On Methodologies for Coordinating Programs

J.-M. Jacquet and I. Linden
Institute of Informatics
University of Namur
Belgium

Abstract This paper discusses a methodological approach for coordinating programs, based on the composition of specifications written in a UNITY-like temporal logic.

Keywords: Coordination, methodology

1 Introduction

Whereas old information systems were classically designed as isolated pieces of code, modern computer systems consist of large numbers of software components that interact one another. The process of software construction is consequently more and more centered on the composition of generic existing packages to construct complex systems. This design change is even accelerated by the rapid expansion of computer networks, which obviously call for integrating and coordinating heterogeneous components, which rely on different computational models and are physically distributed on the net.

A central problem in this approach is to check that data which is assumed to be communicated when pieces of code are constructed is indeed provided by the execution of the whole system composed of all these pieces of code. Curiously enough, although many research efforts have focused on the design of coordination languages and models and have argued that problems of some classes are solved in an easier way in some languages or models, not many efforts have been devoted to programming methodologies and in particular to methods for supporting the above checking problem.

This paper aims at proposing a methodology along this line. It is based on a Unity-like temporal logic, which turns out to be powerful and to provide a practical approach.

Our work has similarities with the work of Abadi and Lamport [1]. However, our contribution in this paper is to provide a simpler calculus by taking profit of the context of coordination and of temporal logic formulae, more abstract than the traces used in [1].

The article [2] proposes a compositional notion of refinement that can be used to reason about coordination of Gamma programs. That work is orthogonal to ours in that we compose specifications written in temporal logic rather than refinements.

The paper [3] presents an extension of the Unity framework. It contains several program structuring mechanisms and puts special emphasis on compositional refinement of both specifications and programs. Another extension is used in our work: it is based on the introduction of agents and, as may be appreciated by the reader, calls for another composition theorem.

A specification language based on tuple-spaces is studied in [4]. It is there argued that this language is flexible enough to specify architectures containing mobile components and give formalization of some common mobility paradigms. However, no counterpart is proposed for the composition of specifications, as we do.

Oikos_single, a specification language for distributed systems based on asynchronous communication via remote writings is introduced in [5]. It uses an extension of Unity to deal with
components and events. A number of compositionality theorems are presented. A continuation of that work for modeling network-aware applications is presented in [6]. In contrast to these pieces of work, we use another extension of logic based on agents, which calls for other forms of compositionality theorems.

Finally, [7] presents a component-based formal approach to the design of distributed systems based on the coordination of the interaction and on the flow of control using graphical notation, refinement and guarded composition. Their work is based on the action system formalism ([8]) instead of temporal logic, as we use.

2 The \( L_\Psi \) language

2.1 Syntax

For the purpose of illustration, we shall use an abstract coordination language, called \( L_\Psi \). It is designed so as to incorporate the main coordination mechanisms. It therefore includes the basic Linda’s \( \text{out}, \text{in}, \text{and rd} \) primitives, for putting a tuple on a shared space (subsequently referred to as store), getting it and checking its presence, respectively. \( L_\Psi \) also includes sequential and parallel composition operators as well as a choice operator in the style of CCS [9]. However, for simplicity, only finite processes are treated here, under the observation that infinite processes can be handled by extending the results of this paper in the classical way, exemplified, for instance, in [10].

Special care is taken for the matching mechanism. We shall extend it so that tuples may specify partial information only. Technically, this leads to the introduction of \( \Psi \)-terms, defined as follows.

**Definition 1** Let \( \text{Scvar} \) be a denumerably infinite set of communication variables and \( \text{Sf} \) a denumerably infinite set of functor names, each one coupled to an arity. Such an association is typically written as \( f/n \) where \( f \) is the functor name and \( n \) is its associated arity. Assume that the sets \( \text{Scvar} \) and \( \text{Sf} \) are disjoint.

As usual, functors of arity 0 are called constants. Their set is subsequently denoted as \( \text{Sconst} \).

**Definition 2** A \( \Psi \)-term is a construct of the form \( f(t_1 = \text{value}_1, \ldots, t_m = \text{value}_m) \) where (i) \( f/n \) is a functor such that \( m \leq n \), (ii) the \( \text{value}_i \)'s are distinct constants, (iii) \( \text{value}_i \) denotes an integer, a string of characters, a \( \Psi \)-term or a communication variable, and (iv) any communication variable appears at most once in the \( \Psi \)-term. A \( \Psi \)-term is said to be closed if it contains no communication variable. The set of \( \Psi \)-terms is subsequently referred to as \( \text{Spterm} \). The set of closed \( \Psi \)-terms is denoted as \( \text{Scpterm} \) in the following.

Matching will lead us subsequently to compare two \( \Psi \)-terms. To that end, we introduce the notion of correspondence.

**Definition 3** Let \( \Psi_1 = f(t_1 = \text{value}_1, \ldots, t_n = \text{value}_n) \) and \( \Psi_2 = f'(t'_1 = \text{value}'_1, \ldots, t'_m = \text{value}'_m) \) be two \( \Psi \)-terms. We say that \( \Psi_1 \) corresponds to \( \Psi_2 \) iff

1. \( f \) and \( f' \) are identical functors with same arities;

2. The set \( \{ t_i : i = 1, \ldots, n \} \) of the items of \( \Psi_1 \) is a subset of \( \{ t'_i : i = 1, \ldots, m \} \) the set of the items of \( \Psi_2 \);

3. for any \( i \) such that \( \text{value}_i \) is an integer or a string of characters, if \( t_i = t'_i \) then \( \text{value}_i = \text{value}'_i \);

4. for any \( i \) such that \( \text{value}_i \) is a \( \Psi \)-term, if \( t_i = t'_i \) then \( \text{value}_i \) corresponds to \( \text{value}'_i \).

Note that, when \( \Psi_2 \) is closed, the correspondence of \( \Psi_1 \) with respect to \( \Psi_2 \) amounts to the existence of a set of values for the communication variables of \( \Psi_1 \) such that \( \Psi_1 \) becomes a subterm of \( \Psi_2 \). If this is the case, we shall subsequently denote this property by \( \Psi_1 \bowtie \Psi_2 \) and by \( \text{bind}(\Psi_1 \bowtie \Psi_2) \) the binding of values to variables of \( \Psi_1 \) whose application to \( \Psi_1 \) returns a closed \( \Psi \)-term corresponding to \( \Psi_2 \). Moreover, we shall denote by \( \text{Sbind} \) the set of all the possibly partial bindings of values to variables.

Composition can be defined on \( \text{Sbind} \) as follows.
Definition 4 For any θ, µ elements of Sbind, define θµ as the following function of Svar → (N ∪ Sstring ∪ Scerpt ∪ {⊥}): 
\[
θµ(x) = \begin{cases} 
\mu(x) & \text{if } \mu(x) \neq ⊥ \\
θ(x) & \text{if } \mu(x) = ⊥ 
\end{cases}
\]

We are now in a position to define the language LΨ. According to the philosophy of coordination languages, the LΨ language focuses on interaction and communication. Other languages have to be used to specify the computations. Moreover, a means must be provided to interface the two aspects: interactions/communications and computations. This is subsequently achieved by communication variables used both in the communication primitives and the language used to code the computations. Hence, we shall subsequently assume the existence of a set of instructions Sinstr of some programming language. These instructions can be of any kind and of any paradigm; the only constraint is that they cannot directly interact with the shared space and cannot modify values assigned to variables.

The language LΨ is formally defined by the following grammar.

Definition 5 Let Ψ be a Ψ-term, Ψc be a closed Ψ-term and I denote an instruction of Sinstr. The language LΨ is the set of agents A defined by the following grammar. On the point of terminology, the constructs c are subsequently called communication actions.

\[
c ::= \text{tell}(ψc) | \text{nask}(ψ) | \text{ask}(ψ) | \text{get}(ψ) \\
A ::= c | A ; A | A | A | A + A | I
\]

2.2 Operational semantics

2.2.1 Configurations 

LΨ computations may be modelled by a transition system written in Plotkin’s style. To easily express termination, we shall introduce particular configurations composed of a special terminating symbol E together with a shared space and a binding for communication variables. For uniformity purposes, we shall abuse language and qualify E as an agent.

Definition 6 Define the extended set of agents Seagent by the following grammar
\[
Ae ::= E | c | A ; A | A | A | A + A | I
\]

Moreover, we shall subsequently assert that the structure (Seagent, E, ; , ||) is a bimonoid and simplify elements of LΨ accordingly.

Definition 7 Define Sstore as the set of finite multisets of closed Ψ-terms. Moreover, define the set of situations Ssit as the set Sbind × Sstore.

Definition 8 Define the set of configurations Sconf as Seagent × Ssit. Configurations are denoted as \(\langle A \mid (θ, σ)\rangle\), where A is an (extended) agent and (θ, σ) is a situation.

2.2.2 Transition rules

The transition rules defining the operational semantics of the language are reported in Figure 1. They are based on the consideration of two points.

On the one hand, instructions of Sinstr may consult and use the values of communication variables. However, it is possible that their actual execution depends on these values. The predicate executable is introduced to model the possible non execution of instructions. For any \(I ∈ Sinstr\) and any \(θ ∈ Sbind\), executable(I, θ) is true iff I can be executed on θ. Note that, in view of the coordination philosophy, the execution of I cannot depend on the contents of the shared space since it cannot access it. Moreover, the result of the execution cannot modify the contents of this shared space and of the bindings reported by θ. Hence, the execution of any instruction is of no consequence for our semantics.

On the other hand, in order to keep the communication between agents in a pure form through the shared space and thus in order to avoid a side communication of agents by means of communication variables, we impose that agents obtained by parallel composition are formed from components which do not share communication variables. The following \(Var\) function helps to grasp this idea technically.
3 Composing specifications

3.1 A programming logic

Instead of describing the behavior of agents in terms of operational transitions, we now propose to describe them by means of specifications, which essentially consist of first-order formulae of some temporal logic. To that end, two kinds of actors need first to be distinguished: the agent under consideration and its environment. Since the environment is, in general, composed of several agents, it is technically convenient to consider the two actors

\[\exists \Psi \subseteq \mathcal{P}(\text{Scvar}) \quad \forall \alpha \subseteq \mathcal{P}(\text{Scvar}) \quad \alpha \neq \emptyset \quad \text{as sets of agents. In the following, we shall assume a denumerably infinite set } \text{Sag} \text{ of agents and denote by } \mathcal{P}_{\text{ns}}(\text{Sag}) \text{ the set of non-empty and strict subsets of } \text{Sag}. \text{ Such subsets are typically denoted by the letter } \alpha, \text{ possibly subscripted, and their complement in } \text{Sag} \text{ is denoted by } \bar{\alpha}. \]

The resulting transition system is in Figure 2.

\textbf{Definition 10} Let } \alpha \text{ be a set of agents of } \mathcal{P}_{\text{ns}}(\text{Sag}). \text{ The transition relation is defined as the smallest relation of } (\text{Seagent } \times \text{Ssit}) \times \mathcal{P}_{\text{ns}}(\text{Sag}) \times (\text{Seagent } \times \text{Ssit}) \text{ satisfying the rules of figure 2. As usual, the more suggestive no-
tation $C_1 \not\rightarrow C_2$ is employed instead of $(C_1, \alpha, C_2) \in \rightarrow$. It is also understood that $A \parallel E, E \parallel A$ and $E; A$ are rewritten as $A$.

The transition relation induces an operational semantics in a very natural way. Before defining it, we first introduce the notion of computational histories.

**Definition 11**

1. Situations of $S_{sit}$ are typically denoted by the string $ss$ possibly super- or subscripted.

2. Define the set of computational histories $S_{Echist}$ as the set of (possibly infinite) sequences of the form $ss_1 \overset{\alpha_1}{\rightarrow} ss_2 \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} ss_n \overset{\alpha_n}{\rightarrow} \ldots$

   with $ss_1, ss_2, \ldots, ss_n \in S_{sit}$, $\alpha_1, \alpha_2, \ldots, \alpha_n \in P_{ns}(Sag)$.

3. For any history $h$ of $S_{Echist}$, we shall denote by
   
   (a) $h[n]$, the prefix of $h$ of length $n$ : $ss_1 \overset{\alpha_1}{\rightarrow} ss_2 \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} ss_n$ ;
   
   (b) $\text{length}(h)$, the length of $h$;
   
   (c) $h_{k,s}$, the $k^{th}$ situation of $h$;
   
   (d) $h_{k,a}$, the agent set $\alpha_k$ of the $k^{th}$ transition of $h$.

4. The set $S_{Echist}$ can be turned into a complete metric space by endowing it with the following distance $d : S_{Echist} \times S_{Echist} \rightarrow [0, 1]$: for any $h_1, h_2 \in S_{Echist}$:

   $$d(h_1, h_2) = 2^{-\sup\{n : h_1[n] = h_2[n]\}}$$

   with the convention that $2^{-\infty} = 0$.

It will be convenient to be able to compare histories and omit the steps that don’t really change the situation or to extend a history by steps that don’t change the situation. We are thus lead to the following definitions.

**Definition 12**

1. For any history $h$ of $S_{Echist}$, we denote by $\tilde{h}$ the history obtained by replacing in $h$ every maximal sequence $s \overset{\alpha_i}{\rightarrow} s \overset{\alpha_{i+1}}{\rightarrow} s \ldots$ by the only state $s$.

2. Two histories $h_1$ and $h_2$ of $S_{Echist}$, are called stuttering-equivalent if $\tilde{h}_1 = \tilde{h}_2$. In this case, we note $h_1 \simeq h_2$.

3. For any history $h$ of length $m$, we denote by $\hat{h}$ some arbitrary history such that $h \simeq h$ and $\hat{h}[m] = h$.

**Definition 13** Define the operational semantics $O_g : P_{ns}(Sag) \times Seagent \rightarrow P(S_{Echist})$ as follow : for any $\alpha \in P_{ns}(Sag)$ and any $A \in Seagent$,

$$O_g(\alpha)(A) = \{ (\theta_1, \sigma_1) \overset{\alpha_1}{\rightarrow} (\theta_2, \sigma_2) \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} (\theta_n, \sigma_n) : (A | (\theta_1, \sigma_1)) \overset{\alpha_1}{\rightarrow} (A_2 | (\theta_2, \sigma_2)) \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} (A_n | (\theta_n, \sigma_n)) \neq \}

\cup \{(\theta_1, \sigma_1) \overset{\alpha_1}{\rightarrow} (\theta_2, \sigma_2) \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} (\theta_n, \sigma_n) \cdot \cdot \cdot : (A | (\theta_1, \sigma_1)) \overset{\alpha_1}{\rightarrow} (A_2 | (\theta_2, \sigma_2)) \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} (A_n | (\theta_n, \sigma_n)) \overset{\alpha_n}{\rightarrow} \ldots, \alpha_i \subseteq \alpha \text{ or } \alpha_i \subseteq \bar{\alpha} \text{ for } i = 1, \ldots, n \}

Sets of histories that are closed under the equivalence $\simeq$ are defined as properties. An agent is then said to satisfy a property if all the computational histories of its operational semantics are in the set corresponding to the property. More precisely, given a set of agents $A \in P_{ns}(Sag)$, the agent $A$ is said to satisfy property $P$ iff $O_g(\alpha)(A) \subseteq P$ holds. This is subsequently denoted by $A sat_{\alpha} P$

Moreover, it is of usual practice in concurrency theory to define so-called safety and liveness properties. In informal terms, a safety property states that nothing bad can happen whereas a liveness property asserts that something good will happen. Equally, it is often said that safety and liveness properties respectively correspond to invariant and progression properties. This intuitive perception has been shown to correspond to closed and dense subsets in [11].

In topological reasoning, one can prove that any property is the intersection of a safety and a liveness property.
Five properties have been identified in [12] as useful: unless, initially, stable, invariant, and leadsto. The last one is a liveness property; the others are safety properties. We reformulate them to allow for the introduction of agent sets.

**Definition 14** Let $\alpha \in \mathcal{P}(\text{Sag})$ be a set of agents. Assume some theory to write formulae and to determine whether they hold on situations. The properties unless, initially, stable, invariant, and leadsto are defined as follows:

$p$ unless$_\alpha q$

\[
\equiv \{ h : \forall k \leq \text{length}(h) : (h_{k,s} \models p \land \neg q) \\
\land (h_{k,a} \subseteq \alpha) \Rightarrow (h_{k+1,s} \models p \lor q) \}\]

$p$ leadsto $q$

\[
\equiv \{ h : \forall k \leq \text{length}(h) : (h_{k,s} \models p) \Rightarrow \\
(\exists j \in [k, \text{length}(h)] : h_{j,s} \models q) \}\]

Initially $p \equiv \{ h : h_{1,s} \models p \}$

Stable$_\alpha p \equiv p$ unless$_\alpha$ false

Invariant $p$

\[
\equiv \{ h : h_{1,s} \models p, \forall k \leq \text{length}(h) : h_{k,s} \models p \}\]

One could of course try to verify these properties directly on the computational histories. However, it is, in general, simpler and more abstract to use a proof system. The properties just developed being very close to those of Unity, we refer the reader to [12] for such a proof system.

### 3.2 A specification format

Before composing the effect of agents to get that of a set of parallel agents, one has first to specify the behaviors of the former agents in a suitably abstract way. The above properties provide a nice way of grasping the behavior of an agent. However, as already said, this behavior depends, in general, on the behavior of the environment of the agent. We are thus naturally lead to characterize the behavior of an agent by means of a property, guaranteed to hold, provided its environment satisfies another property. Stated in other terms, the behavior of an agent is best specified by a formula of the form $E \Rightarrow G$, where $E$ is the property to be satisfied by the environment of the agent and $G$ is the property then verified by the agent.

### 3.3 A composition principle

The next natural step in the composition methodology consists of characterizing sets of parallel agents by similar specifications, of extending the notion of property satisfaction in a straightforward way to sets of parallel agents, and then of composing the properties.

**Lemma 15** For any $A, B \in \text{Seagent}$ and any disjoint $\alpha, \beta \in \mathcal{P}_{\text{ns}}(\text{Sag})$, one has $h \in \mathcal{O}_g(\alpha \cup \beta)(A \parallel B)$ iff $h \in \mathcal{O}_g(\alpha)(A)$ and $h \in \mathcal{O}_g(\beta)(B)$ hold.

**Lemma 16** For any $A \in \text{Seagent}$, $\alpha \in \mathcal{P}_{\text{ns}}(\text{Sag})$, and properties $E, G$ such that (i) $E$ is a safety property whose unless properties only use $\tilde{\alpha}$ as label, (ii) $G$ is a safety property containing no initialy property and whose unless property only use $\alpha$ as label, (iii) $A \text{ sat}_\alpha (E \Rightarrow G)$. Then for any history $h$ of $\mathcal{O}_g(\alpha)(A)$, if $i \geq 1$ such that $h[i] \not\in G$, then $i > 1$ and $h[i-1] \not\in E$.

**Proposition 17** Let

- $\Pi = \{A_1, \ldots, A_n\}$ be a set of agents that are simultaneously executed;
- $\alpha_1, \ldots, \alpha_n$ be disjoint elements of $\mathcal{P}_{\text{ns}}(\text{Sag})$ and $\alpha$ their union;
- $E, E_1, \ldots, E_n$ be safety properties whose unless properties are labeled by $\tilde{\alpha}, \alpha_1, \ldots, \alpha_n$, respectively;
- $G, G_1, \ldots, G_n$ be (general) properties whose safety parts, $G^*, G_1^*, \ldots, G_n^*$, embody no initially properties and have $\alpha, \alpha_1, \ldots, \alpha_n$ as labels of their unless properties.
Then the following inference rule is valid:

\[ A_i \text{ sat}_{\alpha_i} (\mathcal{E}_i \Rightarrow \mathcal{G}_i) \quad i=1, \ldots, n \]

\[ \mathcal{E}_i, \mathcal{G}_i^0, \ldots, \mathcal{G}_{i-1}^0, \mathcal{G}_{i+1}^0, \ldots, \mathcal{G}_n^0 \models \mathcal{E}_i \]

\[ \mathcal{E}_i, \mathcal{G}_i^0, \ldots, \mathcal{G}_n^0 \models \mathcal{G}^* \]

\[ \mathcal{E}, \mathcal{G}_1, \ldots, \mathcal{G}_n \models \mathcal{G} \]

\[ \Pi \text{ sat}_{\alpha} (\mathcal{E} \Rightarrow \mathcal{G}) \]

4 Conclusion

This paper has sketched a methodological approach for coordinating programs, based on the composition of specifications written in a UNITY-like temporal logic. Lack of space has prevented us from giving all the details. However, the reader interested to know more about our approach is invited to consult the extended version, available as [13].

References


