Composita: Bringing Order to Agent Communications

Extended Abstract

Luc Bläser

University of Applied Sciences Rapperswil
Institute for Software
lblaeser@hsr.ch

Abstract

Keeping order in message communication is essential for building complex actor- or agent-oriented programs. Unfortunately, the standard models do not offer much support in this regard: Communication only remains an implicit effect of actor/agent implementations. To improve this structural deficiency, we advocate EBNF-style grammar protocols for defining bilateral communications across agents or actors.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages

Keywords agents; actors; message communication; grammar-based protocols

1. Introduction

With the Actor model [1, 11], message exchange can soon become highly complex because programs lack an explicit specification of *what* messages may be transmitted between *which* actors in *which* order. Naturally, these questions are decisive for a correct combined behavior, but unfortunately, an intended protocol is only implicitly given by the implementation of the involved actors. Without particular care, the complexity grows quickly and finally becomes unmanageable.

This structural deficiency can only be overcome by fitting up agents/actors with a proper interface that provides an abstract description of their potential external interactions. We have taken such an approach in our Composita language [3, 5]. For this purpose, we deviate from the tradi-tional actor notion and elevate the model to more component-oriented agents, i.e. active objects with explicit interfaces. The interfaces serve to define bidirectional communications in a formal notation, namely EBNF-based protocols. We present the key concepts and advantages of this model.

2. Composita Model

The Composita language [4] establishes an agent-based model with self-active objects, called *components*, and protocol-structured communications. On the one hand, it resembles the Actor model insofar that the objects only interact via asynchronous message sending and reception. On the other hand, it distinguishes itself from the Actor model in terms that message transmissions are only allowed between predefined agent-to-agent connections and messages within the communications have to adhere an explicit protocol. The distinction deliberately serves for the sake of structural stringency of message communication. The subsequent explanation focuses on the concepts in more detail.

```
COMPONENT Restaurant
  OFFERS RestaurantService;
(* … *)
END Restaurant;

COMPONENT Customer
  REQUIRES RestaurantService;
(* … *)
END Customer;
```

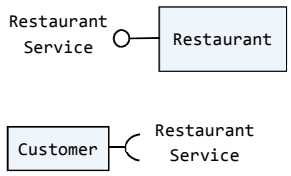

Figure 1. Component templates (left) with instances in diagram notation (right).

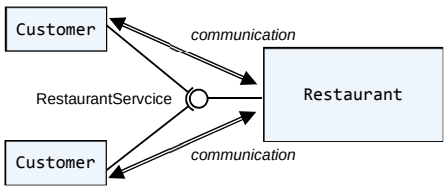

Figure 2. Separate communication per pair of connected instances.

2.1 Agent-Based Components

In Composita, a *component* represents a unit of active stateful behavior that interacts with the outside via interfaces. A component has *offered* and *required* interfaces, see Figure 1. An offered interface represents a communication endpoint of the current component that can be used by external instances. Conversely, a required interface denotes a foreign communication endpoint that can be used by the current component. Components can be wired by connecting required interfaces to offered interfaces, supporting many-to-many connections. The interface enables an independent communication between each pair of connected components, see Figure 2.

2.2 Communication Protocols

All feasible message transmissions of a communication are thereby defined by a protocol following the EBNF-syntax [13], see Figure

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
INTERFACE RestaurantService;
  IN RequestTable(nofSeats: INTEGER)
  (
    OUT Unavailable
  |
    [ OUT MomentPlease ]
    OUT ComeIn
    IN Enter
    { IN Order(description: TEXT) }
    { OUT ServeMeals }
    IN PayAndLeave
  )
END RestaurantService;
```

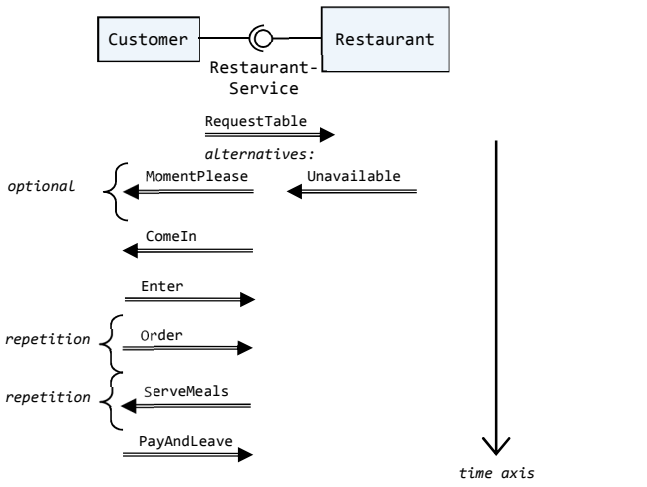

Figure 3. Interface defining the communication protocol.

3. Messages are to be declared with an identifier and a list of parameters that represents values to be carried as message content. Messages with the declared `IN` direction are sent from an external instance to the component offering the interface, vice versa for `OUT` messages. In EBNF, a concatenation of expressions denotes a sequence, curly braces represent arbitrary repetitions (including zero times), vertical bars mean alternatives and square brackets indicate options. To obtain full context-sensitive grammars, the grammar could be organized in productions mapping non-terminal symbols to grammar expressions.

For general implementations, protocols need to be moni-tored at runtime, as in our system [3, 4]. Communication protocols establish contracts between the involved component/agents, making protocol violations apparent, and allowing each side to be changed or replaced under clear conditions, namely as long as they adhere to the protocol.

3. Related Work

Occam [12] already introduced a protocol specification of alternatives and sequences, however limited to unidirectional channels. The concept of synchronizers [6] employs transition rules with pattern matching that is general but probably less evident: the possible flow of messages as part of logical communication needs to be derived from applying the transitions, while a structured protocol grammar denotes all valid message sequences quite directly and simply. For example, saying that message `a` causes `b` or `c`, and that message `b` triggers `a`, is less obvious than our EBNF notation of `{ a b } c`. Sing# [7] is similar, expressing protocols as transitions in a logical state machine. The comparison of a state machine versus EBNF notation has structurally similarities to comparing *goto*-based logic with structured programming. The work of Astley et al. [2] introduces protocols with an arbitrary amount of roles, event and message operations, as well as an informal description of effects on events/operation. Still, it only describes effects per message/event separately, not as structured as a grammar. Florijin's grammar-based protocols [8] captures context-sensitive properties and parallelism across a set of agents. We deliberately only focus on bilateral communication, as a tradeoff between expressiveness and simplicity. Our protocols are also written in a proper syntax, not only as functional parser rules.

The presented EBNF-protocols are inspired by the dialog concept of Zonnon [9] and Active C# [10]. We add some refinements, such as the high-level messages with arbitrary content instead of directly exchanging tokens and single data values. The Composita programming language and system are described in previous publications [3–5].

4. Conclusions

Clearly structured communication is an essential prerequisite for designing more complex actor- or agent-oriented programs. By the example of our Composita language, we show how this may be achieved: Agents can be upgraded to components with explicit interfaces, specifying their interactions with the outside at an abstract level. EBNF-style protocols therein establish an expressive but simple contract for the bidirectional communication between two instances.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] M. Astley, D. C. Sturman, and G. Agha. Customizable Middleware for Modular Distributed Software, Communications of the ACM, 44(5):99-107, 2001.

[3] L. Bläser. *A Component Language for Pointer-Free Concurrent Programming and its Application to Simulation*. ETH Diss. 17480, ETH Zürich, Nov. 2007.

[4] L. Bläser. *A High-Performance Operating System for Structured Concurrent Programs*. Workshop on Programming Languages and Operating Systems (PLOS) 2007, Stevenson WA, USA, In ACM Digital Library, Oct. 2007.

[5] L. Bäser. *A Component Language for Structured Parallel Programming*. Joint Modular Languages Conference (JMLC) 2006, Oxford, UK, In Lecture Notes in Computer Science 4228, Springer, Sep. 2006.

[6] S. Frølund and G. Agha. *A Language Framework for Multi-Object Coordination*. In Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Jul. 1993.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, et al. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. In Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06), Apr. 2006.

[8] G. Florijin. *Object Protocols as Functional Parsers*. In Proc. of the 9th European Conference on Object-Oriented Programming, ECOOP095, Aug. 1995.

[9] J. Gutknecht. *Zonnon Language Report*. ETH Zurich, http://www.zonnon.ethz.ch, 2009.

[10] R. Güntensperger and J. Gutknecht. *Active C#*. In Proc. of the Intl. Workshop on .NET Technologies, May 2004.

[11] C. Hewitt, P. Bishop, and R. Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence*. Proceedings of the 3rd International Joint Conference on Artificial intelligence IJCAI'73. 1973.

[12] Imnos Ltd. *Occam 2 Reference Manual*. Prentice-Hall, 1988.

[13] N. Wirth. *What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?* Communications of the ACM, 20(11):822-823, November 1977.