

Specification and Composition of Software Components: formal methods meet standards

CARLO MONTANGERO AND LAURA SEMINI

Dipartimento di Informatica
Università di Pisa
{monta,semini}@di.unipi.it
www.di.unipi.it/~{monta,semini}

Abstract. We describe how to use a combination of formal methods and standard middleware to approach COP with a coordination based attitude. Separating coordination from functionality, we foster the independent implementation of specific coordination templates on the middleware of interest. We discuss how a specific formal approach can be exploited to derive the interoperability skeleton in CORBA and C++ of the most common interaction template, i.e. client-server.

1 Introduction

Like in many other situations in software design, also in COP it is useful to divide and conquer, and both the component developer and the component integrator can gain from this attitude. The concerns to be separated are functionality and coordination. Indeed, coordination [1, 3] defines the ways components interact to reach the common goals when they are composed in a larger system, and can be largely independent from the specific functionalities of a given application. The separation is useful both at the high (specification) level and at the middle (interoperability) level. At the specification level, where the component integrator operates, separating functionality and coordination helps in understanding how to compose the components at hand. Indeed, there are well established coordination patterns, that can be studied per se, and lessen the needed cognitive load when attacking a new composition. On the other side, i.e. when developing components, coordination defines the structure that the middleware must implement to allow the components to interoperate correctly. Separating coordination from functionality, we foster the independent implementation of specific coordination templates on the middleware of interest. For instance, coordination may lead to the definition of skeletons in some middleware, like CORBA, to be completed by the code implementing the component functionalities. More in general, we can say that coordination defines the implementation of the interoperability level of a framework.

Formal methods can play a major role in COP. Precisely because the actors are programmatically independent, they need to have reliable ways to share precise knowledge of the artifacts they use or produce, independently of the particular technology (programming languages, middleware, ...) they are using. Formal methods offer exactly this kind of independence and precision, since they provide abstract models to share when operating with components. For instance, they provide ways to understand the potentialities and limitations of a coordination pattern, without the need to consider the specific middleware in use. Also, they can provide ways to make precise the specifications of the components and of their contextual dependencies, and to prove in advance global properties of a given composition, i.e. that the composition will meet the specifications it addresses.

An ideal world would see a unique universal middleware for interoperability in COP. In the real world the situation is muddier, but a number of standards (CORBA, COM, DCOM) are emerging, and it is natural to consider them as targets of component development.

This paper describes how we are using a combination of formal methods and standard middleware to approach COP with the coordination based attitude described above. We

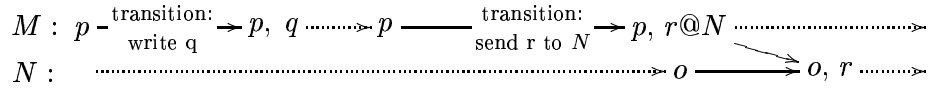
discuss how a specific formal approach, Oikos-*adtl* [4], can be exploited to derive the interoperability skeleton in CORBA and C++ of the most common interaction template, i.e. client-server.

We first briefly present the abstract computational model and the temporal logic system to reason on abstract computations. We then outline how components and coordination template can be specified and verified. Finally, we sketch how the skeleton middleware can be systematically derived and justified from the logical specification.

2 Background: Oikos-*adtl* and composition templates

Oikos-*adtl* is a specification language for distributed systems based on asynchronous communications. 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 *composition templates*. 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 [2] to deal with components and events.

The Computational Model. A system \mathcal{T}_M is characterized by a list of components $\mathcal{M} = M, N, O \dots$, and a set \mathcal{T} of state transitions. A computation is a graph of states like the one in the figure below.



Syntax and Semantics. A specification shapes $\mathcal{M} : \ll \mathcal{F} \gg$, where \mathcal{M} is a set of component names and \mathcal{F} is a set of formulae. Formulae describe computations relating *state formulae*, which express properties of single computation states. Formulae are built using the temporal operators CAUSES_C, CAUSES, NEEDS, BECAUSE_C, BECAUSE, WHEN, and those of Unity. They shape like:

$$M : p \text{ WHEN } c \text{ CAUSES } S : q \quad (1)$$

$$S : q \text{ WHEN } c \text{ BECAUSE } (S : r \wedge M : p) \quad (2)$$

$$M : p @ O \text{ CAUSES_C } O : p \quad (3)$$

With *event* p we mean that property p becomes true. (1) reads: event p in M when c holds causes condition q in S and states that any state of M in which c holds and p becomes true, is eventually followed by a state of S satisfying q , as in the computations to the left, below. Formula (2) states that event q can occur in S when c holds only if previously r and p have hold in S and M , respectively, like in the computation to the right, below.



The state formula $M : p @ O$, means that component M wants to send p to O : (3) is an axiom for Oikos-*adtl*. It says that messages are immediately sent and are guaranteed to arrive. Suffix c stands for *closely*: for instance, in (3), CAUSES_C requires messages to be sent immediately. Operator BECAUSE_C requires that the condition enabling the event happened in the same or in the previous state. NEEDS requires a condition to hold when an event occurs.

Composition Templates. To specify and prove the global coordination properties of a distributed system is often a complex task. A coordination template defines a composition schema for a set of components: the global properties of their parallel composition can be obtained as theorems. Templates shape:

$$\begin{array}{c}
M^1, \dots, M^n : \ll \mathcal{F} \gg \\
\sqsubseteq \frac{\frac{\vdash_{M^1, \dots, M^i} \mathcal{F}' \quad \vdash_{M^{i+1}, \dots, M^n} \mathcal{F}''}{\vdash_{M^1, \dots, M^n} \mathcal{F}}}{M^1, \dots, M^i : \ll \mathcal{F}' \gg \parallel M^{i+1}, \dots, M^n : \ll \mathcal{F}'' \gg}
\end{array}$$

The schema above shows the typical structure of our composition theorems, and reads: the parallel composition of any pair of systems $\mathcal{T}'_{M^1, \dots, M^i}$ and $\mathcal{T}''_{M^{i+1}, \dots, M^n}$ satisfying \mathcal{F}' and \mathcal{F}'' resp., satisfies \mathcal{F} . A system satisfies the set of formulae \mathcal{F} iff all its computations are models of all the formulae in \mathcal{F} .

The proofs are sequences of applications of composition templates, and show a general pattern: first (reading them bottom–up) lift local properties to the global level, and then apply transitivity or other composition rules.

A large set of composition rules are available to reason on Oikos–*adtl* specifications. To give a flavor, we list the most useful ones. (5) is a lifting rule for `BECAUSE_C`: the new component can be an unforeseen cause for A . (4) is a transitivity result for `CAUSES`.

$$\frac{\vdash_{\mathcal{M}} M : A \text{ CAUSES } O : C \quad \vdash_{\mathcal{M}} O : C \text{ CAUSES } N : B \quad \vdash_{\mathcal{M}} O : C \text{ BECAUSE } M : A}{\vdash_{\mathcal{M}} M : A \text{ CAUSES } N : B} \quad (4)$$

$$\frac{\vdash_{\mathcal{M}} M : A \text{ BECAUSE_C } N : B}{\vdash_{O, \mathcal{M}} M : A \text{ BECAUSE_C } (N : B \vee O : A @ M)} \quad (5)$$

3 Using Composition Templates

A simple example of composition template is the client–server template, which is still one of the most common approaches to component composition: we used it to try out our approach. The template shows the local descriptions of client C and server S , and derives the properties of their interaction:

$$C, S : \ll C : req(X) \text{ CAUSES } C : ans(X, V) \quad (6)$$

$$C : ans(X, V) \text{ BECAUSE } S : f(X, V) \quad (7)$$

$$C : ans(X, V) \text{ BECAUSE } C : req(X) \gg \quad (8)$$

\sqsubseteq

$$C : \ll C : \text{INV } server(S), \quad (9)$$

$$C : req(X) \text{ WHEN } server(S) \text{ CAUSES_C } C : send_req(X, C) @ S, \quad (10)$$

$$C : send_req(X, C) @ S \text{ NEEDS } req(X), \quad (11)$$

$$C : ans(X, V) \text{ NEEDS } false, \quad (12)$$

$$C : f(X, V) @ S \text{ NEEDS } false \gg \quad (13)$$

\parallel

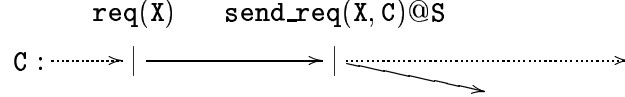
$$S : \ll S : send_req(X, C) \text{ CAUSES_C } S : ans(X, V) @ C, \quad (14)$$

$$S : ans(X, V) @ C \text{ BECAUSE } S : f(X, V), \quad (15)$$

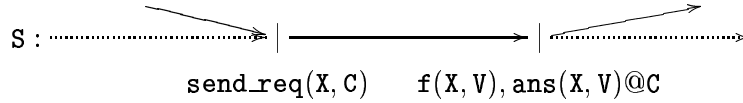
$$S : f(X, V) \text{ BECAUSE } S : send_req(X, C), \quad (16)$$

$$S : send_req(X, C) \text{ NEEDS } false \gg \quad (17)$$

Following [5], a server component is characterized by the fact that a request to the server carries explicitly the name of the client, in order to deliver correctly the response. The client knows the server (9), sends it its requests (10, 11), and does not produce (12) or compute (13) on its own answers to the requests. Computations of the client look like:

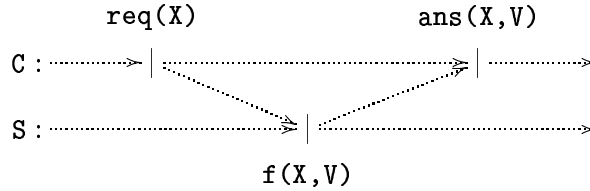


The server answers the requests of the client (14). Answers are produced after computing some value V (15), and this is done only upon a request (16). Finally, servers do not produce requests (17). Computations of the server look like:



The local specifications fix the interface of the server, and the most abstract functional constraint, i.e. that a value has to be computed by the server before answering.

The components can be developed independently, since the pattern, having fixed the interactions, projects on each component its responsibility. Besides, the pattern proves in advance that the interaction between client and server satisfies some global properties: requests will receive an answer (6); answers respect the values computed by the server (7) and are received only upon request (8). The following picture shows a computation satisfying these properties.



Our case study is taken from the CORBA documentation, and considers a simple Stock Exchange. The system is composed of a *Quoter* and a set of clients C_i . Clients interact with the Quoter to be informed on the values of some stocks at the Stock Exchange. In particular, the generic client C_i interacts with the Quoter if interested in the value of stock X ($stock_value(X)$). The Quoter computes the current value V for X ($current(X, V)$), and sends the data to the client ($quote(X, V)$).

$$\begin{aligned}
 Quoter, C_i : \ll C_i : stock_value(X) \text{ CAUSES } C_i : quote(X, V), \\
 C_i : quote(X, V) \text{ BECAUSE } Quoter : current(X, V), \\
 C_i : quote(X, V) \text{ BECAUSE } C_i : stock_value(X) \gg
 \end{aligned}$$

The example shows a client–server interaction among the clients and the Quoter. We thus instantiate the client–server template and obtain the following local specifications, where $get_quote(X, C_i)$ encodes the request of C_i :

$$\begin{aligned}
 Quoter : \ll (\text{for all } i) \\
 Quoter : get_quote(X, C_i) \text{ CAUSES_C } Quoter : quote(X, V)@C_i, & (18) \\
 Quoter : quote(X, V)@C_i \text{ NEEDS } current(X, V), & (19) \\
 Quoter : current(X, V) \text{ NEEDS } get_quote(X, C_i), & (20) \\
 Quoter : get_quote(X, C_i) \text{ NEEDS } false \gg & (21)
 \end{aligned}$$

$$Ci : \ll Ci : \text{INV } server(Quoter), \quad (22)$$

$$Ci : \text{Stock_value}(X) \text{ WHEN } server(Quoter) \text{ CAUSES_C}$$

$$Ci : \text{get_quote}(X, Ci)@Quoter, \quad (23)$$

$$Ci : \text{get_quote}(X, Ci)@Quoter \text{ NEEDS } \text{Stock_value}(X), \quad (24)$$

$$Ci : \text{quote}(X, V) \text{ NEEDS } \text{false}, \quad (25)$$

$$Ci : \text{current}(X, V)@Q \text{ NEEDS } \text{false} \gg \quad (26)$$

4 Implementing a coordination template

The abstract coordination template described above can be transformed once for all in a concrete interaction skeleton in a standard middleware. We exemplify this with CORBA and C++. Given that our logic setting is asynchronous, we use the recent specification of CORBA asynchronous method invocation (AMI, [6]).

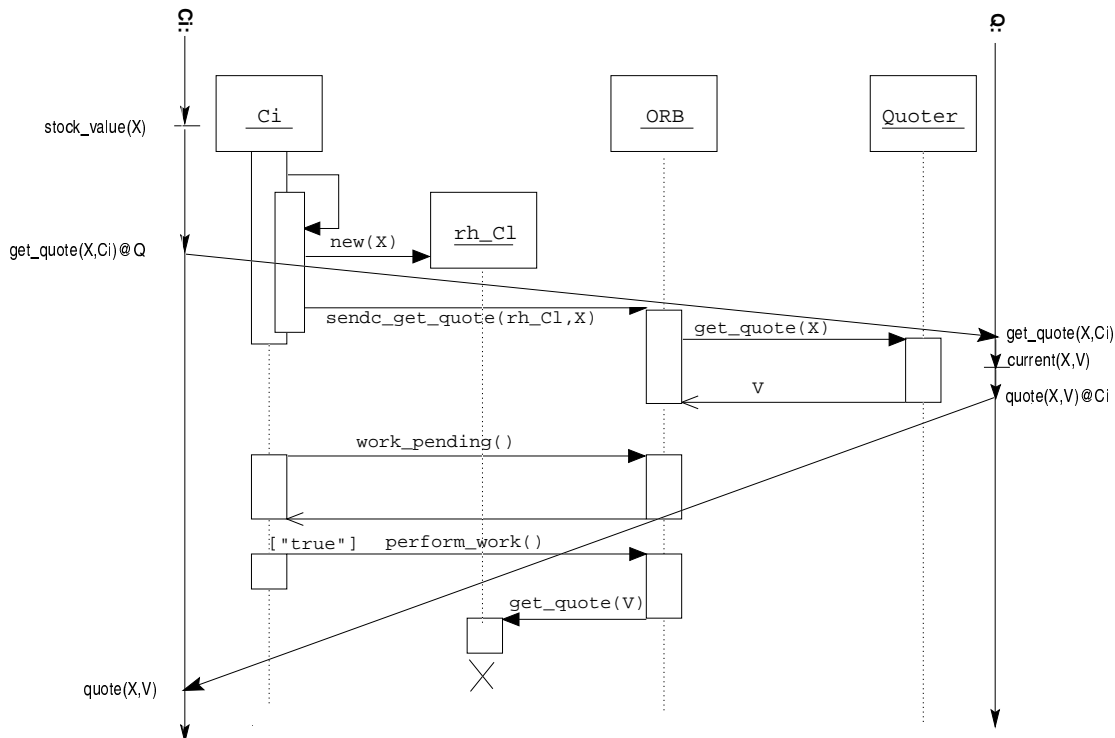


Fig. 1. Correspondence template-CORBA

AMI has been conceived so that no changes are needed, with respect to the synchronous case, on the side of the callee (the server side). It is up to the caller (the client) and to the ORB to cope with the differences with synchronous invocations. This is apparent in the UML sequence diagram in Fig. 1: the interactions on the left side (client-ORB) are more complex than those on the right side (ORB-server). The figure uses the Stock Exchange example to illustrate the correspondence between the computation template and CORBA. The server interface

```
interface Quoter {
    double get_quote(in string X);
}
```

can be derived systematically from the logical specification. The CORBA compiler generates also a skeleton class implementing the server:

```
class Quoter_i : public virtual POA::Quoter {
    ...
public:
    ...
    CORBA::Double get_quote(const char* X) {
        CORBA::Double V;
        \\ insert here the code to compute V,
        \\ such that current(X, V) holds
        return V;
    };
};
```

As the superposition of the client–server template on the sequence diagram in Fig. 1 should make clear, what is left on the server side is the implementation of the server functionality, as specified by predicate *current*.

From interface `Quoter`, the CORBA compiler generates also a stub C++ class, which exports a method `void sendc_get_quote(AMI_Quoter_Handler, char*)` that the client uses to invoke the `Quoter` method `get_quote(char*)` remotely via the ORB. The second argument of the asynchronous call is the C++ transposition of the original argument to `get_quote`, and is delivered to the server by the ORB, via the server method. The first argument is a callback handler created by the client. This object exports a method `get_quote(double)` that the ORB exploits to return the answer asynchronously.

The skeleton of the callback handler can also be generated from the specification:

```
class Handler : public POA::AMI_Quoter_Handler {
private:
    CORBA::X_type X ;
    // initialized to the original argument to get_quote
..
public:
    void req(CORBA::V_type V) {
        // insert here the continuation code of the client:
        // quote(X,V) has been established
    };
    ...
};
```

The client is not blocked, and uses the ORB methods `work_pending()` and `perform_work()` to control when to receive the answer via the handler.

The destructor of the client object can be defined to implement the busy–waiting cycle that is necessary, according to the AMI specification, to force the ORB to deliver the answer from the server, when available. In the simplest cases, this allows the client code to terminate immediately after the asynchronous call, and let the handler complete the computation, once the answer is available. This is suggested by the comment in the code above. In other cases, e.g. when the client has to exploit several answers, it may be necessary to split the code between the client and the handler: this may require some refinement steps, to decide how to proceed. How much of these refinements can be standardized is still a matter of investigation.

5 Conclusions

We think that our approach shows that formal methods can play an essential role in characterizing component coordination at the abstract level, identifying the interactions between components and their context, to a point where standard skeletal implementations can be rigorously derived, for a large set of standard middleware. This can liberate component oriented programming from the burden of repetitive tasks, leaving space to more ingenious activities, related to the specifics of the problem at hand.

Acknowledgements

This work was partly supported by the ESPRIT W.G. 24512 COORDINA and the Italian MURST project SALADIN. R. Dolfi experimented with CORBA. D.C. Schmidt and T. Flagella offered helpful support (remotely and locally, respectively).

References

1. N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 5(2):97–107, 1989.
2. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading Mass., 1988.
3. P. Ciancarini and A. Wolf, editors. *Proc. 3rd Int. Conf. on Coordination Models and Languages, COORDINATION 99*, volume 1594 of *Lecture Notes in Computer Science*, Amsterdam, April 1999. Springer-Verlag.
4. C. Montangero and L. Semini. Composing Specifications for Coordination. In [3], pages 118–133.
5. 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.
6. OMG. CORBA Messaging Joint Revised Submission. Technical Report orbos/98-05-05, Object Management Group, Framingham, MA, 1998.