Communication-based Development of Systems Using Standard Programming Languages (Extended Abstract)

Annette Stümpel

Institute of Software Technology and Programming Languages
University of Lübeck, Lübeck, Germany
stuempel@isp.uni-luebeck.de
http://www.isp.uni-luebeck.de

1 Proposal

Implementations often do not show much resemblance to the initial specifications describing the communication between the system components. In order to reduce this discrepancy and to strengthen the impact of communication-based specifications, we promote a communication-oriented style of programming, which is possible in some standard programming languages.

2 Motivation

Modern software systems are usually structured into communicating subsystems and components, see Fig. 1. A user of the system can often observe only the communication between the components and the environment. The internal design, such as the state and the algorithms inside the components, remains hidden.

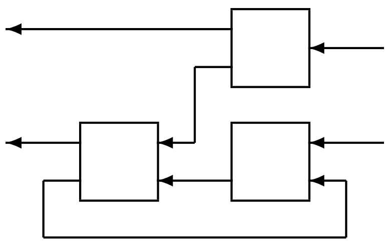

Fig. 1. System composed of communicating components

A specification of a system first describes the black box view, which concentrates on the observable behaviour. The communication of the system components with each other and with their environment characterizes this external view. Modelling whole conversations is also essential for the specification of (web) services ([1, 4]).

For the communication-based specification software developers employ graphical models such as message sequence charts [6] or UML sequence diagrams [7] or even less formal notations. Stream processing [2, 9] constitutes a formal model for communication-based specifications. Stream functions describe the input / output behaviour of components and can be composed to larger systems.

Although the communication-based view is the essential part of the specification which the programmer as well as the user understand, the implementation often does not reflect that specification. There is a gap between the communication-based specification and the implementation because the implementation is often not constructed from the communication-based specification.

With some programming languages, even some standard ones, it is possible to mould the communication-based specification into a specification which clearly reflects the underlying communication-based specification. We show how to achieve this with a lazy functional language, Haskell, and an object oriented language with threads, Java.

The communication-oriented implementations can be obtained from communication-based specifications in the setting of stream processing. In an initial step, these input / output specifications are enriched with states by a design decision and formal transformations [3]. The resulting state transition machine with input and output establishes the relationship between the previous input history of a component and its current state. Depending on the current state and the current input message the state transition machine records the outputs caused by the input and the successor state.

3 Haskell

The functional language Haskell supports communication-based programming with its lazy lists.

For a straightforward implementation of a network of components in Haskell, each component is implemented by a function operating on lists. We introduce a name for each communication channel.

We obtain the composite network as mutually recursive equations: For each component an equation determines the tuple of output streams referred to by their names resulting from applying the component's function to the tuple of input streams also referred to by their names.

Lazy lists are well suited for modelling the operational progress of the system with lists representing the contents of the communication channels. A lazy list is a list whose elements are already evaluated in an initial part. At a certain point the remaining list is unknown. It is even unknown whether there are further elements in the list at all. This concept corresponds exactly to a stream, which records the sequence of messages transmitted on a channel until a certain point of time: the initial part has already been observed but it is not clear whether further messages will appear, let alone their content.

For example, the empty stream $\langle\rangle$ corresponds to the empty lazy list $\perp$. The stream $\langle 2, 6, 3, 1\rangle$ corresponds to the lazy list `2 : 6 : 3 : 1 :` $\perp$. Note that there is no direct syntax for lazy lists in the programming language Haskell.

The laziness of the stream lists has to be taken carefully into account for the implementation of the components by functions over lists. Simple transformations on specifications may not transfer from the stream notation to the Haskell notation. Haskell functions which refer to larger parts of the input stream may block the whole network. Haskell functions which refer only to the first element of their input streams do not block the network. A state transition table of a state transition machine validates this shape of specification.

If the components' communication-based specifications are already provided in the form of a state transition machine, the network can straightforwardly be implemented in Haskell using lazy lists.

4 Java

Java provides threads for communication-based programming. A straightforward approach implements the components and the channels as concurrent threads. Each channel thread records the messages sent via the channel and not yet read by the recipient in a FiFo-queue, and each component thread continually reads messages from its connected input channels and writes messages to its connected output channels.

The STREAMS tool developed at the University of Lübeck [5] supports the simulation of systems in Java using this approach. The STREAMS simulator is a graphical tool which visualizes the stepwise execution of systems. The tool provides an editor for building systems from the implemented components and connecting output ports of components with input ports with the same type. Furthermore, the tool provides a simulation mode in which the user can manually select the component for the next execution step or the user can select between various concurrent simulation modes in which the components can execute several steps concurrently.

The tool provides implementations for channels and superclasses for the components. The state transition machines can be coded in a quite straightforward way. Each component must contain a method implementing the single transition steps. This method may use the following important methods for accessing its input and output channels:

- checking whether an input channel is empty
- inspecting the first element of an input channel
- removing the first element of an input channel
- writing an element to an output channel

Each component is equipped with a standard graphical representation. Enhanced graphical representations can be obtained by overwriting some simple methods.

In this way we get implementations in Java which clearly reflect the communication-based specifications. In large parts these implementations can even be generated automatically from state transition tables of state transition machines [8].

5 Conclusion

Communication-based specifications play an essential rôle in the development of systems built from communicating components. Such systems can only operate successfully if each component always provides exactly the desired service. Programs which are systematically constructed from these specifications are supposed to be developed more efficiently and to deserve more confidence in their correctness than programs which do not reflect the specified communication.

Besides languages which are designed for the communication-based implementation, such as Erlang, there are also some standard programming languages which can be used for a communication-oriented programming style. Haskell with its lazy lists and Java with its threads are good examples.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer, 2004.
2. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Monographs in Computer Science. Springer, 2001.
3. W. Dosch and A. Stümpel. Deriving state–based implementations of interactive components with history abstractions. In I. Virbitskaite and A. Voronkov, editors, *Perspectives of Systems Informatics (PSI 2006)*, number 4378 in Lecture Notes in Computer Science, pages 180–194. Springer, 2007.
4. W. Dosch and A. Stümpel. Implementing services by partial state machines. In M. Nielson, A. Kučera, P. Miltersen, C. Palamidessi, P. Tůma, and F. Valencia, editors, *Theory and Practice of Computer Science (SOFSEM 2009)*, number 5404 in Lecture Notes in Computer Science, pages 241–254. Springer, 2009.
5. B. Hoffmeister. Entwicklung eines Simulators zur Darstellung der Kommunikation in verteilten Systemen. Semesterarbeit am Institut für Softwaretechnik und Programmiersprachen der Medizinischen Universität zu Lübeck, 2001.
6. International Telecommunication Union (ITU-T). *Z.120: Message Sequence Chart (MSC)*, 2004.
7. Object Management Group (OMG). *Unified Modeling Language: Superstructure*, 2.2 edition, 2009.
8. A. Stackebrandt. Component-based development of ePayment systems. Diplomarbeit, Universität zu Lübeck, 2008.
9. A. Stümpel. *Stream Based Design of Distributed Systems through Refinement.* Logos Verlag Berlin, 2003.