International Conference on Innovative Views of the .Net Technology

Prototyping Concurrent Systems in Cw

Nuno F. Rodrigues, Luís S. Barbosa

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
{nfr, lsb}@di.uminho.pt

Abstract. Software architecture is currently recognized as one of the most critical design steps in Software Engineering. The specification of the overall system structure, on the one hand, and of the interactions patterns between its components, on the other, became a major concern for the working developer. Although a number of formalisms to express behaviour and supply the indispensable calculational power to reason about designs, are available, the task of deriving architectural designs on top of popular component platforms has remained largely informal. This paper introduces a systematic approach to derive, from behavioural specifications written in Cw, the corresponding architectural skeletons in the Microsoft .NET framework in the form of executable code.

1 Introduction[2]

1.1 Motivation: Behaviours for Architectures

This paper introduces a systematic approach to derive [10] prototype implementations from behavioural systems specifications written in the [11] process algebra. This complements and extends previous research by the authors in architectural prototyping in [6] documented in [14]. Before jumping in technicalities, however, let us first motivate the envisaged approach.

Over the last decade the specification of software architectures [5, 4] has been recognized as a critical design step in software engineering. Its role is to make explicit the underlying structure of a software system, identifying its components and the interaction dynamics among them, i.e., the *behavioural patterns* which characterize their *interactions*.

Classical process algebras (like, , [11] or [7]) on the other hand, emerged over the last thirty years as calculi to understand and reason about systems where interaction and concurrency play a significant, even dominant, role. It is not surprising that such

[2]The research reported in this paper is supported by FCT, under contract POSI/ICHS/44304/2002, in the context of the PURe project.

calculi, which embodied precise notions of behaviour and observational equivalence, as well as specific proof techniques, were often integrated in the design of generic *architectural description languages* (ADL). Typical examples are WRIGHT [1], based on , and DARWIN [9] or PICCOLA [8], which integrate a number of constructions borrowed from the -calculus [13, 12].

It is not the purpose of this paper to introduce a new description language for software architectures, not even to suggest additional features to existing languages. Our motivation is essentially *pragmatic*: suppose behavioural requirements for a given system are supplied as a collection of process algebra expressions; how can such requirements be incorporated on the design of a particular system? In other words, how can such requirements be animated and, which is even more important, how can they guide the overall design of the application architecture?

Our implementation target is the Cw programming language developed for the .NET framework [6] for component-based, distributed application design. Behavioural specifications, on the other hand, are written in the CCS [11] notation. The paper contribution is basically a strategy to implement such CCS specifications on top of Cw .NET. Rather than relying in a specific ADL, we resort to behavioural specifications in a particular process algebra to extract the overall structure of the system, identifying its active components with the declared processes, the interaction vocabulary, as recorded in the specification actions, and the, eventually, distributed, execution control, from the specification body.

The prototyping strategy proposed in this paper is described in section 2 and its application to a small example — the specification of the dinning philosophers problem — discussed in section 3. For quick reference, the next subsection provides a (rather terse) introduction to CCS and Cw.

1.2 : An Overview

The CCS notation [11] describes labelled transition structures interacting via a particular synchronization discipline imposed on the labels. Such synchronization discipline assumes the existence of *actions* of dual polarity (called *complementary* and represented as, , and), whose simultaneous occurrence is understood as a synchronous handshaking, externally represented by a non observable action .

Sequential, non deterministic behaviours are built by what in are called *dynamic* combinators: *prefix*, represented by a.P, where a denotes an action, for action sequencing, and *sum*, P+Q for non deterministic choice. The *inert* behaviour is represented by 0. Their formal semantics is given operationally by the following transition rules:

$$\frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \frac{E \xrightarrow{\alpha} E'}{E + F \longrightarrow E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$$

As shown by the rules above, dynamic combinators are sensible to transitions and disappear upon completion. Differently, *static* combinators persist along transitions, therefore establishing the system's architecture. This group includes the *parallel* composition, P|Q, and *restriction* new K P, where K is a set of actions declared internal to process P, i.e., not accessible from the process environment. Their operational semantics is as follows:

$$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'} \qquad \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$$

$$\frac{E \xrightarrow{\alpha} E'}{\mathbf{new}\,\{\beta\}\ E \xrightarrow{\alpha} \mathbf{new}\,\{\beta\}\ E'}\ (\text{if}\,\alpha \notin \{\beta, \overline{\beta}\})$$

On top of process terms a number of notions of observational equivalence are defined based on the capacity of processes to simulate each other behaviour (or an observable subset thereof). This entails a number of equational laws which form the basis of a rich calculus to reason and transform behavioural specifications. Such laws range, for example, from asserting the fact that both *sum* and *parallel* are abelian monoids, idempotent in the first case, to the powerfull *expansion law* which enables the unfolding of a process as a sum of all of its derivatives computed by the transition relation.

Typically, the architecture of a system composed of several processes running in parallel and interacting with each other is described by what is known in CCS as a *concurrent normal form*

$$\mathbf{new}\,K\ (P_1 \mid P_2 \mid ... \mid P_n)$$

where K is the subset of local (i.e., internal) actions (or communication ports) and each process has the shape of a non empty non deterministic choice between alternative execution threads.

Such a specification format seems to match reasonably well with the informal description of a software architecture as a collection of computational components (represented by processes P1 to Pn) together with a description of the interactions between them (represented by actions whose scope is constrained by the scope of the new operator. While this abstraction ignores some other fundamental aspects of architectural descriptions (namely non functional features such as performance measures or resource allocation), it provides a usefull starting point for the software engineer.

In such a context, the following sections discuss how such behaviour expressions can be prototyped in Cw to set the overall architectural structure of a software system.

Interestingly enough, as such description is based on a notation which supports a well-studied calculus, one becomes equipped with the right tools to transform architectural designs at very early phases of the design process.

1.3 Cw

Cw[10] is an extension to the language at two different levels: data type support for XML and table manipulation, on the one hand, and new asynchronous concurrency abstractions, based on the join calculus [3], on the other. The language brings to life a model of concurrency rich enough to be applicable both to multithreaded applications running on a single machine and to the orchestration of asynchronous, event-based components interacting over a (wide area) network.

The major contribution of Cw to concurrent programming, which constitutes our main interest in Cw, is the introduction of *chords* and a new mechanism for asynchronous method calls.

Chords contrast with normal methods where for each method declaration corresponds a body containing the code of its implementation. Thus, in a chord a method implementation can be associated to a set of methods. The code corresponding to a chord only executes when all the methods in the head of the chord have been previously called. Since chords may have return values, a problem pops up: which caller method should take the return value. The answer lies on a restriction in the definition of *chords*, where *chords* bearing return values can have at most one synchronous method in their head. Note however that this restriction to *chords* does not prohibit a *chord* to be composed of only asynchronous methods, which constitutes a useful construction for some behaviour implementations. *Chords* in this last case, cannot have any return values and their declarations must be preceded with the Cw keyword `when`.

Cw supplies C# with a new mechanism for asynchronous method calls that diverges from the one already present in C#. In C#, asynchronous method calls are decided by the calling method, therefore if one wants to call a method asynchronously it is him who decides to do so, by encapsulating the call with extra code, making use of C# threading capabilities. adds a new method call mechanism where it is the called method who decides how the call will behave, synchronously or asynchronously, and without the need for any extra threading code. Such methods which can decide that they can be called asynchronously have their own declarations preceded by the `async` keyword, and as explained above, they cannot have any return values.

The next section reports on the use of Cw as a target language for prototyping behavioural specifications. Experience shows that translations become closer to the corresponding CCS specification and smaller (in terms of the amount of code written). The details of this approach are outlined below.

2 From CCS to Cw

The process of prototyping CCS specifications in Cw is similar to the corresponding translation to C# [14]. In this case, however, implementations become more concise and faithful to their corresponding specifications.

This section focus on the prototyping process, starting from arbitrary CCS specifications of a system behaviour to derive its skeleton architecture in Cw. The qualification *skeleton* is a keyword here. Actually, we do not aim at deriving the whole system, but just resorting to the behavioural requirements, as expressed by the CCS specifications, to automatically derive the bare structures of implementations, i.e., their building blocks and corresponding interaction and synchronization restrictions.

Thus, one is not particularly concerned with the flow of actual values as arguments of methods or constructors, nor with how some eventually critical algorithms, specific to individual components, will perform. At this level, one is rather interested in issues like the way all processes communicate, what kind of messages do they pass to each others, what are their internal states at some point, how control flow is performed, how processes evolve in time and the implications of such evolutions in the other processes that also compose the system.

2.1 Processes

CCS Processes are implemented as Cw classes, with the same name as the corresponding CCS process identifier. It is inside this public class that all other process constructions are implemented. Bearing this in mind, the prototyping process is described in the sequel.

2.2 Actions

CCS actions are implemented either by methods, as before, or by *chords* to reflect cases of dependence on other ports or to maintain strict sequencing control in process execution. The distinction between the use of chords and methods and when to use one or the other will be made clear below.

The implementation of CCS actions in Cw falls in three different generic cases, depending on the nature of the ports inside the overall CCS system. Thus, one finds different implementation approaches for output ports with complementary input ports, input ports with complementary output ports and ports without any correspondent complementary port. These three cases will be discussed in detail in the following paragraphs.

2.2.1 Ports Without Complementary Ports

This is the simpler implementation case where a port without any complementary ports is implemented as a normal Cw method. For input ports, the Cw derived method bears the same name of the equivalent port in CCS .For output ports, the Cw method is preceded by a `c_` for output ports. Following is an example of an implementation of

a port p under these circumstances. The red italic parts of this example, as well as in subsequent examples, refers to code that changes according to the needs of the implementing system or of the context in which the method appears. In both cases, the text is self explanatory.

```
public void c_p() {
   p code
}
```

2.2.2 Input Ports With Complementary Output Ports

Input ports without complementary ports are implemented as chords of two methods, a synchronous and an asynchronous one. The asynchronous method is responsible for signaling that the complementary output port, of the one being implemented, is available for simultaneous execution. This asynchronous method has the name of the correspondent port in CCS preceded by `obs_`. The synchronous method is the one that truly implements the CCS port and is defined in a chord which depends on of the previous asynchronous method. The synchronous method has the exact name of the port being identified. The following example illustrates the implementation of a port p under these conditions.

```
public async obs_p();
public void p() & obs_p() {
    p code
}
```

2.2.3 Output Ports with Complementary Input Ports

Methods implementing output ports, with corresponding complementary ports that need to be executed simultaneously, wait to be called from the latter. A problem arising from the fact that this methods are being called is that if they are not allowed to execute (because of the sequential order imposed by the CCS specification to the process), they cannot make the current thread to stop. Otherwise the process in which the output port is implemented will never evolve making the parts of system that depend on it stopping too.

In order to overcome this undesirable effect associated to the requests from complementary input ports to their respective output ports, one introduces a new asynchronous method that captures the execution requests for these ports. Since asynchronous methods do not cause the calling thread to stop, systems evolution is no longer dependable on the requests for output ports. Even more, these asynchronous methods act like a request queue, where all the requests are stored and delivered as soon as some chord consumes them. The requests listener methods have the name of the output port, for which they receive requests, preceded by `allow_c_`. The next example presents the implementation of a generic port p under these conditions.

```
public async request_obs_p(object obj) {
    c_p(obj);
}
```

```
public void c_p(object obj) {
    c_p code

    if(obj is CallerType1) {
        (CallerType1 obj).obs_p();
    }
    .
    other possible requesters
    .
    if(obj is CallerTypeN) {
        (CallerTypeN obj).obs_p();
    }
}
```

The sequence of `if`'s statements in the example to test the type of the requesting object, is due to the inexistente of dynamic casts in . This way, at implementation time, one as to know which type of objects may perform requests to a certain port.

How requests for a specific output port arriving to a process are treated, i.e. whether they will be attended or not, is another issue which depends on the sequential execution of a process. This topic is discussed in the next section.

2.3 Sequential Behaviour

In CCS sequential behaviour is achieved through the prefix operator. Consider, for example, the specification of an elementary vending machine which receives a coin, retrieves a coffee and finally returns to the initial state:

$$M \equiv coin.\overline{coffee}.M$$

Note that the . operator of the above example is responsible for the fact that in process one can only perform action after action has been done. Even more, in process M the execution of action is immediately followed by a single execution of action .

It is imperative that such behaviours be strictly translated to the implementation of such a system. To guarantee the correct sequential execution of processes, two different cases must be addressed, namely input ports and output ports.

Methods implementing input ports are always called by their immediate preceding ports. This simple implementation rule guarantees by itself the correct sequential execution of input ports.

For methods implementing output ports the problem is a bit harder because these methods are not called by the ones which sequently precede them, but by any method which requires their use synchronization. They must therefore be guarded by a semaphore insuring the sequential evolution of the process. This semaphore is implemented by an asynchronous method (mentioned as `allow_c_p()` in the example) which is then bounded to the port resulting in a *chord*. This is illustrated in the following example for a generic port p with complementary port .

```
public async allow_c_p();

public void c_p(object obj) & allow_c_p() {
    c_p code
```

```
    if(obj is CallerType1) {
        ((CallerType1) obj).obs_p();
    }
    .
    other possible requesters
    .
    if(obj is CallerTypeN) {
        ((CallerTypeN) obj).obs_p();
    }
}
```

2.4 Reactions

The way reactions are implemented has already been partially revealed in the previous section. As mentioned above output ports with corresponding complementary ports in the specification wait to be called by the latter. Input ports, on the other hand, are always called by their predecessors and, therefore, are not required to be bounded to semaphores to ensure their correct sequential execution.

Nevertheless, input ports with corresponding complementary ports, which need to be executed simultaneously, are also implemented as chords to force simultaneity of action occurrence[9] . Such a port must perform a previous call to the asynchronous method `request_obs_Port()` in the process(es) holding its corresponding output port. This call acts like a request activation, signalising the beginning of an active waiting state.

2.5 Alternative Reactions

Alternative reactions are implemented by defining their initial ports as chords bounded to a semaphore (`alternative()`). This is a private asynchronous method which becomes available whenever the choice for the alternative reaction is also available.

With the introduction of alternative reactions, one is also opening space for none determinism to arise. However, it can be prototyped by performing random choices in the actions previous to the alternative reaction. Note, however, that one only needs to make random choices whenever the first execution port after an alternative choice is an input port, because output ports execution initiative is not determined by the process in which they are specified.

2.6 The Parallel Architecture

Every process prototype is equipped with an asynchronous `start()` method which wakes up the process and starts its execution. The parallel composition in the

[9] understood here as atomicity in the sense that both actions occur in an atomic way, that is, without being interleaved by other events.

CCS specification is then implemented by calling the `start()` method of each process composed in a parallel context.

3 Dining Philosophers Example

To illustrate the prototyping of systems in , a solution to the dining philosophers problem [2] was developed.

The dinning philosophers problem is a classical resource sharing problem. It consists of five philosophers sitting at a table with five forks and five dishes of food. Each philosopher can be in three different states. A first one where he is thinking, situation in which he is not handling any fork, a second one where he is trying to grab two forks in order to eat, and a third one where the philosopher is eating by holding two forks. Since each philosopher needs two forks to eat, there can be at most two philosophers eating at the same time. The deadlock situation in which each philosopher is grabbing one fork must be excluded from the solution.

Following the methodology presented in this paper, one starts from a CCS specification of the problem, which is hopefully self-explanatory.

$$
\begin{aligned}
P_{12} &\doteq \overline{think}.P_{12} + fork_1.fork_2.\overline{eat}.\overline{fork_1}.\overline{fork_2}.P_{12} \\
P_{23} &\doteq \overline{think}.P_{23} + fork_2.fork_3.\overline{eat}.\overline{fork_2}.\overline{fork_3}.P_{23} \\
P_{34} &\doteq \overline{think}.P_{34} + fork_3.fork_4.\overline{eat}.\overline{fork_3}.\overline{fork_4}.P_{34} \\
P_{45} &\doteq \overline{think}.P_{45} + fork_5.fork_4.\overline{eat}.\overline{fork_5}.\overline{fork_4}.P_{45} \\
P_{51} &\doteq \overline{think}.P_{51} + fork_5.fork_1.\overline{eat}.\overline{fork_5}.\overline{fork_1}.P_{51}
\end{aligned}
$$

$$
\begin{aligned}
F_1 &\doteq \overline{fork_1}.fork_1.F_1 \\
F_2 &\doteq \overline{fork_2}.fork_2.F_2 \\
F_3 &\doteq \overline{fork_3}.fork_3.F_3 \\
F_4 &\doteq \overline{fork_4}.fork_4.F_4 \\
F_5 &\doteq \overline{fork_5}.fork_5.F_5
\end{aligned}
$$

$$
C \doteq P_{12} \mid P_{23} \mid P_{34} \mid P_{45} \mid P_{51} \mid F_1 \mid F_2 \mid F_3 \mid F_4 \mid F_5
$$

Because of its size and details, it is not possible to present the entire Cw implementation here, but a representative implementation of each class of processes follows[10] .

[10]The complete implementation is available at `http://wiki.di.uminho.pt/wiki/bin/view/Nuno`

```
public class P12 {

    public Fork1 pfork1;
    public Fork2 pfork2;

    public async start() {
        c_think();
    }

    public void c_think() {
        Console.WriteLine("Phil_12 is thinking...");
        Random ran = new Random();
        System.Threading.Thread.Sleep(ran.Next(5000));

        double d = ran.NextDouble();
        if(d >= 0.5) {
            c_think();
        } else {
            pfork1.request_obs_fork1(this);
            fork1();
        }
    }

    public async obs_fork1();
    public void fork1() & obs_fork1() {
        Console.WriteLine("Phil_12 takes fork 1");
        pfork2.request_obs_fork2(this);
        fork2();
    }

    public async obs_fork2();
    public void fork2() & obs_fork2() {
        Console.WriteLine("Phil_12 takes fork 2");
        c_eat();
    }

    public void c_eat() {
        Console.WriteLine("Phil_12 is eating...");
        Random ran = new Random();
        System.Threading.Thread.Sleep(ran.Next(5000));
        allow_c_fork1();
    }

    public async request_obs_fork1(object obj) {
        c_fork1(obj);
    }

    public async allow_c_fork1();
    public void c_fork1(object obj) & allow_c_fork1() {
        Console.WriteLine("Phil_12 releases fork 1");
        if(obj is Fork1) {
            ((Fork1) obj).obs_fork1();
        }
        allow_c_fork2();
    }

    public async request_obs_fork2(object obj) {
        c_fork2(obj);
    }

    public async allow_c_fork2();
    public void c_fork2(object obj) & allow_c_fork2() {
        Console.WriteLine("Phil_12 releases fork 2");
        if(obj is Fork2) {
            ((Fork2) obj).obs_fork2();
        }
```

```
            Random ran = new Random();
            double d = ran.NextDouble();
            if(d >= 0.5) {
                c_think();
            } else {
                pfork1.request_obs_fork1(this);
                fork1();
            }
        }
    }


    public class Fork1 {

        public async start() {
            allow_c_fork1();
        }

        public async request_obs_fork1(object obj) {
            c_fork1(obj);
        }

        public async allow_c_fork1();
        public void c_fork1(object obj) & allow_c_fork1() {
            if(obj is P51) {
                ((P51) obj).obs_fork1();
                ((P51) obj).request_obs_fork1(this);
            }
            if(obj is P12) {
                ((P12) obj).obs_fork1();
                ((P12) obj).request_obs_fork1(this);
            }

            fork1();
        }

        public async obs_fork1();
        public void fork1() & obs_fork1() {
            allow_c_fork1();
        }
    }
```

It is also interesting to compare the implementation obtained by applying the presented method with the one present in the Cw documentation [10]. There, the boundaries of each identity in the system are not clear nor are the means by which they communicate with each other or the resource sharing is accomplished.

By taking our approach to the problem, one starts by specifying the solution in CCS, which is a concise and formal language in which the reasoning about the solution is focused on the important aspects of problem and not on possible implementation problems.

From this specification a implementation is derived which reflects many of the characteristics of the equivalent , notably a concise description of the solution, a clear definition of the intervening entities in the system and a precise notion of the behavioural evolution of each process taking part in the overall system.

This way, in the implementation of the dinning philosophers above, it is easy to inspect the involving entities, since each process corresponds to a different file, and analyse the different process behaviour at the light of the derivation rules. Communication between the different identities and their interrelation is also easily perceived

by inspecting the `main` function where references to the communicating processes are crossed and where the instances of the processes are created.

4 Conclusions and Future Work

The example discussed above is quite illustrative of the distance between architectural specifications, in suitable formalisms, and these implementations. The objective of this paper is to narrow such a gap.

A systematic method to derive implementations is already a step in this direction by eliminating possible (and probable) human implementation errors.

Our basic claim is that, at early phases of the software architecture design, the software architect should only need to worry about the actual architectural details of the problem. By reasoning in CCS and following the present method, the software architect is released from the burden of thinking about possible implementation details, that can but complicate the quest for a desirable architectural solution. Moreover it is well-known that the gap between architectural specifications and implementations only increases, at least linearly, with the complexity of the problems being addressed.

A limitation of the presented method and a good topic for future work, is the one arising from processes having the same port appearing in different choice branches. In these cases, a possible solution would involve the renaming of this port to a fresh name and the reflection of the renaming in the rest of the process implementation.

A second topic for future work is the extension of the proposed method in order to capture typed value-passing arguments between ports.

Finally, we are working on the development of a tool to implement the presented ideas, from CSS specifications to their derived implementations, extending the translator for documented in [14].

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[2] E. W. Dijkstra. Cooperating sequential processes. pages 65–138, 2002.

[3] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR' 96*. Springer Lect. Notes Comp. Sci. (1119), 1996.

[4] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors and events. In M. Bernardo and P. Inverardi, editors, *Third International Summer School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003)*. Springer Lect. Notes Comp. Sci, Tutorial, (2804), Bertinoro, Italy, September 2003.

[5] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering (volume I)*. World Scientific Publishing Co., 1993.

[6] E. Gunnerson. *A Programmer's Introduction to* . Apress, 2000.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[8] M. Lumpe. *A -calculus Based Approach to Software Composition*. PhD thesis, University of Bern, January 1999.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conference*, 1995.

[10] Microsoft Research. *Documentation*, 2004.

[11] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[12] R. Milner. *Communicating and Mobile Processes: the -Calculus*. Cambridge University Press, 1999.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[14] N. Rodrigues and L. S. Barbosa. Prototyping behavioural specifications in the .Net framework. In A. Mota and A. Moura, editors, *Proc. 7th Brazilian Symposium on Formal Methods (SBMF'2004)*, pages 108–118. UFP, November 2004.