Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Transfer Request Broker: Resolving Input-Output Choice

Oliver FAUST, Bernhard H. C. SPUTH and Alastair R. ALLEN

Department of Engineering, University of Aberdeen, Aberdeen AB24 3UE, UK

{o.faust, b.sputh, a.allen}@abdn.ac.uk

Abstract. The refinement of a theoretical model which includes external choice over output and input of a channel transaction into an implementation model is a long-standing problem. In the theory of communicating sequential processes this type of external choice translates to resolving input and output guards. The problem arises from the fact that most implementation models incorporate only input guard resolution, known as alternation choice. In this paper we present the transaction request broker process which allows the designer to achieve external choice over channel ends by using only alternation. The resolution of input and output guards is refined into the resolution of input guards only. To support this statement we created two models. The first model requires resolving input and output guards to achieve the desired functionality. The second model incorporates the transaction request broker to achieve the same functionality by resolving only input guards. We use automated model checking to prove that both models are trace equivalent. The transfer request broker is a single entity which resolves the communication between multiple transmitter and receiver processes.

Keywords. guard resolution, hardware process networks, CSP, relation matrix.

Introduction

Communicating processes offer a natural and scalable architecture for distributed systems, because they make it possible to design networks within networks and the dependencies within these networks are always explicit. The process algebra CSP provides a formal basis to describe the processes and the communication between them. CSP semantics are powerful enough to allow the analysis of systems in order that failures such as non-determinism, deadlock and livelock can be detected. Within the CSP algebra external choice over input and output of a channel transaction are an important abstraction for the design of distributed systems. These constructs allow natural specification of many programs, as in programs with subroutines that contain call-by-result parameters. Furthermore, they ensure that the externally visible effect and behaviour of every parallel command could be modeled by some sequential command [1,2].

However, problems arise during the transition from theoretical CSP models to practical implementations. To ease this transition, Hoare [1] introduced the concept of channels. Channels are used for communication in only one direction and between only two processes. The combination of message and channel constitutes the output guard for the sender and the input guard for the receiver. This combination makes up a CSP event that constitutes the theoretical concept of a guard. External choice over input and output of a channel transaction describes the situation when sender processes offer sets of output guards and receiver processes offer sets of input guards for the environment to choose from. This situation is explicitly allowed

by the CSP process algebra and indeed necessary to ensure that every parallel command can be modelled by some sequential command [3]. Very early in the history of CSP Bernstein [4] pointed out the practical difficulties in the resolution of directed guards. In general external choice is possible between both channel inputs and outputs, but in this paper we consider only external choice of channel inputs for the implementation model. This concession is necessary to keep the solution simple enough so that it can be implemented in hardware logic.

As stated in the previous paragraph, to resolve output and input guards is an old problem. There are many proposed solutions for sequential or semi-parallel processing machines. For example, Buckley and Silberschatz [2] claim that they did an effective implementation for the generalized input-output construct of CSP. However, programming languages (such as occam-π [5]) and libraries (CTJ [6], JCSP [7,8], libCSP [9], C++CSP [10], etc), offering CSP primitives and operators do not incorporate these solutions. The main reason for this absence is the high computational overhead which results from the complexity of the algorithm used to resolve input and output guards [11]. To avoid this overhead, these programming environments provide synchronous message passing mechanisms to conduct both communication and synchronisation between two processes. Under this single phase commit mechanism, a sender explicitly performs an output of a message on a channel while the corresponding receiver inputs the message on the same channel. Such a communication will be delayed until both participants are ready. Choice is only possible over input channels; an outputting process always commits. In other words, only input guards can be resolved.

More recent attempts to solve choice problems involve an external entity. Welch et al. propose an external entity (oracle) which resolves choice between arbitrary multiway synchronisation events [12,13]. The advantage is that this method can be implemented with the binary handshake mechanism. The method was proposed for machine architectures in the form of an extension to programming languages and libraries. These extensions allow the implementation of sophisticated synchronisation concepts, such as the alting barrier [14]. However, the oracle uses lists to resolve the choice between arbitrary multiway sync events. These lists are not practical in hardware implementations, because in hardware process networks there is no dynamic allocation of memory. Similarly, Parrow and Sjödin propose a solution to the problem of implementing multiway synchronisation in a distributed environment [15]. Their solution involves a central synchroniser which resolves the choice.

This paper presents a solution for the choice resolution problem, suitable for implementation in hardware logic, that only supports input choice resolution. We introduce the Transfer Request Broker (TRB), an external entity which resolves output and input guards. The system relies entirely on a single phase commit mechanism. Therefore, no fundamental changes are required on the communication protocol. This solution is particularly sound for process networks implemented in hardware logic. These hardware processes are never descheduled and they never fail for reasons other than being wrongly programmed. Furthermore, all hardware processes are executed in parallel, therefore the TRB concept does not introduce delay caused by scheduling.

We approach the discussion in classical CSP style. Section 1.1 specifies the problem and Section 1.2 introduces a specification model. This model is a process network which incorporates the resolution of input and output guards. Section 1.3 discusses the choice resolution problem which results from the specification model with input and output guards. We provide evidence that input and output choice is not possible with a single phase commit mechanism. Therefore, the next step is to refine the specification into an implementation model which uses only input choice resolution. This refinement makes the implementation model less expressive but easier to implement. From a theoretical perspective, we have to increase the model complexity to compensate for this lack of expression. Section 1.4 details the implementation model which incorporates an additional TRB process. On the down side, the TRB process makes the implementation model more complex. On the upside, only choice resolution of

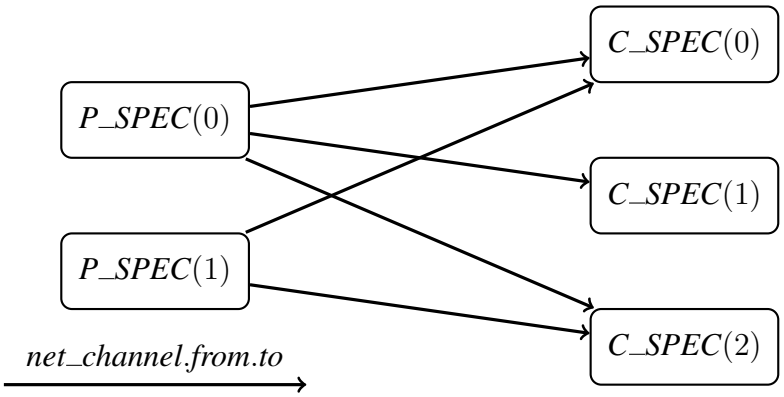

Figure 1. *SPECIFICATION* process network

input guards is required and this resolution can be achieved with the single phase commit mechanism. Having specification and implementation models leads us naturally to refinement checks. In Section 2 we prove that specification and implementation are trace equivalent. However, we could not establish the same equivalence for failures and divergences. This point is addressed in the discussion in Section 3. Also in the discussion section we introduce a practical example which shows the benefits of having the ability to allow external choice over input and output of a channel transaction in an implementation model. This also shows the practicality of the TRB concept. The paper finishes with concluding remarks.

1. Transfer Request Broker

The idea behind the TRB concept is to have a mediator or broker, which matches multiple inputs against multiple outputs, resolving those choices in a way that guarantees progress in the overall system where possible. The big advantage of this concept arises from the fact that all entities or nodes involved in the system need only comply with a simple binary handshake mechanism.

1.1. Problem Specification

We illustrate the problem by specifying an example process network which involves the resolution of input and output guards. The network incorporates receiver and transmitter processes which communicate over channels. External choice ensures that the environment can choose to transfer a message from one of multiple transmitters to one of multiple receivers.

In the specification environment a sender process can transfer messages over one of multiple output channels. Similarly, a receiver process is able to receive messages from one of multiple input channels. The *SPECIFICATION* process network, shown in Figure 1, depicts such a scenario. The process network contains two producer processes, *P_SPEC*(0) and *P_SPEC*(1). These producer processes generate messages and offer the environment the choice over which channels these are transferred to consumer processes. The *SPECIFICATION* process network contains three consumer processes, *C_SPEC*(0), *C_SPEC*(1) and *C_SPEC*(2). Figure 1 shows five *net_channel*s. These channels transfer messages from producer to consumer. The following list describes the network setup:

1. *net_channel*.0.0 connects *P_SPEC*(0) to *C_SPEC*(0);
2. *net_channel*.0.1 connects *P_SPEC*(0) to *C_SPEC*(1);
3. *net_channel*.0.2 connects *P_SPEC*(0) to *C_SPEC*(2);
4. *net_channel*.1.0 connects *P_SPEC*(1) to *C_SPEC*(0);
5. *net_channel*.1.2 connects *P_SPEC*(1) to *C_SPEC*(2).

Mathematically, the *net_channel*s establish a relation between producers and consumers. A relation exists when a channel between producer and consumer exists. We can express such a relation as a matrix where a '1' indicates a relation exists and a '0' indicates that no relation exists. We construct a matrix which relates producers to consumers in the following way: the 0s and 1s in a particular matrix row describe the connection of one producer. Similarly, the entries in a column describe the connection of a particular consumer. The network, shown in Figure 1, contains 2 producers and 3 consumers, therefore the relation matrix is 2 by 3. We define: row 1 contains the connections of producer $P\_SPEC(0)$ and row 2 contains the connections of producer $P\_SPEC(1)$. Furthermore, the connections of consumer $C\_SPEC(0)$ are described in column 1. Similarly, the connections of $C\_SPEC(1)$ and $C\_SPEC(2)$ are described in columns 2 and 3 respectively. These definitions result in the following relation matrix:

	$C\_SPEC(0)$	$C\_SPEC(1)$	$C\_SPEC(2)$
$P\_SPEC(0)$	1	1	1
$P\_SPEC(1)$	1	0	1

(1)

This relationship matrix provides the key to an elegant solution of the output and input guard resolution problem. But, before we attempt to explain the problem and solution in greater detail, the following section provides a formal model of the functionality we want to achieve.

1.2. Specification Model

A specification model is entirely focused on functionality. Therefore, we use the full set of abstract constructs provided by the CSP theory. To be specific, we use external choice for reading and writing. This leads to a very compact specification model which describes the desired functionality. This model serves two purposes. First, it allows the designer to focus on functionality. Second, the model provides a specification with which an implementation model can be compared.

The CSP model starts with the definition of the relationship matrix. This matrix describes the complete network, therefore it is called *net*. Equation 2 states the relation matrix for the network shown in Figure 1.

$$net = \begin{bmatrix} 1\,1\,1 \\ 1\,0\,1 \end{bmatrix} \tag{2}$$

The *net* matrix is the only parameter in the model, therefore all other constants are derived from this matrix. Equation 3 defines *data* as the set of messages which can be transferred over the *net_channel*s. To keep the complexity of the model low, the set contains only two messages $(0, 1)$.

$$data = \{0..1\} \tag{3}$$

The following two equations define the constants n as the number of rows and m as the number of columns of the *net* matrix respectively.

$$n = \text{dim\_n}(net) \tag{4}$$

$$m = \text{dim\_m}(net) \tag{5}$$

where the function $\text{dim\_n}(A)$ extracts the number of rows from a matrix A and $\text{dim\_m}(A)$ extracts the number of columns of a matrix A.

Next, we interpret the process network, shown in Figure 1, in a process-centric way. Each producer is connected to a set of consumers. This consumer set is entirely governed by the indexes of the consumers, therefore it is sufficient to state only the consumer IDs in the index set. The following function extracts the connection set of a particular producer process $P(i)$.

$$\text{p\_set}(i) = \text{locate}(\text{get\_n}(i, net), 1) \tag{6}$$

where the function $\text{get\_n}(i, A)$ extracts the row vector i from matrix A and the function $\text{locate}(vec, val)$ returns a set with the positions of val in the vector vec.

The following function extracts the connection set of a particular consumer process $C(j)$.

$$\text{c\_set}(j) = \text{locate}(\text{get\_m}(j, net), 1) \tag{7}$$

where the function $\text{get\_m}(j, A)$ extracts the column vector j for the matrix A.

After having defined all necessary constants and helper sets, we start with the process definitions. The first process to be defined is $P\_SPEC(i)$, where i is the process index. The process reads a message x from the input channel $in.i$. After that, $P\_SPEC(i)$ is willing to send the message over one of the *net_channel*s which connects the producer process to a consumer process. After the choice is resolved and the message is sent, the process recurses.

$$P\_SPEC(i) = in.i?x \rightarrow \Box_{j \in \text{p\_set}(i)} \underbrace{net\_channel.i.j!x}_{\text{output guards}} \rightarrow P\_SPEC(i) \tag{8}$$

where $\Box_{j \in \text{p\_set}(i)} net\_channel.i.j!x$ indicates an indexed external choice over all connected *net_channel*s. The channel-messages which are sent out constitute the output guards for particular *P_SPEC* processes.

The $C\_SPEC(j)$ process waits for a message from one of its connected *net_channel*s. After having received a message x the process is willing to send it on via the output channel $out.j$ before it recurses.

$$C\_SPEC(j) = \Box_{i \in \text{c\_set}(j)} \underbrace{net\_channel.i.j?x}_{\text{input guards}} \rightarrow out.j!x \rightarrow C\_SPEC(j) \tag{9}$$

The channel-messages which are sent out constitute the input guards for particular *C_SPEC* processes.

The specification system composition follows the process network diagram shown in Figure 1. We start by observing that the producers, in this case $P\_SPEC(0)$ and $P\_SPEC(1)$, do not share a channel. That means they can make progress independently from one another. In Equation 10 we use the interleave operator '$|||$' to model this independence. Similarly, the consumers exchange no messages. Therefore, Equation 11 combines all consumer processes $C\_SPEC(j)$ into the *CONSUMER_SPEC* process with the interleave operator.

$$PRODUCER\_SPEC = \Big|||_{i \in \{0..n-1\}} P\_SPEC(i) \tag{10}$$

$$CONSUMER\_SPEC = \Big|||_{j \in \{0..m-1\}} C\_SPEC(j) \tag{11}$$

where $\Big|||_{i \in \{0..1\}} P\_SPEC(i)$ represents the indexed interleave operator which expresses: $P\_SPEC(0) \;|||\; P\_SPEC(1)$.

The *SPECIFICATION* process combines *CONSUMER_SPEC* and *PRODUCER_SPEC*. In this case the processes which are combined depend on one another to make progress. To be specific, the *PRODUCER_SPEC* process sends messages to the *CONSUMER_SPEC* process via the *net_channel*s. We say that *PRODUCER_SPEC* and *CONSUMER_SPEC* execute in parallel, however they have to agree upon all messages which can be transferred via the *net_channel*s. Equation 12 models this behaviour by combining *PRODUCER_SPEC* and *CONSUMER_SPEC* via the alphabetised parallel operator.

$$SPECIFICATION = CONSUMER\_SPEC \underset{\{|net\_channel|\}}{\Big\|} PRODUCER\_SPEC \quad (12)$$

where $\|_{\{|net\_channel|\}}$ is the alphabetised parallel operator. The expression $\{|\ net\_channel\ |\}$ indicates the set of all events which can be transferred over *net_channel*, i.e.:

$$\{|\ net\_channel\ |\} = \{net\_channel.i.j.x \mid i \in \{0..1\}, j \in \{0..2\}, x \in \{0..1\} \wedge \neg((i = 1) \wedge (j = 1))\}$$

The definition of the *SPECIFICATION* process concludes our work on the functionality model. We have now a model against which we can measure any implementation model.

1.3. Transfer Request Broker Model

In the specification model external choice allows the environment to choose over which channel *P_SPEC*(i) outputs a message and the same operator enables the environment to choose over which channel *C_SPEC*(i) inputs a message. The fundamental problem for an implementation system is: Who resolves this choice? Neither producers (*P_SPEC*) nor consumers (*C_SPEC*) are able to perform this task with a single phase commit algorithm. To support this point, we analyse three different transfer scenarios which could occur in the process network shown in Figure 1.

First, we assume that the choice is resolved by the producer and the consumer engages in any one transfer which is offered. We dismiss this approach with the following counter example. The producer process *P_SPEC*(0) has resolved that it wants to send a message to consumer process *C_SPEC*(2). At the same time the producer process *P_SPEC*(1) has resolved that it wants to send a message to the same consumer process *C_SPEC*(2). Now, according to our assumption *C_SPEC*(2) is not able to resolve the choice between the messages offered by *P_SPEC*(0) and *P_SPEC*(1), at the same time it is impossible to accept both messages. Clearly, this counter example breaks the approach that the producers resolve the choice.

The second approach is that the consumer resolves the choice and the producer reacts to the choice. In this case, the counter example is constructed as follows: *C_SPEC*(0) and *C_SPEC*(1) have resolved to receive a message from *P_SPEC*(1). Now, *P_SPEC*(1) faces a similar dilemma as *C_SPEC*(2) in the previous example. According to the assumption that the consumer resolves the choice, *P_SPEC*(1) is not allowed to choose and it is also not allowed to service both requests.

The last scenario gives both consumer and producer the ability to resolve the choice. We break this scheme with yet another counter example. *P_SPEC*(0) tries to send to *C_SPEC*(0). However, *C_SPEC*(0) blocks this message, because it has resolved the input choice such that it wants to receive a message from *P_SPEC*(1). In this example, *P_SPEC*(1) is unwilling to send a message to *C_SPEC*(0), because it has resolved to output a message to *C_SPEC*(2). Unfortunately, *C_SPEC*(2) has decided to wait for a message from *P_SPEC*(0). However, this message will never come, because *P_SPEC*(0) tries to send to *C_SPEC*(0). *A classical deadlock.*

This small discussion shows that there is no *simple* solution to the choice resolution problem. The fundamental problem is that individual processes are not aware of what happens around them in the network. In other words, a process with an overview is missing. These considerations lead to the proposal of a *TRB* process. This *TRB* process is aware of the network state. This additional process allows us to use the simple receiver choice resolution method. That means no process in the network incorporates external choice for writing.

For example, the *SPECIFICATION* process network can be in one of up to $243\,(= 3^5)$ different states (5 fairly independent processes, each with 3 states). To achieve the awareness of the network state the *TRB* process has to know the state of the individual producer and

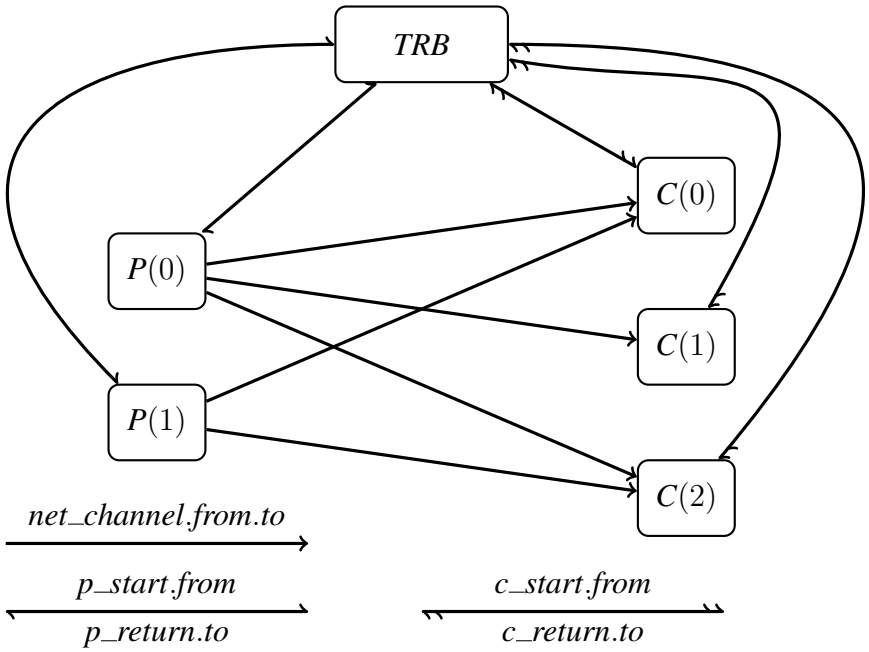

Figure 2. *IMPLEMENTATION* process network

consumer processes. In the following text the group of consumers and producers are referred to as clients of the *TRB*. The clients transfer their state to the *TRB* via *p_start.from* and *c_start.from* channels. The *TRB* communicates its decision via *p_return.to* and *c_return.to* channels.

Figure 2 shows the *IMPLEMENTATION* process network. Each client process is connected to the *TRB* via a *start.from* and a *return.to* channel. The message passing channels *net_channel.from.to* connect producer processes with consumer processes. The relation matrix of Equation 1 describes the network setup.

1.4. CSP Implementation Model

Similar to the definition of the specification model, we adopt a bottom up design approach for the definition of the CSP implementation model. First, we define the individual processes, shown in Figure 2, then we assemble the *IMPLEMENTATION* process network. The model uses the network matrix, defined in Equation 2, and the constants defined in Equations 3 to 5.

The first process to be defined is the producer process $P(i)$. Initially, this process waits for a message from the input channel *in.i*. After having received a message x the process registers with the *TRB*. The registration is done by sending a vector of dimension m to the *TRB* via the *p_start.i* channel. The entries in this vector are either 0 or 1. A 1 indicates that the producer process wants to send a message. The position of the 1 within the vector indicates the consumer address. The *TRB* resolves the request and returns the index of the selected consumer via a message over the *p_return.i* channel. After having received the index (*select*) of the consumer process the $P(i)$ process is ready to send out the message x over the channel *p_return.i*?*select* before it recurses. The following equation defines the producer process $P(i)$.

$$\begin{aligned} P(i) = {} & in.i?x \rightarrow p\_start.i!\text{get\_n}(i, net) \\ & \rightarrow p\_return.i?select \rightarrow net\_channel.i.select!x \rightarrow P(i) \end{aligned} \tag{13}$$

where $\text{get\_n}(i, A)$ returns the ith row of matrix A.

Initially, the consumer process $C(j)$ is ready to register with the *TRB*. In this case registration means sending the jth column vector of the network matrix *net*. This indicates that

$C(j)$ is willing to receive a message from any one of the connected input channels. The *TRB* resolves this request and sends the result back to the $C(j)$ process via the *c_return.j* channel. The result is the index (*select*) of the producer which is selected to send a message. Now, the consumer process waits on the particular *net_channel.select.j* channel. After having received a message from this channel the consumer sends out the message over the *out* channel before it recurses. The following equation defines the consumer process $C(j)$.

$$\begin{aligned} C(j) = {} & c\_start.j!\text{get\_m}(j, net) \\ & \rightarrow c\_return.j?select \rightarrow net\_channel.select.j?x \rightarrow out.j!x \rightarrow C(j) \end{aligned} \tag{14}$$

The last and most complex process defines the TRB functionality. Indeed this task is so complex that it is broken into 5 processes. The first of these processes is $TRB(p\_array, c\_array)$. The *p_array* parameter is an $n \times m$ matrix which holds the information about the registered producers. Similarly, the *c_array* parameter is an $n \times m$ matrix which holds the information about the registered consumers. Initially, the *TRB* process is ready to receive the registration information from any one of its clients. This is modelled as external choice over two indexed external choices. The first indexed external choice chooses between *p_start* messages and the second indexed external choice chooses between *c_start* messages. If the complete choice construct is resolved and a message is transferred via the *p_start* channel, then *p_array* is updated with the registration information before the process behaves like $EVAL(p\_array, c\_array, 0, 0)$. To be specific, *p_array* is updated by replacing the *i*th row, where *i* is the index of the producer which registered, with the registration vector *p_vector*. As a matter of fact, *p_vector* and the channel index represent the registration information. Similarly, if the external choice over two indexed external choices is resolved in favour of a *c_start* message then *c_array* is updated with the registration information before the process behaves like $EVAL(p\_array, c\_array, 0, 0)$.

$$\begin{aligned} & TRB(p\_array, c\_array) = \\ & \left(\begin{array}{l} \Box_{i \in \{0..n-1\}} \; p\_start.i?p\_vector \rightarrow EVAL(\text{set\_n}(i, p\_vector, p\_array), c\_array, 0, 0) \\ \Box \\ \Box_{j \in \{0..m-1\}} \; c\_start.j?c\_vector \rightarrow EVAL(p\_array, \text{set\_m}(j, c\_vector, c\_array), 0, 0) \end{array} \right) \end{aligned} \tag{15}$$

where the function $\text{set\_n}(i, vec, A)$ replaces the *i*th row vector of matrix *A* with *vec* and the function $\text{set\_m}(j, vec, A)$ replaces the *j*th column vector of matrix *A* with *vec*.

The $EVAL(p\_array, c\_array, i, j)$ process compares the entries in *p_array* with the entries in *c_array*. The parameters *i* and *j* describe row and column of the entry to be compared. If both *p_array* and *c_array* have a 1 at position i, j then the process resolves the choice such that producer $P(i)$ has to send a message to consumer $C(j)$. We model this case with the process $RETURN\_P(p\_array, c\_array, i, j)$. In all other cases the process continues to compare entries at other matrix positions. This is done by checking whether or not *j* defines the position of the last entry in a row, i.e. $j = m - 1$. If this is the case then the process behaves like $TRB'(p\_array, c\_array, i, j)$, in all other circumstances the column index is incremented $(j = j + 1)$ before the process recurses.

$$\begin{aligned} & EVAL(p\_array, c\_array, i, j) = \\ & \left(\begin{array}{l} \textbf{if } \text{get\_nm}(i, j, c\_array) = 1 \textbf{ and } \text{get\_nm}(i, j, p\_array) = 1 \textbf{ then} \\ \quad RETURN\_P(p\_array, c\_array, i, j) \\ \textbf{else if } j = m - 1 \textbf{ then} \\ \quad TRB'(p\_array, c\_array, i, j) \\ \textbf{else} \\ \quad EVAL(p\_array, c\_array, i, j + 1) \end{array} \right) \end{aligned} \tag{16}$$

where the function get_nm(i,j,A) returns the entry $a_{i,j}$ of matrix A.

The $RETURN\_P(p\_array, c\_array, i, j)$ sends out the message j, which is the resolution result, to the producer $P(i)$ via the channel $p\_return.i$. After the message is transferred the process behaves like $RETURN\_C(p\_array, c\_array, i, j)$. In effect, clearing all 1s in row i indicates that $P(i)$ is unable to send out any more messages in this resolution round.

$$\begin{aligned} &RETURN\_P(p\_array, c\_array, i, j) = \\ &\quad p\_return.i!j \rightarrow RETURN\_C(\text{set\_n}(i, \text{vzeros}(m), p\_array), c\_array, i, j) \end{aligned} \tag{17}$$

where the function vzeros(m) generates an m dimensional zero vector.

The $RETURN\_C(p\_array, c\_array, i, j)$ sends out the message i, which is the resolution result, to the consumer $C(j)$ via the channel $c\_return.i$. After the message is transferred the process clears all 1s in column j of c_array and behaves like $TRB'(p\_array, c\_array, i, j)$. In effect, clearing all 1s in column j indicates that $C(j)$ is unable to receive any further messages in this resolution round.

$$\begin{aligned} &RETURN\_C(p\_array, c\_array, i, j) = \\ &\quad c\_return.j!i \rightarrow TRB'(p\_array, \text{set\_m}(j, \text{vzeros}(n), c\_array), i, j) \end{aligned} \tag{18}$$

The $TRB'(p\_array, c\_array, i, j)$ detects whether or not the entries in the last row were checked. If this is the case, then the process behaves like $TRB(p\_array, c\_array)$. In all other circumstances the row index i is incremented and the column index j is set to 0 before the process recurses to $EVAL(...)$ for another round of entry checking.

$$TRB'(p\_array, c\_array, i, j) = \left(\begin{array}{l} \textbf{if } i = n-1 \textbf{ then} \\ \quad TRB(p\_array, c\_array) \\ \textbf{else} \\ \quad EVAL(p\_array, c\_array, i+1, 0) \end{array} \right) \tag{19}$$

The *TRB* definition concludes the description of the individual processes in the *IMPLEMENTATION* process network. In the next step we connect the individual processes such that they form the process network, as shown in Figure 2. From this figure we observe that the *TRB* process and the additional channels do not alter the fact that the producers do not communicate among themselves. Therefore, they form an independent group of processes. This group is modelled as the *PRODUCER* process in Equation 20. Similarly, the consumers form a group of independent processes. This group is established as the *CONSUMER* process in Equation 21.

$$PRODUCER = \Big|\Big|\Big|_{i \in \{0..n-1\}} P(i) \tag{20}$$

$$CONSUMER = \Big|\Big|\Big|_{j \in \{0..m-1\}} C(j) \tag{21}$$

The parallel combination of *PRODUCER* and *CONSUMER* forms the *NETWORK* process. The two processes, which engage in the parallel construct, must agree on all messages sent over the *net_channel*s. This is modelled in the following equation:

$$NETWORK = PRODUCER \underset{\{|net\_channel|\}}{||} CONSUMER \tag{22}$$

Now, we are able to define the *IMP* process as the parallel combination of *NETWORK* and *TRB* processes. Equation 23 shows that the *TRB* process is initialised with two zero matrices, i.e. both p_array and c_array are initialised with $n \times m$ zero matrices.

$$IMP = NETWORK \underset{\{|p\_start, c\_start, p\_return, c\_return|\}}{||} TRB(\text{zeros}(n,m), \text{zeros}(n,m)) \tag{23}$$

where the function $\text{zeros}(n,m)$ returns an $n \times m$ zero matrix. The alphabetised parallel operator ensures that all clients can register with and receive a solution from the *TRB* process.

$$IMPLEMENTATION = IMP \setminus \{| \, p\_start, c\_start, p\_return, c\_return \, |\} \tag{24}$$

where $P \setminus \{a\}$ is the hiding operation which makes all events (messages) a internal to P. Hiding the communication to and from the *TRB* is necessary for model checking.

For the sake of clarity this section introduced particular scenarios, i.e. network setups to explain the TRB concept. These particular network setups do not prove that the concept works with other network configurations. To build up trust, the algorithm was tested with other network setups. The tests are described in the following section. These tests were successful, therefore confidence is growing that the algorithm works for arbitrary network configurations which can be described by a relation matrix.

2. Automated Refinement Checks

We use the FDR tool [16] to establish that both *SPECIFICATION* and *IMPLEMENTATION* do not contain any pathological behaviours. This is done in the following section. Furthermore, we use FDR to compare *SPECIFICATION* and *IMPLEMENTATION*. This is done in Section 2.2.

2.1. Basic Checks: Deadlock, Divergence and Non-determinism

Parallel and concurrent systems can exhibit pathological behaviours such as deadlocks and livelocks. These problems arise from the fact that in such systems independent entities communicate. To be specific, a deadlock occurs if two or more independent entities prevent each other from making progress. A livelock occurs when a system can make indefinite progress without engaging with the outside world. CSP models the independent entities as processes which communicate over channels. Equation 25 instructs the automated model checker FDR to verify that the *SPECIFICATION* process is deadlock and divergence (livelock) free.

$$\begin{array}{l} \textbf{assert } SPECIFICATION \text{ :[deadlock free [F]]} \\ \textbf{assert } SPECIFICATION \text{ :[divergence free]} \end{array} \tag{25}$$

In the CSP community it is custom to publish the output of the model checker to support the claim that a particular test was successful. Figure 3 shows the output of the model checker. A ✓ indicates that this particular test was successful.

In Equation 26 we instruct FDR to verify that the *IMPLEMENTATION* process is deadlock and divergence free.

$$\begin{array}{l} \textbf{assert } IMPLEMENTATION \text{ :[deadlock free [F]]} \\ \textbf{assert } IMPLEMENTATION \text{ :[divergence free]} \end{array} \tag{26}$$

Figure 3 shows that both tests were successful.

As a final test on individual processes we establish whether or not a process is deterministic. A process is deterministic if it reacts in the same way to the same input. In the following equation we instruct FDR to verify that both *SPECIFICATION* and *IMPLEMENTATION* processes are deterministic.

$$\begin{array}{l}\textbf{assert } \textit{SPECIFICATION} \text{ :[deterministic [FD]]} \\ \textbf{assert } \textit{IMPLEMENTATION} \text{ :[deterministic [FD]]}\end{array} \tag{27}$$

Figure 3 shows that the *IMPLEMENTATION* process is not deterministic. Hiding the communication to and from the *TRB* process causes this behaviour. To support this statement, in Equation 28 we test whether or not the *IMP* process is deterministic. The *IMP* process is the *IMPLEMENTATION* with no hidden communication.

$$\textbf{assert } \textit{IMP} \text{ :[deterministic [FD]]} \tag{28}$$

Figure 3 shows that *IMP* is deterministic.

2.2. Trace Refinement

The refinement operation relates two processes. There are different types of refinement, such as trace and failure refinement. In this section we consider only trace refinement. Trace refinement tests the safety of a particular process. A trace is a sequence of events observed by the outside world (outside of the process itself). A system is safe if and only if the *IMPLEMENTATION* can only exhibit a subset of the traces from the *SECIFICATION*. In other words, the *IMLEMENTATION* refines the *SPECIFICATION*.

In this particular case we establish that the *IMPLEMENTATION* process can exhibit the same traces as the *SPECIFICATION* process. Therefore, the *IMPLEMENTATION* is also safe. With Equation 29, we test whether or not the *SPECIFICATION* process is refined by the *IMPLEMENTATION* process.

$$\textbf{assert } \textit{SPECIFICATION} \sqsubseteq_T \textit{IMPLEMENTATION} \tag{29}$$

where $\sqsubseteq_T$ is the trace refinement operator.

Next, we establish that the *SPECIFICATION* process exhibits a subset of all *IMPLEMENTATION* traces.

$$\textbf{assert } \textit{IMPLEMENTATION} \sqsubseteq_T \textit{SPECIFICATION} \tag{30}$$

Figure 3 shows that both refinement tests were successful. That means, the *IMPLEMENTATION* process is able to exhibit a subset of the *SPECIFICATION* traces and the *SPECIFICATION* process exhibits a subset of the *IMPLEMENTATION* traces. This implies that both processes exhibit exactly the same traces. This is a very important result, because it establishes that the *SPECIFICATION* is trace equivalent with the *IMPLEMENTATION*. The *SPECIFICATION* process models the desired functionality, therefore we have established that the implementation model, i.e. the *IMPLEMENTATION* process complies with the specification on the level of traces.

3. Discussion

This section discusses three issues raised in the main body of the paper. First, we examine the theoretical concept of failure refinement and its application to the TRB model. We highlight practical difficulties and point out possible solutions. The second issue under discussion is the practical application of the TRB concept. Even though input and output choice resolution appears to be such a natural concept, there are not many text book applications. One reason for this absence is that in the past such systems were difficult to implement and therefore text book authors focused on structures which did not include such constructs. For example, many text books discuss structures such as: farmer-worker and client-server. The final issue is the relevance of the TRB concept to hardware logic implementation.

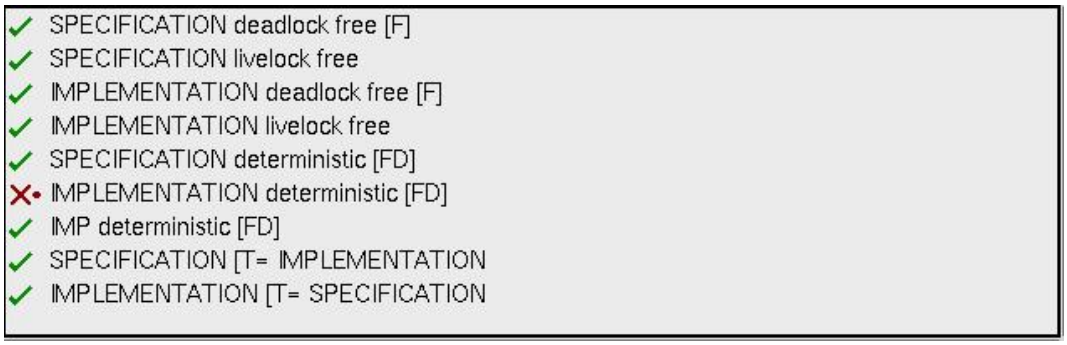

Figure 3. Output of the FDR model checker.

3.1. Failure Refinement

Failure (deadlock or/and livelock) refinement ensures that a particular process exhibits only a subset of the allowed traces and a subset of allowed stable failures associated with a particular trace. Section 1.1 defines the *SPECIFICATION* process and Section 1.4 defines the *IMPLEMENTATION* or implementation process. The *IMPLEMENTATION* process does not refine the *SPECIFICATION* in terms of stable failures. In this case, *IMPLEMENTATION* exhibits more stable failures then *SPECIFICATION*. The reason for this failure to refine is the sequential nature of the resolution operation in the *TRB* process. Sequential nature means that the *TRB* establishes a fixed sequence in which the individual sender and receiver can communicate. This sequence is enforced by stepping through Equations 16, 17 and 18. In contrast the *SPECIFICATION* process is free from such prioritisation and therefore it has fewer stable failures.

One way to overcome this difficulty is to introduce prioritisation into the *SPECIFICATION*. The idea is to show that a specification with prioritised external choice can be refined into an implementation which incorporates the TRB concept and therefore requires only prioritised alternation (PRI ALT). However, this paper aims to introduce the TRB concept. Therefore, we limited the scope of the automated checks to trace refinement. Failure refinement would have shifted the scope away from this introduction, because prioritised alternation requires extensive introduction.

3.2. Application

As mentioned before, there are not many textbook examples of systems which require input and output choice resolution. One of the few examples is an N lift system in a building with F floors. This example was presented by Forman [17] to justify the introduction of multiparty interactions in Raddle87. The lift problem concerns the logic to move lifts between floors. We limit the discussion to the following constraints:

1. Each lift has a set of buttons, one button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited (i.e. stopped at) by the lift.
2. Each floor has two buttons (except ground and top), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The buttons are cancelled when a lift visits the floor and is either travelling in the desired direction, or visiting the floor with no requests outstanding. In the latter case, if both floor request buttons are illuminated, only one should be cancelled. The algorithm used to decide which floor to service should minimise the waiting time for both requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests (or model a "holding" j floor).

4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority.
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.

From these constraints it is clear that there is a need for an external entity which manages the requests from the various buttons. The TRB concept can be used to solve this problem. All we have to do is to reinterpret the *IMPLEMENTATION* process network shown in Figure 2. The producer processes *P* model the buttons of the lift system and the consumer *C* processes model the lifts. Now, whenever a lift is free it registers with the TRB and whenever a button is pressed it illuminates and registers with the TRB. The TRB matches the requests from the buttons with the available lifts. After the choice is resolved, the consumer (lift) knows where to go and the producer (button) knows where to send the message to. The message is only exchanged when the lift stops at the desired floor. After the message is transferred, the button illumination is cancelled.

The TRB functionality, defined in Section 1.4 is a very crude solution to this particular problem, because it does not incorporate any algorithms to minimise the waiting time. However, the *IMPLEMENTATION*, shown in Figure 2, is proven to be deadlock and divergence free and transmitter processes can send messages to one of multiple receiver processes. Furthermore, it offers the same traces as a specification system which resolves input and output guards.

3.3. On Hardware Logic Implementation

In the introduction of his article about output guards and nondeterminism in CSP Bernstein mentions that an external entity which resolves input and output guards is impractical [4]. His argument was basically that this external entity would create a serious bottleneck, because of all the context switches necessary to schedule all processes involved in the message exchange. Now, this is only true for systems which have a scheduler. Say, all machine architectures with a multitasking operating system have such a scheduler. However, there is no scheduler in hardware process networks. In such systems all processes are executed in parallel, there are no context switches necessary. From this perspective the TRB concept does not introduce a performance penalty.

Another big advantage of hardware logic implementation is the trade-off relationship between execution time and silicon area. The *TRB* process, defined in Equations 15 to 19, has a complexity of roughly $O(TRB) = n \times m$ where n is the number of producer processes and m is the number of consumer processes. However, in hardware logic systems this does not mean that the processing time increases with $n \times m$. Depending on the implementation, the processing time might stay constant and the silicon area increases with $n \times m$. Therefore, the processing time bottleneck which is introduced by the TRB concept depends on the particular implementation.

A last point which highlights the practicality of the *IMPPLEMENTATION* model of the TRB concept is the efficiency with which binary matrices and vectors can be implemented in hardware logic. All hardware description languages provide direct support for such constructs. Due to this direct support there is no resource waste. Say, the 3 dimensional binary vector, which is transferred over the *p_start* channels, is represented by only 3+2 signals. The two additional signals are necessary for the channel functionality. This efficiency keeps the routing overhead manageable.

4. Conclusion

In this paper we present a solution to the long-standing problem of refining a model with input and output guards into a model which contains only input guards. The problem has practical relevance, because most implementation models incorporate only the resolution of input guards. We approach the solution in classic CSP style by defining a specification model which requires the resolution of input and output guards. This specification model provides the opportunity to test the functionality of the implementation model. The implementation model resolves only input guards. In other words the implementation model utilises only alternation. We achieved this by incorporating the transfer request broker process. The TRB concept is an elegant solution to the arbitration problem. The concept comes from fusing graph theory with CSP and introducing matrix manipulation functionality to CSP_M. The combination of these three points makes the TRB concept unique.

Automated model checking establishes that specification as well as implementation models are deadlock and divergence free. Furthermore, we prove that specification and implementation are trace equivalent. This is a strong statement in terms of safety, i.e. if the specification exhibits only safe traces then the implementation exhibits also only safe traces. However, trace equivalence is a weak statement about the functionality of the implementation, because it implies that the implementation is able to exhibit the same traces as the specification. The phrase 'is able to' indicates that the implementation can choose not to exhibit a particular event, even if it is within its traces. In other words, the implementation may have more stable failures then the specification. To resolve this problem may require prioritised external choice in the specification model, but these considerations would shift the focus away from the introduction of the TRB concept, therefore we leave this for further work.

The TRB concept allows us to extend the set of standard implementation models with systems which require the resolution of input and output guards. One of these examples is the *N* lift system. The TRB is used to mediate or broker between the requests from various buttons in the system and the individual lifts. Apart from introducing additional standard problems, the TRB concept eases also the refinement of specification into implementation models. Many specification models incorporate the resolution of output and input guards. With the TRB concept such models can be refined into implementation models which have to resolve only input guards without loss of functionality. These implementation models can be easily mapped into communicating sequential hardware processes executed by flexible logic devices. Under the assumption that there was no error in the mapping operation, the resulting implementation has the same properties as the implementation model.

The implementation model is particularly relevant for process networks in hardware logic. All hardware processes are executed in parallel, this limits the performance penalty incurred by introducing the additional TRB process. This is a big difference to software processes where scheduling problems might make the TRB impracticable. One last thought on system speed. The most expensive operation in the TRB process is the matrix checking. This is a logic operation therefore it can be done with a registered logic circuit which takes only one clock cycle to do the matrix checking. However, the area requirement for the logic circuit is of the order $O(n \times m)$ where n is the number of senders and m is the number of receivers. This makes the TRB concept very fast but not area efficient. But this constitutes no general problem for practical systems, because it is always possible to exchange processing speed and chip area. Therefore, the TRB concept benefits the design of many practical systems.

Acknowledgements

This work was supported by the European FP6 project "WARMER" (contract no.: FP6-034472).

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, first edition, 1985. http://www.usingcsp.com/cspbook.pdf Accessed Feb. 2007.

[2] G. N. Buckley and Abraham Silberschatz. An Effective Implementation for the Generalized Input-Output Construct of CSP. *ACM Trans. Program. Lang. Syst.*, 5(2):223–235, 1983.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[4] Arthur Bernstein. Output Guards and Nondeterminism in "Communicating Sequential Processes". *ACM Trans. Program. Lang. Syst.*, 2(2):234–238, 1980.

[5] David C. Wood and Peter H. Welch. The Kent Retargettable occam Compiler. In Brian C. O'Neill, editor, *Proceedings of WoTUG-19: Parallel Processing Developments*, pages 143–166, feb 1996.

[6] Jan F. Broenink, Andrè W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, mar 1999.

[7] P H Welch. Communicating Processes, Components and Scaleable Systems. Communicating Sequential Processes for Java (JCSP) web page, http://www.cs.kent.ac.uk/projects/ofa/jcsp/. Accessed Feb. 2007.

[8] Peter H. Welch and Jeremy M. R. Martin. A csp model for java multithreading. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 114, Washington, DC, USA, 2000. IEEE Computer Society.

[9] Rick D. Beton. libcsp - a Building mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 239–250, sep 2000.

[10] Neil C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, sep 2003.

[11] Geraint Jones. On guards. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 15–24, sep 1987.

[12] Peter H. Welch. A Fast Resolution of Choice between Multiway Synchronisations. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, page 389, sep 2006.

[13] Peter H. Welch, Frederick R. M. Barnes, and Fiona A. C. Polack. Communicating complex systems. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 107–120, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Peter H. Welch, Neil C. Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, jul 2007.

[15] Parrow and Sjödin. Multiway Synchronization Verified with Coupled Simulation. In *CONCUR: 3rd International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 1992.

[16] Formal Systems (Europe) Ltd., 26 Temple Street, Oxford OX4 1JS England. *Failures-Divergence Refinement: FDR Manual*, 1997.

[17] I. R. Forman. Design by decomposition of multiparty interactions in Raddle87. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 2–10, New York, NY, USA, 1989. ACM.