Towards Verification of Systems of Asynchronous Concurrent Processes

Marek Rychlý *
rychly@fit.vutbr.cz

Abstract: Distributed information systems as applications of networked information technology systems create needs for reliable architectures with emphasis on formal specification, verification and validation. In fact, software confederations and global software systems connect many concurrent processes into asynchronous network systems, e.g. via service-oriented architecture or multiple agent architecture. A reusable object-oriented software framework for systems of asynchronous concurrent processes based on the modified asynchronous network model provides a framework for implementation of asynchronous distributed systems and their translation into the process algebra. This paper is about a formal specification and verification of systems implemented using the framework by means of the calculus of mobile processes (π-calculus).

Keywords: Asynchronous Network Model, System of Asynchronous Concurrent Processes, π-Calculus, Formal Specification, Verification, Validation

1 Introduction

Distributed systems can be found in many present information technology systems. Extensive information systems create needs for distributed architectures consisting of many nearly independent and spread subsystems. A definition of networked information technology (NIT) systems [1] as information technology systems, which are responsible for processing data into information over geographically dispersed regions, sets new challenges to formal specification and verification. Indeed, NIT system, which is a distributed system of concurrent processes can be found in every physical, natural or social process where subsystems possess independence, autonomy, and their own sense of timing. In practice, this type of system architecture leads to service oriented architectures or multi-agents systems, but there is missing an abstract simple object-oriented model, which is independent of concrete implementation of infrastructure parts.

In this paper, there is shortly described an object-oriented reusable framework for systems of asynchronous concurrent processes based on a modified asynchronous network model. The goal of the framework is to provide an abstract object-oriented model to separate parts of a distributed system by means of interfaces, and to reduce time and effort involved in implementation. After a brief description of the modified asynchronous network model and the mentioned framework, this paper will describe a formal specification of framework-based systems using π-calculus and way to their verification.

* Department of Information Systems, Faculty of Information Technology, Brno University of Technology, Bozetechova 2, Brno 612 66, Czech Republic

2 Modified Asynchronous Network Model

An asynchronous network model (ANM) is a formal model of asynchronous communicating concurrent processes [2]. ANM consists of a directed graph $G=(V,E)$, where V is set of nodes and E is set of edges. Each node $v_i \in V$ is associated with a process P_i and has in-neighbors and out-neighbors. Each directed edge $e_j \in E$ of the graph is associated with a communication channel, which connects its neighbor nodes in a given direction. Both processes and channels can be modelled as an arbitrary I/O automaton (a labelled transition system model with output, internal and always enabled input actions and a fair execution, see [2]), with operations:

- $send(m)_{i,j}$ to send message m from process i to process j (an out-neighbor in the graph G),
- $receive(m)_{j,i}$ to receive message m from process i (an in-neighbor) by process j.

There are many kinds of channels (channels with failures, reliable reordering channels, etc.) that support usability of ANM for asynchronous systems of real word. An extension [3] of the *universal reliable FIFO channel* [2] supplemented with some multicast features will be used in this paper. In fact, we split a channel into two components, named "port" and "link", where the link is a network buffer able to receive a message from many processes (senders) and deliver it to at most one of many listening processes (receivers). The port is an interface between a process and link, implementing operations *send* and *receive* and being able to deliver sent messages to only one link, but accept a message from many links while receiving (see Figure 1).

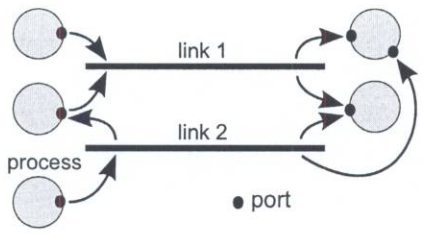

Figure 1. An example of the modified asynchronous network model

The translation of this modified ANM (with process, port and link) into original one (with process and channel, as in [2]) is possible, because the process connected with neighbours using classic channels can emulate a link. Therefore the extension is a conservative extension, which is adapted to better support of reusability (a three-layer framework introduced in the next chapter).

3 Framework for Systems of Asynchronous Concurrent Processes

This part of the paper will shortly introduce an object-oriented reusable framework for systems of asynchronous concurrent processes based on the modified ANM [3]. The goal of the framework is to provide an abstract object-oriented model to separate parts of a distributed system by means of interfaces, to reduce time and effort involved in implementation, and allow describing formal specification of such system and verify it.

The modified ANM splits a system into three parts (processes, ports and links), where each part has its own role in the process of communication. The framework can use these parts in the vertical view as nearly independent layers (see Figure 2):

1. *Process Layer (Application Layer)* — The top layer of the architecture, which is aware of needs of the application. Objects in this layer represent processes. There are *atomic processes* and *composite processes*.

2. *Port Layer (Presentation Layer)* — The middle layer, which provides an interface between the process layer and the link layer. Operations *send* and *receive* are implemented at this layer and are used by processes from process layer. We suppose an asynchronous send and a synchronous receive.
3. *Link Layer (Transport Layer)* — The bottom layer is responsible for connecting communicating processes. It implements universal reliable FIFO channel, or in general some of other channel type. This layer provides an abstraction of an underlying network, realized for example using an inter-process communication or a service-oriented architecture.

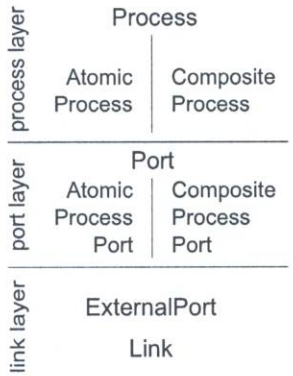

Figure 2. Vertical and horizontal view of framework architecture

In the horizontal view, *the process layer* consists of two entities: `AtomicProcess` and `CompositeProcess`. The first one represents indivisible functional entity at the level of an abstraction, which is used (such as services, library functions, calls of objects' methods, etc.). A composite process, as the second entity of the process layer, represents a subsystem of communicating processes designed using this framework with the interface for input messages. Subprocesses contained in the composite process can be both types, atomic and composite.

Similarly to the previous layer, *the port layer* contains `AtomicProcessPort`, which makes an interface of atomic process (implements send and receive operations), and `CompositeProcessPort`, which is used to send messages from processes outside towards processes inside a composite process (the opposite direction is realized directly). In fact, a composite process port should be the only connection (a proxy) for sending messages to a port of some inner process.

The link layer is presented to higher layers as a black-box accessible through interface `Link`, which connects process ports using interface `ExternalPort` and design pattern "observer". For our purpose, the implementation of this layer in the framework is not important. Suffice it to know that the link layer acts as a (buffered) reliable link, which transports a message from each of processes connected to the link as senders to one of receiving processes (the choice of a group of processes listening on such link is in an unspecified order).

The Figure 3 shows a schematic view of class hierarchy and relations in the framework for systems of asynchronous concurrent processes. Implementation details of the framework are out of scope of this paper, but a full description of the framework can be found in [3].

4 Semantics of Systems of Asynchronous Concurrent Processes

The framework for systems of asynchronous concurrent processes described in this paper uses a modified asynchronous network model to catch communication structure and hierarchy of

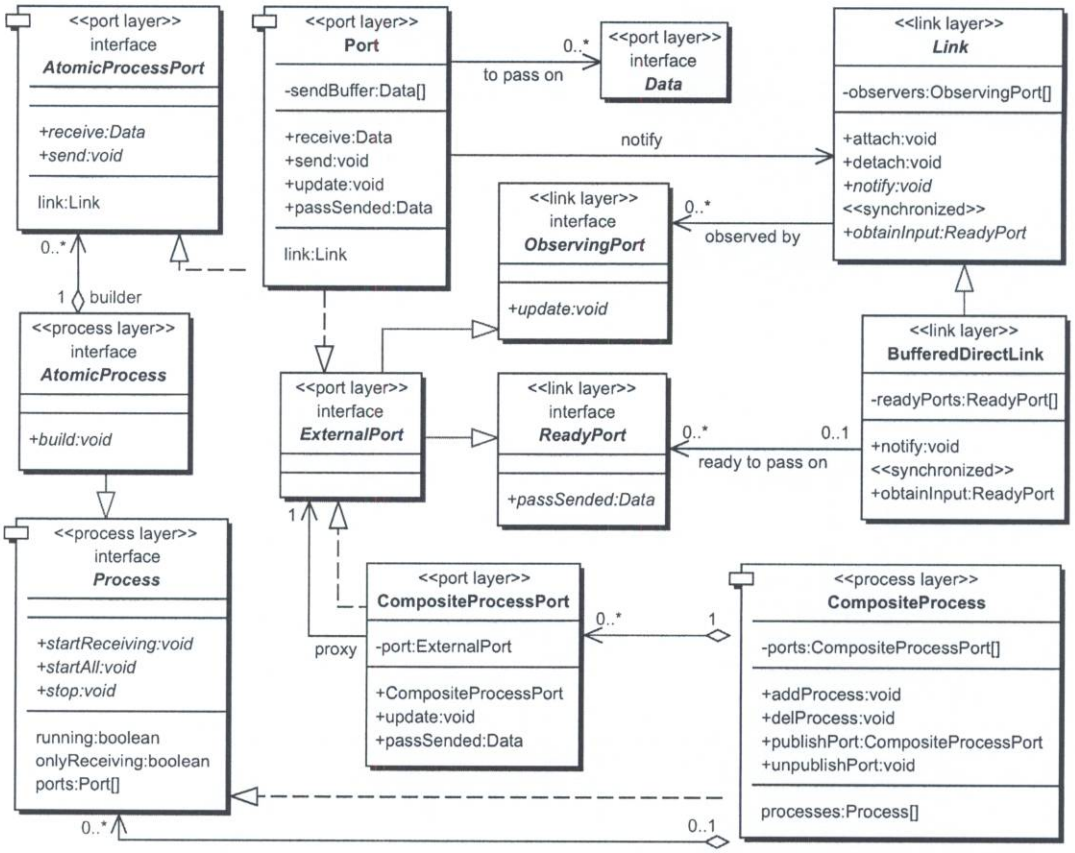

Figure 3. A schematic view of class hierarchy and relations in the framework (from [3])

processes. The semantics of a system implemented using the framework can be formally described in a process algebra (more specifically in the π-calculus), as it will be shown in the next paragraphs.

There exist many successful models formulated using process algebra, but one of the most spread algebraic frameworks is *Calculus of Communicating Systems* (CCS). The *π-calculus* (known as "calculus of mobile processes", see [4]) is an extension of CCS, which allows modelling of systems with dynamic communication structures (i.e. mobile processes). The π-calculus uses only two concepts:

- *a process* (agent) — an active communicating entity in the system, atomic or expressed in π-calculus (denoted by uppercase letters in expressions),
- *a name* (port) — anything else, e.g. channel, variable, data, or also process in high level view (denoted by lowercase letters in expressions).

A process is formally defined in π-calculus using induction. At first, the process `0` is a π-calculus process (null process). If processes P and Q are π-calculus processes, following expressions are also π-calculus processes with given syntax and semantics (the operational semantics of the π-calculus is described and explained in [4]):

1. $\overline{x}\,y\,.P$ sends name y via port x and continues as process P,
2. $x(y).P$ receives name y via port x and continues as process P,
3. $\tau.P$ does an internal (silent) action and continues as process P,

4. $(x)P$ creates new name x in a context of process P and continues as process P,
5. $[x = y]P$ proceeds as P if names x and y are identical, else behaves like a null process,
6. $P \mid Q$ proceeds as parallel composition of processes P and Q,
7. $P + Q$ proceeds as either process P or process Q (a non-deterministic choice),
8. $A(y_1, \ldots, y_n)$ behaves as process with substitution $P\{y_1/x_1, \ldots, y_n/x_n\}$ where (the parametric process) agent A is defined as process P and names $x_1, \ldots, x_n$ occur free in process P.

The interaction between two processes is done through ports, which are identical in their context, according to the previous semantics. For example, the system, which is defined as process $\overline{x}\,y\,.P \mid x(z).Q$ (a parallel composition of the process, which sends name y via port x and continues as process P, and the process, which receives name z via port x and continues as process Q), can perform a communication step (via port x from an inside view and as an internal action τ from an outside view of the system). After this communication, the system is defined as process $P \mid Q\{y/z\}$ (all free occurrence of z in Q are replaced by y).

In this paper, the π-calculus will be used for description of an asynchronous network model and its implementation in the framework for systems of asynchronous concurrent processes. The main reason for using the π-calculus is its ability to catch dynamic communication structure of systems implemented using the framework.

Now, suppose we have a system implemented using the framework. At first, for each *atomic process* (instance of `AtomicProcess`) there is given semantics in π-calculus. This is necessary because behaviour of a process is hidden at the level of abstraction, which is used in the framework. However, the framework specifies an interface of each process using the port and so the designer of a system in the framework is forced to use this communication interface and to design process interface compatible with specification in π-calculus.

A *port p* (instance of `AtomicProcessPort`) of a process in the framework is expressed as two channels in π-calculus: p_{in} for receiving and p_{out} for sending. A *link* (instance of `Link`) in the framework, which transmits messages from ports represented in π-calculus as $q1_{out}, \ldots, qm_{out}$ to ports represented as $p1_{in}, \ldots, pn_{in}$, can be expressed as a process in π-calculus (the repeated non-deterministic choice of two opposite channels and a communication between them):

$$\text{link}(p1_{in}, \ldots, pn_{in}, q1_{out}, \ldots, qm_{out}) = \sum_{i=1}^{n} \sum_{j=1}^{m} qj_{out}(x).\overline{pi_{in}}\,x\,.\text{link}(p1_{in}, \ldots, pn_{in}, q1_{out}, \ldots, qm_{out})$$

It is necessary to remark, that if one port in the framework is expressed as one channel in π-calculus, a message would be transferred incorrectly from one link to another one without the action of a process, which owns the port. For that reason, a port cannot be expressed as one channel in π-calculus. Also the definition of a link cannot be created without a non-deterministic choice, because otherwise a message would be sent to the incorrect channel of an inactive process, while another process is ready to receive it.

The last two entities of the framework are the composite process (instance of `CompositeProcess`) and the port of a composite process (instance of `CompositeProcessPort`), which acts as a proxy transferring messages from processes outside towards processes inside composite process. The purpose of *composite process* is to hide processes inside it from outside views. In the π-calculus, the composite process can be defined as a parallel composition of its internal processes, where *ports of a composite process* act as its public names — the composite process is a parametric process with the ports of a composite process as its parameters, according to the last part of the definition of π-calculus process from the beginning of this chapter.

4.1 Semantics of Sample System

Suppose we have a simple *system with client-server architecture* implemented using the framework, which contains one atomic process (representing a client) and one composite process (representing a server). In this system, operations send and receive (listening) are implement using ports as process interfaces. The system works as follows (see Figure 4):

1. the server listens for requests on link `l`,
2. the client sends a request (with a pointer to private link `k`, where it listens) into link `l`,
3. when the server receives the request, it creates another concurrent process `P` to handle the request,
4. then process `P` creates link `j`, where it listens, and sends pointer to this link into link `k`,
5. at the end of the scenario process `P` and the client process communicate through link `k` (from process `P` towards the client process) and link `j` (form the client process towards process `P`).

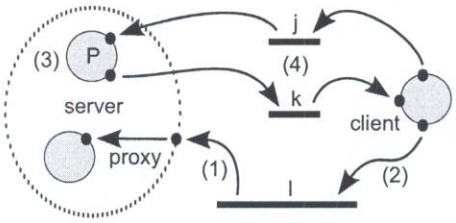

Figure 4. The client-server scenario in modified asynchronous network model[1]

The client-server system can be described in π-calculus as a parallel composition of processes $Client(l_{out})$, $Server(l_{in})$ and $link(l_{in}, l_{out})$ as follows (where the link is defined, as it was shown before, and $Client_{connected}$ and $Server_{connected}$ are some processes in π-calculus, which communicate together using two channels):

$$\text{Client}(l_{out}) = (k_{in}).\overline{l_{out}}\, k_{in}\, .k_{in}(j_{out}).\text{Client}_{\text{connected}}(k_{in}, j_{out})$$
$$\text{Server}(l_{in}) = \text{Server}(l_{in}) \mid l_{in}(k_{out}).\text{P}(k_{out})$$
$$\text{P}(k_{out}) = (j_{in}).\overline{k_{out}}\, j_{in}\, .\text{Server}_{\text{connected}}(j_{in}, k_{out})$$

The first line defines the parametric process of a client (with parametr l_{out}), which creates new port k_{in}, sends it via port l_{out} (the second step in the scenario), and receives a name from port k_{in} as j_{out}. Finally, the client process continues as the process representing a connected client (the last step in the scenario). The definition of the server process (the second and third line) is a recursive parallel composition, where the first part ensures permanency of the server process and the second part is similar to the definition of the client process (the first, third and the next steps in the scenario).

5 Verification of Systems of Asynchronous Concurrent Processes

Formal semantic of systems implemented using the framework described in this paper can be used to formal validation, testing equivalence or verifying general temporal and functional

[1]In the schema of the client-server scenario (Figure 4) links are used as simple point-to-point channels. But in real applications there will be many ports connected to every public link (e.g. many clients connected to one server).

properties. There are many papers about verification of systems of mobile processes expressed in π-calculus, such as [5], [6] or [7]. The next paragraphs will focus on using *The Mobility Workbench* (MWB, see [5]) for validation and verification of a system implemented in the framework.

Suppose we have a system implemented in the framework and its formal specification according to previous section of this paper. Using MWB we can for example:

- verify if processes are *strong and weak open bisimulation equivalent* (see [4]) — this is useful, e.g. after optimization or extension of a system implemented using the framework or its parts, when a proof of equivalence is needed,
- verify if a process contains *deadlocks* and obtain their descriptions — i.e. to check, if it is always possible to move the process under communication action to another state,
- *simulate* each step of system execution and *communicate* with the system using system interface (free channels) and interfaces of processes inside the system (non-free channels) — this can be used to debug a system, for fault finding in a given environment, and for black-box testing of a system, which is implemented using the framework.

5.1 Verification of Sample System

We use the outcome of Section 4.1 and express processes in π-calculus as input data for MWB (see Example 1). At first, the recursion in the parallel composition in process *Server*(l_{in}) has to be eliminated, because MWB couldn't handle it. In practice, the recursive parallel composition in *Server*(l_{in}) can be replaced with a finite number of concurrent processes (the "weaker" system can be sufficient to a simulation of executions with a predictable number of steps). For demonstrating purpose, the server from the example needs only one server process (no concurrency) for a communication with one client process (in the example, it is denoted by the comment "reduced"). Furthermore, processes $Client_{connected}$ and $Server_{connected}$ have to be defined and therefore, for demonstrating purpose, both of these processes will be specified as null processes.

```
agent Client(lout) = (^kin)'lout<kin>.kin(jout).ClientConn<kin,jout>
agent Server(lin) = lin(kout).P<kout> (*reduced*)
agent P(kout) = (^jin)'kout<jin>.ServerConn<jin,kout>
agent System = (^lin,lout)(Server<lin> | Client<lout> | Link<lin,lout>)
agent Link(in,out) = out(x).'in<x>.Link<in,out>
agent ClientConn(kin,jout) = 0
agent ServerConn(jin,kout) = 0
```

Example 1. Source code of the client-server system in MWB

The first three lines in Example 1 are only the transcription of π-calculus processes from Section 4.1. The fourth line defines the main process, i.e. the enclosed system containing the server, client and the link between them. The next line defines the link process, according to Section 4. The last two lines define connected versions of the client and server processes as null processes.

MWB finds in the main process *System* a deadlock reachable by three commitments, when a connection between the server and client is established and both the server and client finished, but the link process is ready to another communication (connected versions of client and server processes were finished, because of their definition as null processes). A real system should have real processes as connected versions of a client and server (and a server with many concurrent processes), which will be verified.

6 Conclusion

In this paper there was described a formal specification and verification of systems of asynchronous concurrent processes implemented using the object-oriented reusable framework for systems of asynchronous concurrent processes, which is based on a modified asynchronous network model [3]. Both the network model and the mentioned framework were briefly described with emphasis on formal specification and verification by means of the calculus of mobile processes (π-calculus) [4].

The framework for systems of asynchronous concurrent processes keeps specific level of abstraction, therefore semantics of a system designed using the framework cannot be expressed in π-calculus automatically, but has to be described during a design of such a system. In spite of this, the framework specifies design pattern, which a designer of system has to use and which leads to a system design expressible in π-calculus.

The formal description of a system implemented using the framework can be used to formal validation, testing equivalence or verifying general temporal and functional system properties. There exist tools and methods for processing formal specification in process algebras, such as [5], which is referred in this paper and can be used e.g. to prove strong and weak open bisimulation equivalence or the absence of deadlocks.

A future research will be aimed at concrete models of framework application to verify the framework and especially to improve way to formal specification and verification of systems implemented using this framework.

This work has been supported by the Grant Agency of Czech Republic grants No. 102/05/0723 "A Framework for Formal Specifications and Prototyping of Information System's Network Applications".

References

1. Sumit Ghosh. *Algorithm Design for Networked Information Technology Systems*. Springer. New York. 2004.
2. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers. San Francisco, CA, USA. 1996.
3. Marek Rychly. *A reusable framework for systems of asynchronous concurrent processes*. Available from http://www.fit.vutbr.cz/rychly/public/docs/reusable-framework-for-acp-systems/reusable-framework-for-acp-systems.pdf. 2006.
4. Robin Milner, Joachim Parrow, and David J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.
5. Victor Björn and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.
6. Ugo Montanari and Marco Pistore. Finite state verification for the asynchronous π-calculus. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 255–269, London, UK. Springer-Verlag, 1999.
7. Mads Dam. Proof systems for π-calculus logics. In R. de Queiroz, editor, *Logic for Concurrency and Synchronisation*, Trends in Logic, Studia Logica Library, pages 145–212. Kluwer, 2003.