condition to hold when an event occurs; BECAUSE also relates an event and a condition, but specifies that the condition is a necessary one for the event to occur: the condition must hold somewhere before the event occurs. BECAUSE is weaker than NEEDS, which entails that the conditions it relates are synchronous. For this reason because can be distributed, while needs can be used only locally. Operators leads_to, unless, stable, and inv extend consistently the corresponding operators of Unity.

Components interact via *remote writings*, as in multiple tuple-spaces. They are denoted in Oikos_adtl by expressions like $n : q@m$, meaning that component n has caused condition q to be established in component m.

The specification of a system in Oikos_adtl shapes $\mathcal{M} :\ll \mathcal{F} \gg$, where $\mathcal{M} = M^1, \dots, M^n$ is a list of component names and $\mathcal{F}$ is a set of *properties*, i.e. a set of formulae with the structure given in Table 2. A property F can be *local* or *distributed*. In the first case it refers to a single component, in the second case to a set of components.

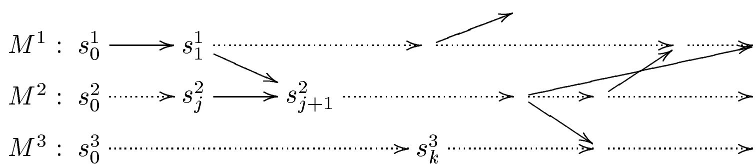
The specification example in Table 1 captures the essence of a very common coordination pattern, namely the *moderated publish-subscribe* pattern. There are $n+1$ components, one of which is the *moderator* M. The other components are interested in publishing ($pub\_int(X)$) and being notified of new items ($notif(X)$). The moderator may reject an item ($rej(X)$), in ways unsaid in this example. The first property defines the liveness in the system: it states that each intention to publish will be honored, either by a rejection or by notifying everybody. The other properties state the most important safety conditions. The second one requires that a notification will occur only if an intention to publish has been manifested by somebody. The third property requires that a rejection can appear in PS^i only if there is no notification for the same item. The fourth one constraints notifications similarly.

Other examples of specifications will be discussed in Sections 3 and 4.

$$
\begin{array}{l}
M,\, PS^1, \dots, PS^n : \ll \quad \text{(for all } i.\text{)} \\
\quad PS^i : pub\_int(X) \text{ CAUSES } (PS^i : rej(X) \vee \bigwedge_j PS^j : notif(X)), \\
\quad PS^i : notif(X) \text{ BECAUSE } (\bigvee_j PS^j : pub\_int(X)), \\
\quad PS^i : rej(X) \text{ NEEDS } \neg notif(X), \qquad PS^i : notif(X) \text{ NEEDS } \neg rej(X) \ \gg
\end{array}
$$

Table 1. A specification example

A system Σ is characterized by a list $\mathcal{M}$ of components names, and a set $\mathcal{T}$ of state transitions, where each transition is associated to a component in $\mathcal{M}$. A computation of a system $\langle \mathcal{M}, \mathcal{T} \rangle$ is an acyclic oriented graph of states like:



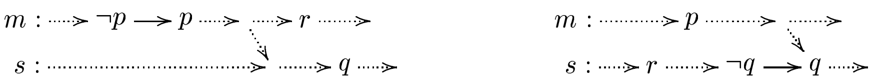
We call s^i_j the j^{th} state of component M^i. A state is a multiset of terms: standard terms p, and terms of the form $p@M$. A computation satisfies:

$$s^i_{j+1} = \begin{cases} t_i(s^i_j), & \text{for some } t_i \in \mathcal{T} \qquad\qquad\qquad \text{or} \\ s^i_j \cup p & \text{and there are } s^k_{l-1},\ s^k_l \text{ s.t. } s^k_l = t_k(s^k_{l-1}) \text{ for some } t_k \in \mathcal{T}, \\ & \text{with } p@M^i \notin s^k_{l-1} \text{ and } p@M^i \in s^k_l. \end{cases}$$

The application of a transition as well as a remote writing define a *next state* relation (plain arrows in the example above). States in our computations are also related by a *future state* relation (dotted arrows in the example above) which is the reflexive and transitive closure of the next state relation. Since the underlying graph is acyclic, the future state relation is antisymmetric and thus a partial order.

We define the parallel composition $\langle \mathcal{M}_1,\ \mathcal{T}_1\rangle \parallel \langle \mathcal{M}_2,\ \mathcal{T}_2\rangle$ of a pair of systems with $\mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset$, as the system $\langle \mathcal{M}_1 \cup \mathcal{M}_2,\ \mathcal{T}_1 \cup \mathcal{T}_2\rangle$.

Oikos_adtl formulae are interpreted on computations. For instance, the following computation, on the left, is a model of $m\!:\!p$ CAUSES $(s\!:\!q \wedge m\!:\!r)$; the one on the right is a model of $s\!:\!q$ BECAUSE $(m\!:\!p \wedge s\!:\!r)$



Formally, given a formula F and a computation $\mathcal{S}$ of $\langle \mathcal{M},\ \mathcal{T}\rangle$, $\mathcal{S}$ is a model of F if and only if: *(a)* $M^i \in \mathcal{M}$ for each sub-formula $M^i : F'$ of F, and *(b)* the states in $\mathcal{S}$ satisfy appropriate conditions given by cases on the structure of F in Table 2.

Finally, a system $\langle \mathcal{M},\ \mathcal{T}\rangle$ satisfies a specification $\mathcal{M}' :\ll \mathcal{F} \gg$ if and only if $\mathcal{M} = \mathcal{M}'$ and all its computations are models of all the formulae in $\mathcal{F}$.

2.1 Composing specifications

How to prove that a single component satisfies its specification is discussed elsewhere [21]. Here we show how to derive the properties satisfied by a larger system from the properties satisfied locally by its components. More in general, we want to derive the specification $\mathcal{M}_1, \mathcal{M}_2 :\ll \mathcal{F} \gg$ of a system $\Sigma_1 \parallel \Sigma_2$ from the specifications $\mathcal{M}_1 :\ll \mathcal{F}_1 \gg$ and $\mathcal{M}_2 :\ll \mathcal{F}_2 \gg$ of Σ_1 and Σ_2, respectively.

We assert:

$\vdash_{\mathcal{M}} \mathcal{F}$	iff, for any $\mathcal{T}$, $\langle \mathcal{M}, \mathcal{T}\rangle$ satisfies $\mathcal{F}$, i.e. iff all its computations are models of all the formulae in $\mathcal{F}$;
$\dfrac{\vdash_{\mathcal{M}} \mathcal{F}}{\vdash_{\mathcal{M}} F}$	iff any $\langle \mathcal{M},\ \mathcal{T}\rangle$ satisfying $\mathcal{F}$, also satisfies F;

Syntax

F	$::=$	$M\colon Local \mid Distr$ with M component name
$Local$	$::=$	$State$ UNLESS $State \mid Atomic$ NEEDS $State$
$Distr$	$::=$	INIT $DiState \mid M\colon Atomic$ BECAUSE $DiState \mid$ $M\colon State$ LEADS_TO $DiState \mid M\colon Atomic$ CAUSES $DiState$
$DiState$	$::=$	$M\colon State \mid DiState \wedge DiState \mid DiState \vee DiState$
$State$	$::=$	$Atomic \mid \neg State \mid State \wedge State$
$Atomic$	$::=$	$n * p \mid p@M$ with $n \in \mathbb{N}$, p a term

Two local operators are derived: $M :$ STABLE $A \stackrel{def}{=} M : A$ UNLESS $false$

$M :$ INV $A \stackrel{def}{=}$ INIT $M : A \wedge M :$ STABLE A

Semantics

$M^i : A$ UNLESS B	:	$\forall j \geq 0.\ s^i_j \models A \wedge \neg B \ \rightarrow\ s^i_{j+1} \models A \vee B$
$M^i : A$ NEEDS B	:	$\forall j \geq 0.\ s^i_j \models A! \ \rightarrow\ s^i_j \models B$
$M^i : A$ BECAUSE DS	:	$(s^i_0 \models A \ \rightarrow\ s^i_0 \models$ BEGIN $DS)$ and $(\forall k > 0.\ s^i_k \models A! \ \rightarrow\ s^i_k \models$ PAST $DS)$
$M^i : A$ LEADS_TO DS	:	$\forall k \geq 0.\ s^i_k \models A \ \rightarrow\ s^i_k \models$ FUT DS
$M^i : A$ CAUSES DS	:	$\forall k \geq 0.\ s^i_k \models A! \ \rightarrow\ s^i_k \models$ FUT DS
INIT DS	:	$s^i_0 \models$ BEGIN DS for some i

Where:

$s^i_j \models n * p$	iff	s^i_j contains at least n occurrences of p
$s^i_j \models p@M^k$	iff	$(\forall l \geq j.\ p@M^k \in s^i_l)$ and $[(M^k \in \Sigma \ \wedge\ s^i_{j-1} \not\models p@M^k) \ \rightarrow\ (\exists s^k_l > s^i_j.\ s^k_l \models p)]$
$s^i_j \models \neg A$	iff	$s^i_j \not\models A$
$s^i_j \models A \wedge B$	iff	$s^i_j \models A$ and $s^i_j \models B$
$s^i_j \models A!$	iff	$(s^i_j \models A$ and $s^i_{j-1} \not\models A)$ or $(j = 0$ and $s^i_0 \models A)$
$s^i_0 \models$ BEGIN $M^k : A$	iff	$s^k_0 \models A$
$s^i_j \models$ FUT $M^k : A$	iff	$\exists s^k_l \geq s^i_j.\ s^k_l \models A$
$s^i_j \models$ PAST $M^k : A$	iff	$\exists s^k_l \leq s^i_j.\ s^k_l \models A$
$s^i_j \models$ OP $DS \wedge DS'$	iff	$s^i_j \models$ OP $DS \ \wedge\ $ OP DS' for OP $\in \{$ BEGIN, FUT, PAST $\}$
$s^i_j \models$ OP $DS \vee DS'$	iff	$s^i_j \models$ OP $DS \ \vee\ $ OP DS' for OP $\in \{$ BEGIN, FUT, PAST $\}$

Table 2. Oikos_adtl syntax and semantics. The semantics defines the conditions a computation $\mathcal{S}$ has to verify to be a model for F, given by cases on the structure of formula F. Note that $p@M^k$ holds if p *has been* sent to M^k, i.e., it holds in a state s^i_j whenever a state $s^i_k < s^i_j$ exists in which the remote writing has been performed.

$\dfrac{\vdash_{\mathcal{M}_1} \mathcal{F}_1}{\vdash_{\mathcal{M}_1,\mathcal{M}_2} F}$ iff, for any system $\langle \mathcal{M}_1,\, \mathcal{T}_1 \rangle$ satisfying $\mathcal{F}_1$ and for any $\mathcal{T}_2$, $\langle \mathcal{M}_1 \cup \mathcal{M}_2,\, \mathcal{T}_1 \cup \mathcal{T}_2 \rangle$ satisfies F;

$\dfrac{\vdash_{\mathcal{M}_1} \mathcal{F}_1 \quad \vdash_{\mathcal{M}_2} \mathcal{F}_2}{\vdash_{\mathcal{M}_1,\mathcal{M}_2} F}$ iff, for any pair $\langle \mathcal{M}_1,\, \mathcal{T}_1 \rangle$, $\langle \mathcal{M}_2,\, \mathcal{T}_2 \rangle$, satisfying $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively, $\langle \mathcal{M}_1 \cup \mathcal{M}_2,\, \mathcal{T}_1 \cup \mathcal{T}_2 \rangle$ satisfies F;

Thus, given Σ_1 and Σ_2 satisfying $\mathcal{M}_1 :\ll \mathcal{F}_1 \gg$ and $\mathcal{M}_2 :\ll \mathcal{F}_2 \gg$ respectively, $\Sigma_1 \parallel \Sigma_2$ satisfies $\mathcal{M}_1, \mathcal{M}_2 :\ll \mathcal{F} \gg$ iff every $F \in \mathcal{F}$ satisfies the last condition above.

2.2 Composition theorems

In this section we discuss some useful theorems, that can be proved exploiting the semantics in Table 2 and the definitions in the previous subsection. We distinguish liveness (Table 3, 1–6) and safety (Table 3, 7–15).

(1) *Reliable Communication*: this theorem characterizes remote writings, linking the sufficient cause in the sender to the effect in the recipient.
(2) *Preserve Causes*: the essential points here are: *a)* the syntax for CAUSES restricts A to be positive, and *b)* the other components can only add to M. Therefore, the sufficient condition A in M cannot be falsified by the new component N.
(3) *Preserve Leads_To*: a similar argument applies to the condition in M leading to B. The side condition restricts applicability to positive conditions in M.
(4) *Preserve Negative Leads_to*: the first premise ensures that in M, A leads to C in absence of condition B. But, being unconstrained in this respect, N might send B to M at any time, and potentially forbid the needed transitions to fire. Hence the second premise.
(5) *Causes Weak Transitivity*: CAUSES is only weakly transitive. Indeed, consider a computation in which first B becomes true, then C appears and disappears before A becomes true, while B keeps holding: it satisfies the first two premises, but not the consequence. Hence the last premise. Note also that in a transitivity chain, only the last condition has to be stable.
(6) *Causes/Leads_To*: there will always be a state in which C holds after B holds, due to the event A, hence the consequence.

(7) *Committing Communication*: something sent cannot be retracted: the sender has committed itself.
(8) *Drop Component*: eliminating M entails that the events in N that are due to remote writings by M, no longer occur. Thus, even less "wrong things can happen" in N.
(9) *Preserve Unless*: the added component cannot falsify A, restricted to be positive by the side condition.
(10) *Preserve Needs*: the argument is similar to Preserve Leads_To. N must be constrained so that it does not interfere with the computations of the components in M, that will not establish A in the absence of B.

Liveness

(1) $\vdash_{M,N} \;\; M : B@N \text{ CAUSES } N : B$

(2) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ CAUSES } B}{\vdash_{N,\mathcal{M}} \;\; M : A \text{ CAUSES } B} \qquad M \neq N$$

(3) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ LEADS\_TO } B}{\vdash_{N,\mathcal{M}} \;\; M : A \text{ LEADS\_TO } B} \qquad A \text{ conjunction of atomic formulae, } M \neq N$$

(4) $$\frac{\vdash_{\mathcal{M}} \;\; (M : A \wedge \neg B) \text{ LEADS\_TO } C \qquad \vdash_N N : \text{ INV } \neg B@M}{\vdash_{N,\mathcal{M}} \;\; (M : A \wedge \neg B) \text{ LEADS\_TO } C} \qquad M \neq N$$

(5) $$\frac{\vdash_{\mathcal{M}} M : A \text{ CAUSES } N : B \quad \vdash_{\mathcal{M}} N : B \text{ CAUSES } O : C \quad \vdash_{\mathcal{M}} O : \text{ STABLE } C}{\vdash_{\mathcal{M}} M : A \text{ CAUSES } O : C}$$

(6) $$\frac{\vdash_{\mathcal{M}} M : A \text{ CAUSES } N : B \quad \vdash_{\mathcal{M}} N : B \text{ LEADS\_TO } O : C}{\vdash_{\mathcal{M}} \;\; M : A \text{ CAUSES } O : C}$$

Safety

(7) $\vdash_{\mathcal{M}} M : \text{ STABLE } p@N$

(8) $$\frac{\vdash_{M,\mathcal{M}} \;\; N : F}{\vdash_{\mathcal{M}} \;\; N : F} \qquad \text{for } F \text{ any safety property, } M \neq N$$

(9) $$\frac{\vdash_{\mathcal{M}} \;\; N : A \text{ UNLESS } B}{\vdash_{M,\mathcal{M}} \;\; N : A \text{ UNLESS } B} \qquad A \text{ conjunction of atomic formulae, } M \neq N$$

(10) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ NEEDS } B \qquad \vdash_N \;\; N : \text{ INV } \neg A@M}{\vdash_{N,\mathcal{M}} \;\; M : A \text{ NEEDS } B}$$

(11) $$\frac{\begin{array}{ll} \vdash_N \;\; N : A \text{ NEEDS } false & \vdash_M \;\; M : B@N \text{ NEEDS } \neg A@N \\ \vdash_N \;\; N : B \text{ NEEDS } false & \vdash_M \;\; M : A@N \text{ NEEDS } \neg B@N \end{array}}{\vdash_{M,N} \;\; N : A \text{ NEEDS } \neg B}$$

(12) $$\frac{\begin{array}{ll} \vdash_N N : B \text{ NEEDS } false & \vdash_M M : B@N \text{ NEEDS } C \\ \vdash_N N : C@M \text{ NEEDS } A & \vdash_M M : C \text{ NEEDS } false \end{array}}{\vdash_{M,N} \;\; N : A \text{ NEEDS } \neg B}$$

(13) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ NEEDS } B}{\vdash_{\mathcal{M}} \;\; M : A \text{ BECAUSE } M : B}$$

(14) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ BECAUSE } M : B}{\vdash_{N,\mathcal{M}} \;\; M : A \text{ BECAUSE } (M : B \vee N : A@M)}$$

(15) $$\frac{\vdash_{\mathcal{M}} \;\; M : A \text{ BECAUSE } N : B \quad \vdash_{\mathcal{M}} \;\; N : B \text{ BECAUSE } O : C}{\vdash_{N,\mathcal{M}} \;\; M : A \text{ BECAUSE } O : C}$$

Table 3. Composition Theorems

(11) *Preserve Mutual Exclusion*: the premises guarantee that N cannot generate neither A nor B locally, and that M can send at most one of them. Stability of $p@N$ is crucial here.
(12) *Conditioning remote writing*: the constraints on B in N, ensure that B can appear in N only from the outside. The constraints on C in M ensure that M can write B into N only if triggered from the outside by C. Since $C@N$ is also constrained in N, when combining N and M we have that, in N, B cannot hold when A appears. That is, B can be sent only after the event A.
(13) *Because Introduction*: BECAUSE is weaker than NEEDS.
(14) *Preserve Because*: the new component can be an unforeseen cause for A.
(15) *Because Transitivity*: if B is necessary for event A, B must hold before or when A becomes true; if also C is necessary for event B, C must hold before or when B becomes true. Therefore, C is necessary for event A, i.e. C holds before or when A becomes true.

3 Coordination patterns

To specify and prove the global coordination properties of a distributed system is often a complex task. A coordination pattern defines a specification schema for a set of components, and derives global properties of their parallel composition. Once valuable composition patterns are available, they can be used to simplify the design process. We introduce two such patterns, *broadcast* and *moderated publish-subscribe*.

3.1 Broadcast

This pattern consists of the parallel composition of a sender P and n receivers S^i $(i \in \{1, \ldots, n\})$ specified, respectively, by:

$$P : \ll\ P : req(X) \text{ CAUSES } P : \textstyle\bigwedge_j notif(X)@S^j,$$

$$\text{(for all } j\text{) } P : notif(X)@S^j \text{ BECAUSE } P : req(X)\ \gg$$

$$S^i : \ll S^i : notif(X) \text{ NEEDS } false, \quad S^i : \text{ STABLE } notif(X),$$

$$\text{(for all } j\text{) } S^i : notif(X)@S^j \text{ NEEDS } false \gg$$

The sender component has to notify all the receivers of any new item X it is requested to publish. Besides, it sends a notification only upon request. Nothing is said here on the request originator. Receivers cannot produce or send a notification, and notifications are stable. This condition is technically needed to apply *Weak Transitivity* to prove the property below. Semantically, $notif(X)$ should be interpreted as “X has been notified”, which cannot be falsified after the event. Composition theorems permit to derive that the parallel composition of sender and receivers has the property that receivers will be notified of all the items following a request, and only of those:

$$P, S^1, \ldots, S^n : \ll \; P : req(X) \text{ CAUSES } \bigwedge_i S^i : notif(X),$$
$$\text{(for all } i\text{) } S^i : notif(X) \text{ BECAUSE } P : req(X) \; \gg$$

3.2 Moderated publish–subscribe

We now reconsider the example of a system composed of a moderator and n components PS^i (publish–subscribers, see Sect. 2), and provide their local specifications:

$$\begin{array}{ll}
M : \ll & \text{(for all } j\text{)} \\
& M : req(X, PS^j) \text{ CAUSES } M : rej(X)@PS^j \vee \bigwedge_i notif(X)@PS^i, \\
& M : notif(X)@PS^j \text{ NEEDS } \bigvee_i (req(X, PS^i) \wedge \neg rej(X)@PS^i), \\
& M : 2 * notif(X)@PS^j \text{ NEEDS } false, \\
& M : rej(X)@PS^j \text{ NEEDS } (req(X, PS^j) \wedge \bigwedge_i \neg notif(X)@PS^i) \; \gg \\
PS^i : \ll & PS^i : pub_int(X) \text{ CAUSES } PS^i : req(X, PS^i)@M, \\
& PS^i : req(X, PS^i)@M \text{ BECAUSE } PS^i : pub_int(X), \\
& PS^i : 2 * req(X, PS^i)@M \text{ NEEDS } false, \\
& PS^i : \; notif(X) \text{ NEEDS } false, \quad PS^i : \; \text{STABLE } notif(X), \\
& PS^i : \; rej(X) \text{ NEEDS } false, \quad PS^i : \; \text{STABLE } rej(X), \\
& \text{(for all } j\text{) } PS^i : notif(X)@PS^j \text{ NEEDS } false, \\
& \text{(for all } j\text{) } PS^i : rej(X)@PS^j \text{ NEEDS } false \; \gg
\end{array}$$

The moderator reacts to a publication request $req(X, PS^j)$ by rejecting it and informing PS^j of the rejection, or by notifying X to everybody. The moderator sends a notification only in the presence of the corresponding request and if the request was not already rejected. Two identical notifications cannot be sent. Finally, the moderator rejects X only in the presence of a publication request for X and if X has not already been notified. The publish–subscribers send a publication request to the moderator for every new publication intention (and only for those). They do not send more than a request for the same item, nor produce or send any notification or rejection. Finally, they keep notifications and rejections.

It can be proved that the composed system satisfies the specification anticipated in Table 1. To prove liveness, apply *Causes Weak Transitivity*, to prove safety apply *Because Transitivity* and *Introduction* on one side, and *Preserve Mutual Exclusion* on the other.

4 Case study

The problem we address in this section is a fragment of the "Conference Management on the Internet"' case study from the COORDINA working group [9]. The case study addresses the coordination of all the activities and people involved in the preparation of a conference proceedings volume, starting from the submission of the drafts. The geographical distribution of the partners, and the use of e-mail to carry over their tasks, are naturally modelled in the asynchronous distributed paradigm used in Oikos_adtl.

We consider a fragment dealing with the allocations of the submitted papers to the PC members, for reviewing. We assume that the task is accomplished in two phases and that m reviewers are needed for each paper. In phase one, each PC member can express some preferences. If a preferred paper is assigned, all the other PC members are notified. To be fair, a maximum of n papers can be assigned to each PC member in this phase: a preference is rejected if the PC member has already been assigned n papers, or if the selected paper is already assigned to m reviewers. Moreover, a paper cannot be assigned twice to the same PC member. In phase two, all the papers not yet assigned are fairly distributed by the PC chair. This phase can be naturally modelled by instantiating the broadcast coordination pattern, with the PC chair as the sender and the PC members as the receivers.

The model is realistic only if no PC member is the author of a submitted paper. However, the changes to the specification needed to accommodate such a case consist in strengthening some conditions.

Phase one can be modelled by instantiating the moderated publish–subscribe pattern, with the PC members as the publisher–subscribers P^i, and a coordination component as the moderator M. The instance renames the components in the pattern. A new property is added, expressing the causes for a rejection, which are specific for this case: see the last line of the moderator specification that follows. The specification of the PC members is omitted, since it is just a renaming of the pattern.

$$
\begin{array}{l}
M : \ll \quad \text{(for all } j \text{ and } k\text{)} \\
M : req(X, P^j) \text{ CAUSES } M : rej(X)@P^j \vee \bigwedge_i ass(X, P^j)@P^i \\
M : ass(X, P^k)@P^j \text{ NEEDS } \bigvee_i (req(X, P^i) \wedge \neg rej(X)@P^i \\
M : 2 * ass(X, P^k)@P^j \text{ NEEDS } false \\
M : rej(X)@P^j \text{ NEEDS } (req(X, P^j) \wedge \bigwedge_i \neg ass(X, P^j)@P^i) \\
M : rej(X)@P^j \text{ NEEDS } \bigvee_i (m * ass(X, \_)@P^i \ \vee \ n * ass(\_, P^j)@P^i) \ \gg
\end{array}
$$

Following the pattern , we derive global properties holding in phase one (see Table 1):

$M,\, P^1, \ldots, P^n : \ll$ (for all j and k.)

$P^j : pub\_int(X)$ CAUSES $(P^j : rej(X) \vee \bigwedge_i P^i : ass(X, P^j))$

$P^j : ass(X, P^k)$ BECAUSE $(\bigvee_i P^i : pub\_int(X))$

$P^j : rej(X)$ NEEDS $\neg ass(X, P^j)),$ $\quad P^j : ass(X, P^j)$ NEEDS $\neg rej(X))$

$P^j : rej(X)$ BECAUSE $M : \bigvee_i (m * ass(X, \_)@P^i \;\vee\; n * ass(\_, P^j)@P^i) \gg$

5 Related Work

Ciancarini et al. [8] present a formal method for construction and verification of open systems whose architecture is modeled by multiple data-spaces. The approach features a theoretical coordination model, PoliS, and rules to translate PoliS specifications in Lamport's TLA [15]. The advantage is that tools available for TLA, like TLP, can be exploited to reason about PoliS specification, albeit at the price of the somewhat cumbersome translation. To our taste, however, PoliS, in line with Lamport's original goals for TLA, is too operational as a specification language. Besides, the approach does not show the desired properties of composition: the proofs take into account the evolution of the system as a whole, rather than decomposing the task into simpler proofs, dealing with local properties and their composition, separately. Finally, it has to be noted that PoliS deals with open systems, while Oikos_adtl does not.

Compositional issues for multiple tuple spaces were discussed by Chen and Montangero [7], albeit in an operational setting.

Goeman et al. introduce the ImpUnity framework [13] to reason on coordination programs. These are written in ImpUnity, a programming language based on Unity, action systems, and the tuple space model of communication. The language permits to declare private variables in programs, and these declarations are used to refine the programs compositionally. As for Swarm and for our approach, program properties are derived using assertion on the form of Hoare triples, but no formal definition of the axiomatic semantics is given.

Compositionality results for programs based on tuple spaces have been proposed also by Jacquet and De Bosschere. In [14], they discuss a methodology based on a logical approach. Program properties are expressed in Unity, properties of the composition of programs are obtained with a rely–guarantee reasoning similar to the one proposed in [1]. However, they only consider compositions of programs acting on a single component.

The main difference between our approach and other proposals of distributed logic relies in the "amount" of knowledge a component can have of the others components in the system. In the logic underlying TROLL [11] non–local assertion are possible only at communication time. In the n–agent logic [16, 19] non–local knowledge is gained at communication time and then kept. In either cases, communications are synchronous handshakings. Besides, non–local assertions can include remote properties which have not been caused, directly or indirectly, by the component in which the assertion is made. We think that Oikos_adtl conservative approach in this respect is more realistic. An interesting issue would be to compare the different approaches by expressing their modalities in terms of the knowledge modalities of [12], as it is partly done in [18].

6 Conclusions

We introduced Oikos_adtl, a specification language for distributed systems with components communicating via asynchronous remote writings. The language is based on a distributed temporal logic, and extends smoothly Oikos_tl, which has been used to specify single components viewed as tuple spaces [21]. We presented a semantic definition of the language, and discussed a number of composition theorems, that can be used to justify coordination patterns. This work provides the basis of a refinement calculus to design distributed systems. To sketch how to proceed towards this goal, we glimpse specification refinement. A specification can be refined either by decomposing a system in two or more subsystems, or by refining its properties $\mathcal{F}$, finding $\mathcal{F}'$ such that $\vdash_{\mathcal{M}} \mathcal{F}'$ justifies $\vdash_{\mathcal{M}} F$, for all F in $\mathcal{F}$. The composition patterns we discussed above, provide examples of decomposition patterns, that in general shape

$$M^1, \ldots, M^n :\ll \mathcal{F} \gg$$

$$\sqsubseteq \qquad \frac{\vdash_{M^1, \ldots, M^i} \mathcal{F}' \quad \vdash_{M^{i+1}, \ldots, M^n} \mathcal{F}''}{\vdash_{M^1, \ldots, M^n} \mathcal{F}}$$

$$M^1, \ldots, M^i :\ll \mathcal{F}' \gg \ \| \ M^{i+1}, \ldots, M^n :\ll \mathcal{F}'' \gg$$

The refinement patterns have to be proved once for all using the composition theorems: to apply the pattern, the designer has only to verify that the premises hold.

To validate the approach we plan to deal with more substantial case studies, like the coordination model for distributed control systems presented in [3]. Besides, a challenging and interesting issue is the extension to cope with open and mobile systems.

References

1. M. Abadi and L. Lamport. Composing Specifications. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, 1989.
2. R.J.R. Back and J. von Wright. *Refinement Calculus. A Systematic Introduction.* Graduate texts in computer science. Springer-Verlag, 1998.

3. M. M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proc. ACM Symp. on Applied Computing SAC'98*, pages 159–168, Atlanta, 1998. ACM press.
4. K. De Bosschere and J.-M. Jacquet. μ^2 Log: Towards Remote Coordination. In P. Ciancarini and C. Hankin, editors, *Proc. Coordination Languages and Models, First International Conference, COORDINATION 96*, volume 1061 of *Lecture Notes in Computer Science*, pages 142–159, Cesena, April 1996. Springer-Verlag.
5. A. Brogi and J.-M. Jacquet. Modelling Coordination via Asynchronous Communication. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282 of *Lecture Notes in Computer Science*, pages 238–255, Berlin, Germany, September 1997. Springer-Verlag, Berlin.
6. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading Mass., 1988.
7. X.J. Chen and C. Montangero. Compositional Refinements of Multiple Blackboard Systems. *Acta Informatica*, 32(5):415–458, 1995.
8. P. Ciancarini, M. Mazza, and L. Pazzaglia. A logic for a coordination model with multiple spaces. *Science of Computer Programming*, 31(2–3):231–261, July 1998.
9. P. Ciancarini, O. Nierstrasz, and R. Tolksdorf. A case study in coordination: Conference management on the internet. Available at: http://malvasia.di.fct.unl.pt/activity/coordina/working/case–studies, 1998.
10. R. DeNicola, G. Ferrari, and R. Pugliese. A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
11. H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for specifying concurrent information systems. In J. Chomicki and G. Saake, editors, *Logic for Databases and Information Syustems*, pages 167–198. Kluver Academic Publishers, 1998.
12. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge.* MIT Press, 1995.
13. H. Goeman, J.N. Kok, K. Sere, and R. Udkin. Coordination in the ImpUnity Framework. *Science of Computer Programming*, 31(2–3):313–334, Jul. 1998.
14. J.-M. Jacquet and K. De Bosschere. On Composing Concurrent Logic Processes. In L. Sterling, editor, *Proc. 12^{th} Int'l Conf. on Logic Programming, ICLP'95*, pages 531–545, Tokio, 1995. The MIT Press, Cambridge, Mass.
15. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
16. K. Lodaya, R. Parikh, R. Ramanujam, and P.S. Thiagarajan. A Logical Study of Distributed Transition Systems. *Information and Computation*, 119(1):91–118, 1995.
17. C. Montangero and L. Semini. Refining by Architectural Styles or Architecting by Refinements. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *2nd International Software Architecture Workshop, Proceedings of the SIGSOFT '96 Workshops, Part 1*, pages 76–79, San Francisco, CA, Oct 1996. ACM Press.
18. R. Ramanujam. Local knowledge assertions in a changing world. In *Proc. TARK IV*, pages 1–17. Morgan Kaufmann, 1996.
19. R. Ramanujam. Locally linear time temporal logic. In *Proc.* 11^{th} *IEEE Symp. on Logic In Computer Science*, pages 118–127. IEEE Computer Society, 1996.
20. L. Semini. *Refinement in Tuple Space Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996. TD 10–96.
21. L. Semini and C. Montangero. A Refinement Calculus for Tuple Spaces. To appear in Science of Computer Programming, available at ftp://ftp.di.unipi.it/pub/Papers/montangero/Oikos_tl.ps.