DESIGNING PARALLEL SPECIFICATIONS IN CCS

Ken Stevens, John Aldwinckle, Graham Birtwistle, Ying Liu,
Department of Computer Science, University of Calgary, Canada.

Abstract

We describe a style of specifying concurrent systems based upon the parallel composition operator of CCS and apply it to several asynchronous hardware examples.

1 Introduction

In this paper we display a straightforward and flexible style for writing down the specifications of concurrent systems in CCS. CCS [4, 5] is a process algebra developed by Milner over the last 20 years. It permits object oriented descriptions of system components. Within itself, a CCS process may carry out a sequences of actions (using .), choose non-deterministically from several courses of actions (using +), or synchronize with another agent (using |). It is limited in scope in that it models interaction but not functionality. On the positive side it is a very sparse language with a very clean semantics. CCS scales hierarchically, and contains equational reasoning proof systems for analyzing behaviours and equalities. It has been used successfully to model and reason about network protocols and self-timed hardware.

Petri net or state graph representations for control and sequencing relations has been a traditional method for designing hardware in both synchronous and asynchronous disciplines. These pictures can be appealing because for simple designs they are easily understood. They match the intuition of the designer while showing ordering, parallelism, and causality between different signals. However, these techniques do not scale well pictorially, and usually lack the formalism necessary to compose and analyze complex systems of graphs for equality and behavioural properties.

In Section 2 we quickly introduce the CCS notation. Section 3 states our style of specification with an example. In Section 4 we apply this style to two token rings.

2 Background

CSS builds from the agent (process or object) *Nil* which can do nothing. From there CCS permits three ways of building more interesting agents.

Prefixing .

The agent $a.\bar{b}.Nil$ can do an a action, then a $\bar{b}$ action, and after that, nothing more. By convention, we overline all output actions. Recursive definitions are allowed; we can define a clock as:
$Clock \stackrel{def}{=} \overline{tick}.Clock$

This clock ticks forever.

When describing asynchronous hardware, the distinction between input transitions (a) and output transitions ($\bar{b}$) is significant. We have enough notation to describe a toggle:

$$Toggle \stackrel{def}{=} a.\bar{b}.a.\bar{c}.Toggle$$

After the first a input transition, the toggle outputs a transition on b. After the second a input transition, the toggle outputs a transition on c. This behaviour cycle then repeats.

Non-deterministic choice +

The C element accepts inputs on a and b in any order and then outputs on z. In CCS this is written

$$C \stackrel{def}{=} a.b.\bar{z}.C + b.a.\bar{z}.C$$

If the C element receives a transition on a, it evolves into the agent $b.\bar{z}.C$. If the C element receives a transition on b, it evolves into the agent $a.\bar{z}.C$. If the C element receives a transition on a and on b, it evolves into one of the above agents non-deterministically.

For brevity, we adopt a convenient notational shorthand and write the C element as $C \stackrel{def}{=} (a.b + b.a).\bar{z}.C$

As a second example,

$$Lock \stackrel{def}{=} \overline{free}.Lock + lock.unlock.Lock$$

can perform either a $\overline{free}$ action or a *lock* action. After a *free* action it evolves into a *Lock* again. After a *lock* action, it evolves into the agent *unlock.Lock* in which state it can only perform an *unlock* action.

Parallel composition |

We use the | operator to allow concurrent operation of agents. Parallel agents will be wired together and synchronize by communicating. A communication can occur when an action is an input to one agent and an output to another, and both are enabled.

0-7803-1443-3/93 $3.00

When a signal is hidden (syntactically via $\backslash\{\}$), the communication becomes local to those agents.

Suppose we wish to describe LC, a lockable C element [1]. When unlocked, this C element behaves like the C element above. When locked, it will still accept inputs, but will not output until unlocked. LC is defined as the parallel composition of two agents — $LockC$ is solely concerned with inputting on a and on b and firing z only when it is safe to do so; $Lock$ deals with the *lock/unlock* interactions and keeps the flag *free* 'waving' when it is in the unlocked state.

$$\begin{aligned} LockC &\stackrel{def}{=} (a.b + b.a).free.\overline{z}.C \\ Lock &\stackrel{def}{=} \overline{free}.Lock + lock.unlock.Lock \end{aligned}$$

$$LC \stackrel{def}{=} (LockC \mid Lock) \setminus \{free\}$$

We shield the line *free* from the environment by hiding ($\backslash\{free\}$). Now whenever the $LockC$ agent requires a transition on *free*, the only place it can get it from is the $Lock$ element, and $Lock$ cannot offer a transition on *free* after a *lock* but before an *unlock*.

It is quite possible to give a specification which spells out the state transitions one by one. Here is one:

$$\begin{aligned} LC &\stackrel{def}{=} LC0 \\ LC0 &\stackrel{def}{=} a.LC1 + b.LC9 + lock.LC5 \\ LC1 &\stackrel{def}{=} b.LC2 + b.LC3 + lock.LC6 \\ LC2 &\stackrel{def}{=} lock.LC4 + lock.LC7 + \overline{z}.LC0 \\ LC3 &\stackrel{def}{=} lock.LC4 + \overline{z}.LC0 \\ LC4 &\stackrel{def}{=} unlock.LC3 + \overline{z}.LC5 \\ LC5 &\stackrel{def}{=} a.LC6 + b.LC8 + unlock.LC0 \\ LC6 &\stackrel{def}{=} b.LC7 + unlock.LC1 \\ LC7 &\stackrel{def}{=} unlock.LC2 + unlock.LC3 \\ LC8 &\stackrel{def}{=} a.LC7 + unlock.LC9 \\ LC9 &\stackrel{def}{=} a.LC2 + a.LC3 + lock.LC8 \end{aligned}$$

Concurrent systems are notorious for the enormous number of states they can enter. The key observation is that specifications spelled out in steps using the . and + operators cannot avoid being long as they describe system evolution state by state. Experience has shown that descriptions written in terms of | can be very much shorter (linear in the number of interactions) and easier to understand because they can present the structure of the communications.

3 Specifying with the composition operator

We take the well-known RGDA arbiter as our running example in this subsection. Given two users of a resource both of which obey the protocol *NCS request grant CS done ack*, the RGDA arbiter ensures that only one will be in its critical section at a time.

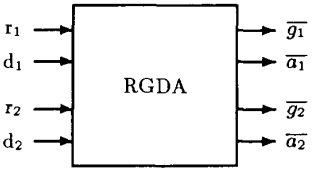

Figure 1: RGDA arbiter

Our proposed style has the following mechanical rules:

1. How many devices does this one connect to — name an agent responsible for each interaction.
   Two users: U_1 and U_2.
2. focus on each agent one by one and write down its sequence of interactions with the device being specified

$$\begin{array}{lllllllll} U_1 & = & r_1. & \overline{g_1}. & d_1. & \overline{a_1}. & U_1 \\ U_2 & = & r_2. & \overline{g_2}. & d_2. & \overline{a_2}. & U_2 \end{array}$$

3. list the timing constraints
   $\overline{g_1}.d_1$ and $\overline{g_2}.d_2$ must be mutually exclusive.
4. inject handshakes to enforce the necessary constraints

$$\begin{array}{lllllllll} U_1 & = & r_1. & \clubsuit. & \overline{g_1}. & d_1. & \heartsuit. & \overline{a_1}. & U_1 \\ U_2 & = & r_2. & \clubsuit. & \overline{g_2}. & d_2. & \heartsuit. & \overline{a_2}. & U_2 \\ S & = & & \clubsuit. & & & \heartsuit. & S & \end{array}$$

5. complete the formal specification by composing the agents and the constraints in parallel and hiding off any internal handshake lines
   $(U_1 \mid U_2 \mid S) \setminus \{g, p\}$
   where

$$\begin{aligned} U_1 &\stackrel{def}{=} r_1.gS.\overline{g_1}.d_1.\overline{pS}.\overline{a_1}.U_1 \\ U_2 &\stackrel{def}{=} r_2.gS.\overline{g_2}.d_2.\overline{pS}.\overline{a_2}.U_2 \\ S &\stackrel{def}{=} \overline{gS}.pS.S \end{aligned}$$

4 Martin's distributed arbiters

Instead of designing a monolithic arbiter, we look instead at a distributed arbiter, where the users are presumed to be spatially well-separated.

Each user goes through the request, grant, done, ack protocol at its own node. The nodes are connected together in a token ring and instead of contention between users, after a request, the user has to wait until the token appears. Martin describes three variants on the theme of a distributed arbiter in [3]. Here are the first two.

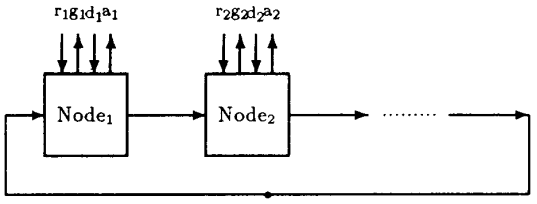

Figure 2: Token ring

4.1 Perpetuum mobile

In this arbiter, when the token reaches a node and there is a request, the token stays until the request is satisfied and then moves on. If there is no request, the token moves straight on. A deficiency with this design is that the token is in motion when there are no requests and this wastes energy.

We specify an individual node as a pair of agents operating in parallel.

$$IF \stackrel{def}{=} req.\overline{grant}.done.\overline{ack}.IF$$
$$TOK \stackrel{def}{=} tin.\overline{tout}.TOK$$

1. IF plays the role of a user following the request, grant, done, acknowledge protocol and
2. TOK plays the role of the token arriving, completing a service if need be, and then moving on.

The timing constraints are:

1. *req* arrives first: service, then pass token on.

   $req \quad tin \quad \overline{grant} \quad CS \quad done \quad \overline{ack} \quad \overline{tout}$
2. *tin* arrives first: no request to serve.

   $tin \quad \overline{tout}$

A more detailed specification is:

$$IF \stackrel{def}{=} req.\clubsuit.\overline{grant}.done.\heartsuit.\overline{ack} + \spadesuit.IF$$
$$TOK \stackrel{def}{=} tin.(\clubsuit.\heartsuit.\overline{tout}.TOK + \spadesuit.\overline{tout}.TOK)$$

where there are three rendezvous points:

1. $\clubsuit$ — when the request is accepted prior to the token arriving;
2. $\heartsuit$ — when the critical section is over and both IF and TOK may proceed independently
3. $\spadesuit$ — when the token arrives and there is no request, so all there is to do is pass the token on.

Notice that when the token arrives, IF will either be in state

- IF (no request as yet so that $\spadesuit$ is on offer), or
- $\clubsuit.\overline{grant}.done.\heartsuit.\overline{ack}.IF$ (a request has been accepted).

The translation into formal CCS is trivial:

$$IF \stackrel{def}{=} req.\overline{ok}.\overline{grant}.done.\overline{ko}.IF + \overline{no}.IF$$
$$TOK \stackrel{def}{=} tin.(ok.ko.\overline{tout}.TOK + no.\overline{tout}.TOK)$$
$$NODE \stackrel{def}{=} (IF \mid TOK) \setminus \{ok, no, ko\}$$

4.2 Reflecting privilege

Alain Martin suggested a variant distributed arbiter whose token does not cycle round the ring when there is nothing to do, but remains at the node where it last did some work until fetched. A drawback of this design is the possibility of livelock.

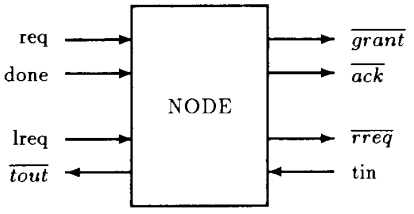

Figure 3: Single node

The token goes anti-clockwise round the ring and requests to fetch it go clockwise. The possible behaviours per node are:

1. The node has the token:
   - a request is accepted: the firing sequence is $req.\overline{grant}.done.\overline{ack}$
   - the token is requested on the left and passed on. The firing sequence is $lreq.\overline{tout}$
2. The node does not have the token:
   - a request is accepted, the token is fetched on the right, then the request is granted: The firing sequence is $req.\overline{rreq}.tin.\overline{grant}.done.\overline{ack}$
   - the token is requested on the left, is fetched on the right, and is then passed on: The firing sequence is $lreq.\overline{rreq}.tin.\overline{tout}$

As usual we specify a node in Martin's distributed arbiter network by interface processes IF and TOK, but this time add in a state machine FSM, which when fired, will fetch the token "on the right".

$$IF \stackrel{def}{=} req.\overline{grant}.done.\overline{ack}.IF$$
$$TOK \stackrel{def}{=} lreq.\overline{tout}.TOK$$
$$FSM \stackrel{def}{=} \overline{rreq}.tin.FSM$$

The FSM will be designed to assure that this node has the token before the TOK or IF processes may proceed. If the token is not present when a request is accepted, the FSM must fetch the token before allowing the requesting process to proceed. When both interfaces request service, the FSM arbitrarily decides whether to pass the token on, or process the request.

The synchronizations are described by

$$\begin{array}{lcl} IF & \stackrel{def}{=} & req.\clubsuit.\heartsuit.\overline{grant}.done.\blacksquare.\overline{ack}.IF \\ FSM & \stackrel{def}{=} & \clubsuit.fetch?.\heartsuit.\blacksquare.FSM \\ & + & \spadesuit.fetch?.\diamondsuit.FSM \\ TOK & \stackrel{def}{=} & lreq.\spadesuit.\diamondsuit.\overline{tout}.TOK \end{array}$$

where FSM splits into

$$\begin{array}{lcl} S_0 & \stackrel{def}{=} & \clubsuit.\overline{rreq}.tin.\heartsuit.\blacksquare.S_1 \; + \; \spadesuit.\overline{rreq}.tin.\diamondsuit.S_0 \\ S_1 & \stackrel{def}{=} & \clubsuit.\heartsuit.\blacksquare.S_1 \; + \; \spadesuit.\diamondsuit.S_0 \end{array}$$

in which there are five rendezvous points:

1. $\clubsuit$ — a user request arrives. The FSM is awakened and will be in state $S0$ (and will have to fetch the token) or in state $S1$ (has the token).
2. $\heartsuit$ — the node has the token and the user request may proceed. IF is woken up and the FSM lies dormant until the transaction is completed.
3. $\blacksquare$ — after the *done* signal is accepted, the FSM is set to state $S1$ and IF may send the *ack*.
4. $\spadesuit$ — a token request arrives. The FSM is woken up and will be in state $S0$ (and will have to fetch the token) or in state $S1$ (has the token).
5. $\diamondsuit$ — the node has the token and the token may be passed out. TOK is woken up and the FSM is set to state $S0$.

which are highlighted in Figure 4.

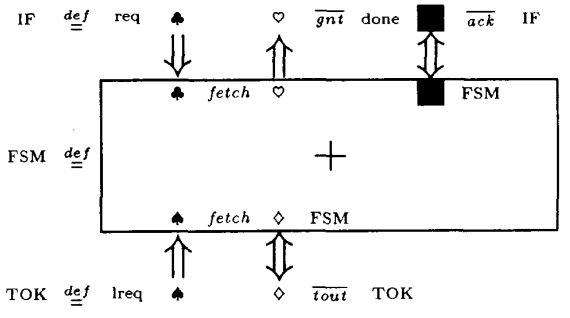

Figure 4: FSM synchronizations

Here is this specification in formal CCS:

$$IF \stackrel{def}{=} req.\overline{gT_0}.pT_0.\overline{grant}.done.\overline{sT_0}.\overline{ack}.IF$$

$$TOK \stackrel{def}{=} lreq.\overline{gT_1}.pT_1.\overline{tout}.TOK$$

$$\begin{array}{lcl} S_0 & \stackrel{def}{=} & gT_0.\overline{rreq}.tin.\overline{pT_0}.sT_0.S_1 \\ & + & gT_1.\overline{rreq}.tin.\overline{pT_1}.S_0 \\ S_1 & \stackrel{def}{=} & gT_0.\overline{pT_0}.sT_0.S_1 \\ & + & gT_1.\overline{pT_1}.S_0 \end{array}$$

**** node initialised to NO TOKEN ****

$$\begin{array}{lcl} N_0 & \stackrel{def}{=} & (IF \mid TOK \mid S_0) \\ & \setminus & \{gT_0, pT_0, sT_0, gT_1, pT_1\} \end{array}$$

**** node initialised to OWN TOKEN ****

$$\begin{array}{lcl} N_1 & \stackrel{def}{=} & (IF \mid TOK \mid S_1) \\ & \setminus & \{gT_0, pT_0, sT_0, gT_1, pT_1\} \end{array}$$

5 Conclusions

This paper presents a simple, intuitive method for mapping complex problem definitions into textual CCS specifications, representing signal relations, sequencing, and parallelism. These specifications can benefit from the formal methods of CCS while avoiding graphical or other non-scalable representations. CCS is perhaps a better model when comparing succinctness, scalability, and equational reasoning.

For further advantages concerning reasoning about specifications see the companion paper in these proceedings [2].

6 Acknowledgements

This research is supported by Equipment and Operating Grants from CMC and NSERC, and studentships from AGT/SEED (JA), Hewlett Packard (KS) and The Alberta Microelectronic Centre (YL).

References

[1] G. Gopalakrishnan. The lockable c element. Technical report, Computer Science Department, University of Utah, 1992.

[2] Y. Liu, J. Aldwinckle, G. Birtwistle, and K. Stevens. Testing the Consequences of Specifications in modal μ. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.

[3] A. Martin. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming*, 5:265–276, 1985.

[4] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[5] D. Walker. Introduction to a Calculus of Communicating Systems. Technical Report ECS-LFCS-87-22, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1987.