Prototyping Behavioural Specifications in the .Net Framework

Nuno Rodrigues and Luís S. Barbosa
{nfr, lsb}@di.uminho.pt

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal

Abstract. Over the last decade, software architecture emerged as a critical design step in Software Engineering. This encompassed a shift from traditional programming towards the deployment and assembly of independent components. 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 Ccs, the corresponding architectural skeletons in the Microsoft .Net framework in the form of executable $C^\sharp$ code. Such prototyping process is automated by means of a specific tool developed in Haskell.

1 Introduction

1.1 Motivation: Behaviours for Architectures

In recent years the specification of software architectures [6, 5] 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, *e.g.*, Ccs [11] or Csp [8]) 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 suprising that such 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 Csp, and Darwin [10] or Piccola [9], which integrate a number of constructions borrowed from the π-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 suplied 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 .NET framework [7] 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 $C^\sharp$ .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 a control architecture for a road/railway cross — discussed in section 4. The systematic character of the approach proposed is tested by the possibility of rendering it automatic: section 3 describes a small tool for the derivation of $C^\sharp$ prototypes from CCS specifications. For quick reference, the next subsection provides a (rather terse) introduction to CCS.

1.2 CCS: 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, *e.g.*, α and $\overline{\alpha}$), whose simultaneous occurrence is understood as a synchronous handshaking, externally represented by a non observable action τ.

Sequential, non deterministic behaviours are built by what in CCS 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 $\mathbf{0}$. 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, $P \mid Q$, and *restriction* $\mathbf{new}\, K\ P$, 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 P_i 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 P_1 to P_n) 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 $C^\sharp$ 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.

2 Prototyping Behaviour in the .Net Framework

This section focus on the prototyping process, starting from arbitrary Ccs specifications of a system behaviour to derive its skeleton architecture in .Net. The qualificative *skeleton* is a keyword here. Actually, we do not aim to derive the whole system, but just to resort 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. Bearing this in mind, the prototyping process is described in the sequel.

2.1 Actions

An action in a Ccs specification corresponds to a method whose name is equal to the action's label in the corresponding implementation. Since such methods typically implement *input ports* in the system, they have invariably data type `void` as the domain of their return values. On the other hand, complementary actions specifying *output ports*, denoted in Ccs by an overline annotation, correspond to methods which may return values of any valid data type.

Accessibility restrictions on methods will be addressed later. For the moment, let us consider all these methods to be public. As an example, consider the following Ccs specification of a simple vending machine which receives a coin, performs an internal computation, retrieves a coffee and finally returns to the initial state:

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

In $C^\sharp$ the *coin* port will be implemented as

```
public void coin() { }
```

In the method body one would define later the corresponding computation which processes the coin reception.

On the other hand, the $\overline{coffee}$ port, which specifies an output port, will be translated as

```
public cof coffee() { }
```

declaring a method able to return a value of type `cof`. Of course, in this example, the choice of returning some value is rather optional, since the action of returning a coffee could be achieved inside the definition of the coffee method, by some internal computation, instead of returning the desired output.

2.2 Processes

Processes in Ccs correspond in $C^\sharp$ to identically named *classes*. Such classes encapsulate all the methods derived from the process ports specification. Therefore, in the previous example, one would get the following $C^\sharp$ class:

```
public class M {
    public void coin() { ... }
    public cof coffee() { ... }
}
```

Note that `class M` implements the *context* for the process, by declaring and grouping its two actions, but still, it does not capture the behaviour of the Ccs process M. In fact, there is no method invocation order subjacent to `class M`, whereas in process M one can only perform method `coffee()` after method `coin()` has been activated. Even more, in process M the execution of method `coin()` is immediately followed by a single execution of method `coffee()`. The specification does not allow, for example, that several calls to `coin()` precede the `coffee()` call or that several calls to `coffee()` follow a `coin()` insertion. Addressing such issues concerned with the process execution order requires some additional control flow code on the implementation side. Such is the topic of the following subsection.

2.3 Reactions

Prototyping sequential port activation, as typically specified in a Ccs expression, requires the introduction of an additional variable for state control. This auxiliary variable, denoted by `state` and simply declared of type string, contains the current state, captured by the name of the last executed method. Operationally, every method must inspect this variable to check whether its value is exactly the identifier of the port that precedes the current one.

For ports corresponding to initial actions on the Ccs specification, a slightly different approach is adopted. In such cases, the corresponding methods must check whether variable `state` is either null or contains one of the port identifiers from the set of ports that precede a (re-)execution of the current process.

The implementation of this process flow control mechanism requires the introduction of three basic functions which analyses the Ccs specification, namely `initialPorts(P)`, `precPorts(P)`, `finalPorts(P)`. Their purpose is to identify the initial, preceding and final actions on a Ccs expression, respectively. Once these functions evaluate, the rest of the implementation process falls into pretty-printing and Class accessibility control routines.

Nevertheless, one still has to prevent that no sequential ports execute simultaneously. To accomplish this, a method must first set the state variable to a particular temporary execution value (in the example the "`processing`" value is used), and release it at the end of its execution, a scheme similar to what is called a semaphore in classical concurrency control. This way, one not only guarantees that no sequential ports execute simultaneously, but also gets a way to

inspect the current state of a particular port. Note that any port in the system is either performing some computation (revealed by the value "`processing`" in the `state` variable) or prepared to be called.

Applying the above translation scheme to the example at hands results in the following $C^\sharp$ code:

```
public class M
{
    private string value;

    public void coin()
    {
        if(state != null || state.Equals("coffee"))
        {
            state = "processing";
             "code from the coin computations"
            state = "coin";
        }
        else { throw new Exception("Process sequence violation."); }
    }

    public cof coffee()
    {
        if(state.Equals("coin"))
        {
            state = "processing";
             "code from the coffee computations"
            state = "coffee";
        }
        else { throw new Exception("Process sequence violation."); }
    }
}
```

2.4 Alternative Reactions

Alternative reactions in a behavioural specification are achieved by the Ccs non deterministic choice combinator +. At the implementation level this combinator is regarded as a special sequence control. This is implemented on the analysis phase carried on by functions `initialPorts(P)`, `precPorts(P)` and `finalPorts(P)`, on which all process control flow stands, which are defined to deal correctly with the choice combinator + while evaluating over the inspected processes.

2.5 Restriction

Interaction restrictions within a process are handled in Ccs by the **new** combinator. Its implementation at the prototype level resorts to the accessibility mechanisms of the .Net platform. Thus, for every variable in the scope of a

Ccs restriction, the corresponding method is set to an `internal` method, rather than a `public` one, as used so far in our toy example.

With this additional step, methods declared `internal` become only available for classes inside the same assembly, isolating them from possible direct interactions with other classes.

Through accessability control, one may regard a .Net prototyping structure as a process execution domain, where every identifier lies within a precise execution scope. Again, a question remains: where should the boundaries of the system be set?

At a first glance one might think that processes are themselves good candidates for the boundary definition of the corresponding classes. This approach, however, would easily lead to a great amount of assemblies (one per process) without taking any direct advantage out of it, even because there can be no bounded variables at the level of the entire system. Thus, a minimalist approach is preferred, where one starts with only one assembly for the entire system, and then relies on each **new** occurrence in the Ccs expression to define process scopes and the corresponding bounded variables. Such scopes are created at implementation time leading to the construction of fresh assemblies with their methods correctly addressed in terms of accessibility.

By following this methodology for prototyping Ccs restrictions, one not only gets a correct isolation of process ports, but also specific process space domains within a system, which can be regarded as smaller (sub)systems of the overall architecture.

With the introduction of subsystems, another characteristic of typical architectural reasoning becomes explicit at the prototyping level: the ability to reason safely on simpler and isolated parts of the entire system.

2.6 The Parallel Architecture

The previous sections have shown how sequential Ccs processes can be correctly implemented in $C^\sharp$[1], but one is still missing the entire picture of a system composed of several interacting processes, as specified by a Ccs parallel expression.

To address this last issue two techniques are presented, a first one, where the execution of the system is totally controlled by a system's analyser, and a second one, closer to the execution model of Ccs, where processes evolve in time by internally reacting to each other until the system reaches a point where it requires interaction with the outside world.

Both ways of encapsulating an entire system and providing a simple way to test it, rely on the introduction of an additional class, called the *system interaction class*. This class encapsulates the entire system, exposing only its free variables and ensuring a correct execution order for all the assembled processes.

The first technique mentioned above relies on a single class with a single method which is able to deal with all the assembled processes. This requires

[1] Actually such a prototyping methodology can be tuned to any object-oriented language, or with some modifications, even to classical imperative ones.

that the state of all processes is kept and, that on every action occurrence, all possible interactions are checked, performing only one reaction at a time.

The second technique builds a system interaction class in a similar way, but for the fact that, for each action occurrence (and corresponding execution call), all the internal reactions are performed until the system stops for communication on an external input or output port or when facing a non deterministic control choice.

At this point, one might think that some of the previous presented strategies, addressing process restriction and correct process order reaction, were unnecessary since the system interaction class already addresses all this issues. However, the system interaction class should be regarded as a simpler way of interacting with the entire system, and not as the only way of interaction. At the prototyping level it is always possible, and even desirable, to make use of single processes or process's domains for interaction in order to test individual parts of the system or, in general, any of its sub-architectures.

3 The Automatic Translator

To automate the task of applying this methodology for deriving $C^\sharp$ architectural skeletons out of CCS specifications, a specific tool was developed in HASKELL.

The translator is based on a two phase procedure. The first phase consists of a parser for the CCS notation which converts the processes' specifications into a suitable HASKELL data type. The implementation of this phase procedure is achieved by the `CCSParser` HASKELL module, which resorts to the PARSEC libraries. Therefore, after the parsing stage, all CCS specifications are encoded in the following data type:

```
data Process a = Port a (Process a)
               | CompPort a (Process a)
               | Sum (Process a) (Process a)
               | Conc (Process a) (Process a)
               | New [a] (Process a)
               | RCall
               | PCall (ProcDef a)
               | ProcessEnd deriving Show

data ProcDef a = PDef (String, Process a) deriving Show
```

The `PDef` type constructor receives a pair containing a process identifier and the process definition itself. The process identifier will be used to define the class for the current process being implemented and also for cross reference calls between processes, specified by the type constructor `PCall` or by complementary port calls.

The data type `Process a` captures a CCS process definition, as presented in section 1, with a minor difference regarding CCS process call. In `Process a` process calls are distinguished in `RCall` for recursive calls to the defining process,

and `PCall` for process calls other then the one being defined. For the last option, one is implied to supply the process identifier and process definition of the process being called, in order to correctly defined inter-class calls at implementation level.

The second phase of the translator performs the calculation of the $C^\sharp$ implementation out of instances of data type `ProcDef a`. This second phase is implemented by the `CCS2DotNet` module, which includes the `buildSystem` function, responsible for the generation of the corresponding $C^\sharp$ code.

Function `buildSystem` receives an instance of data type `ProcDef a`, capturing a CCS system definition, and produces a series of files, each containing a $C^\sharp$ class definition (like the one in the example from next section) for each process defined in the CCS system.

The `buildSystem` function relies on several auxiliary functions, but three of them really constitute the building blocks where the entire Automatic Translator stands upon. These functions analyze the CCS specification and were already mentioned above as central functions for an automatic implementation. They are, respectively, `getFinalPorts`, which computes all the final ports of a given process, `getInitialPorts`, which computes all the available initial ports when a process executes and finally `portPreds`, which finds all the possible preceding ports of a given port in a given system.

4 An Example

As a small case study, consider the specification of a control system governing a crossing between a road and a railway. Notice this example, in despite of its small size, has a number of characteristics which are paradigmatic of the sort of systems this prototyping approach may be useful for. First of all it is a simple and effective system, concerned with a real world situation which embodies safety-critical requirements. Avoidance of deadlock and safe control flow are certainly properties which are required to be formally proved. This can be done within the CCS calculus. Once proved, our prototyping approach allows the software architect to derive an architectural skeleton of the final implementation which is, therefore, correct by construction.
We start with the following CCS specification, due to C. Stirling [14]:

$$\begin{aligned} Road \equiv & car.up.\overline{ccross}.\overline{dw}.Road \\ Rail \equiv & train.green.\overline{tcross}.\overline{red}.Rail \\ Signal \equiv & \overline{green}.red.Signal + \overline{up}.dw.Signal \end{aligned}$$

$$C \equiv new\{green, red, up, dw\}(Road|Rail|Signal)$$

The specification is self-explanatory: basically note that process *Signal* ensures the mutual exclusion of control access to both the (physical) semaphore controlling the railway and the gate governing the road traffic. The overall system is

specified by process C which, presented in the concurrent normal form, exposes the overall system's architecture.

To use the prototype derivator to automatically implement process C as a skeleton system architecture in .NET, one has to perform the two-phase procedure described in the previous section. For illustration purposes, we shall consider here process *Signal* in some detail, and abstract a little of the entire system, though some calls to other processes that interact with *Signal* will appear in the implementation. A similar procedure applies to the other processes.

The first step is to execute function `parseCCS` from module `CCSParser` to the string representation of the entire system. Again, we will focus on the *Signal* with the string representation

`Signal = /green.red.Signal + /up.dw.Signal`

which captures the definition of process *Signal*.

After parsing *Signal* process one gets the correspondent process value in terms of data type `ProcDef a`, *i.e.*,

```
signal = Sum (CompPort "green" (Port "red" RCall)) (CompPort "up" (Port "dw" RCall))

psignal = PDef ("Signal", signal)
```

Once the CCS system is defined as a value of the `ProcDef a` data type, one just has to apply function `buildSystem` to that value. Function

```
buildSystem :: ProcDef String -> IO ()
```

is responsible for creating all the files containing the various $C^\sharp$ classes as implementations of the received CCS process.

Function `buidSystem` relies on many other functions, many of them working exhaustively with strings and string manipulation. To improve the several operations over strings, data type `ShowS = String -> String` was preferred to the detriment of working over just `String`. The advantage of resorting to `ShowS` values, instead of directly working with domain `String`, is that functional composition with `ShowS` maintains linear complexity in functions dealing with many string concatenations.

The result returned by functions working over `ShowS` must then be stimulated with the initial action string (in this case the empty string), and the result written to a `.cs` file or passed to other function. The result of applying function `buildSystem` is the various .cs files implementing each process defined in the CCS specification. By taking a look at *Signal.cs* one would find the following

```
using System;

namespace CCS {
    public class Signal
    {
        private static string state;
```

```
        public static void greenComp(bool b)
        {
            if(state == null || state.Equals("red") || state.Equals("dw") )
            {
                if(!b) { Rail.green(true); }
                state = "processing";
                //(computational details to be supplied)
                state = "green";
            }
            else { throw new Exception("Process sequence violation."); }
        }

        public static void red(bool b)
        {
            if( state.Equals("green") )
            {
                if(!b) { Signal.red(true); }
                state = "processing";
                //(computational details to be supplied)
                state = "red";
            }
            else { throw new Exception("Process sequence violation."); }
        }

        public static void upComp(bool b)
        {
            if(state == null || state.Equals("red") || state.Equals("dw") )
            {
                if(!b) { Road.up(true); }
                state = "processing";
                //(computational details to be supplied)
                state = "up";
            }
            else { throw new Exception("Process sequence violation."); }
        }

        public static void dw(bool b)
        {
            if( state.Equals("up") )
            {
                if(!b) { Signal.dw(true); }
                state = "processing";
                //(computational details to be supplied)
                state = "dw";
            }
            else { throw new Exception("Process sequence violation."); }
        }
    }
}
```

From the code above one can inspect some important additional features that were not treated in previous sections that explained the used method.

The first flagrant unexpected feature is that every method receives a boolean value. This has to do with cross reference calls when treating calls to complementary actions. Its objective is to prevent that the system gets into infinite loop when complementary actions are called. To prevent such undesirable situations, every user call to a method must pass the `false` value as an argument. Only internal flow calls, regarding complementary actions use value `true` to call other complementary actions. This simple methodology guarantees that each method can inspect if it is being called by an internal call and therefore not needing to call the method that called him again from users calls that do need to check if there are complementary actions to be called.

Another unexpected feature is that the definition of specific computations inside each method implementing the process ports is left behind and signalized by the

```
//(computational details to be supplied)
```

marks. It is in this sense that our prototype implementations have a skeleton character. In any case, however, the underlying architecture specified in the Ccs expression has been translated to the .Net framework in a way which is both executable and guarantees, by construction, all the relevant safety-critical properties.

5 Conclusions and Future Work

As shown in the example just discussed, this paper proposes a simple, yet powerful, approach to the automatic derivation of $C^\sharp$ prototypes of behavioural specifications in Ccs. Such $C^\sharp$ code can be used in a number of different contexts. For example, applications developed under stateless environments which abound in the internet, with particular relevance to WebServices. Targeting this last paradigm, one can easily distribute processes in an (inter/intra)net and make use of SOAP to manage all external method calls.

The motivation is exactly the one typically invoked on the use of formal methods: first resort to a formal notation to enable precise expression of requirements and calculation power to discuss correctness and refinement. Then, derive executable prototypes in suitable implementation frameworks closer to the working programmer concerns.

We believe that the working programmer is more and more becoming the working software architect, whose job is essentially to look for suitable software components and plugging them in order to guarantee some desirable behaviour. If Ccs seems to be a sound and relatively well-known calculational formalism, .Net is becoming an almost *de facto* standard for implementing component based applications. The approach, however, is largely independent of the interaction discipline of Ccs: for example, Csp-like synchronizations, as used in some popular ADLs, or broadcast communication, can easily be incorporated as well.

In any case the emphasis is shifted from stand-alone programming to architectural design and, in such a sense, we believe the approach sketched in this paper may be found useful in practice. It should be mentioned that this ideas have been used in the context of a project on architectural reconstruction of legacy systems as well as in an undergraduate course on software architecture taught to third-year students of a Computer Science degree at Minho University.

Current work includes

- The generation of test classes and the derivation of a web-based interface for prototype testing.
- The extension of the prototyping approach to *mobile* applications, in which case behavioural specifications are to be given in the π-calculus.
- The integration of this approach in a methodology for formal specification of software architectures. This basically requires the construction of a library of specifications of typical *software connectors*, and corresponding `.Net` skeletons, able to be re-used in architectural design. Recall that a *software connector* [5, 4, 2] is an abstraction intended to represent the interaction patterns among components, the latter regarded as primary computational elements or information repositories. The aim of connectors is to mediate the communication and coordination activities among components, acting as a sort of glueing code between them. Examples range from simple channels or pipes, to event broadcasters, synchronization barriers or even more complex structures encoding client-server protocols or hubs between databases and applications. All of them can be easily specified in a process algebra notation (as in, *e.g.*, [1, 10]) and, therefore translated to `.Net` skeletons.

Finally, it should be mentioned that $C^\sharp$ itself is also evolving towards the integration of primitive distribution and concurrency control primitives at the language level [3]. This will certainly provide a richer environment for architectural prototyping.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.
2. M. A. Barbosa and L. S. Barbosa. Towards a relational model for component interconnection. In *SBLP'2004 (to appear in the* Jour. of Universal Computer Science*)*, Niteroi, Brasil, May 2004.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for $C^\sharp$. In *Proc. ECOOP 2002*. Springer Lect. Notes Comp. Sci. (2374), 2002.
4. J. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proc. of TAPSOFT'97*, pages 505–519. Springer Lect. Notes Comp. Sci. (1214), 1997.
5. 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.

6. 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.
7. E. Gunnerson. *A Programmer's Introduction to C♯*. Apress, 2000.
8. C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.
9. L. M. *A π-calculus Based Approach to Software Composition*. PhD thesis, University of Bern, January 1999.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conference*, 1995.
11. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
12. R. Milner. *Communicating and Mobile Processes: the π-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. C. Stirling. Modal and temporal logics for processes. *Springer Lect. Notes Comp. Sci. (715)*, pages 149–237, 1995.