Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

A Calculated Implementation of a Control System

Alistair A. McEWAN
Computing Laboratory, University of Kent at Canterbury, UK

Abstract. In this paper, a case study consisting of a plant, and associated control laws, is presented. An abstract specification of the control system is given in Hoare's Communicating Sequential Processes (CSP). Via a series of calculated refinements, an implementation is developed, and translated into a simulation in a Java-based library for CSP, JCSP. Verification of the development process is performed using the model-checker for CSP, FDR. The result is a complete, verified implementation of the control system.

"**Control** (*n*): The apparatus by means of which a machine, as an aeroplane or motor vehicle, is controlled during operation; also, any of the mechanisms of a control apparatus, or collectively for the complete apparatus. " *Oxford English Dictionary, 2nd edition*

1 Introduction

In this paper, a case study consisting of a plant, and control laws, is investigated. An executable simulation of the control system is developed via a full, verified, top down procedure from an existing abstract specification. Proof obligations are discharged using FDR. The case study in question is a steam boiler—a device which accepts water input from a set of pumps, boils the water, and allows steam to escape through sets of valves. The boiler can be seen as analogous to, for instance, a cooling system in a nuclear reactor. An instantiation of the abstract control system is developed, and its correctness verified. This instantiation is then transformed into an executable refinement, and from this, a simulation is produced by translating this model into a Java-based library for CSP, JCSP.

Development of the new device is done in a top-down manner. The existing abstract specification of the device is refined in several stages into models where the implementation is exposed gradually. It is our belief that, by taking an abstract specification, and using a calculational approach to refining it into a concrete model, a high level of confidence can be achieved in the correctness of the implementation. The contribution of this paper, therefore, can be summarised as a demonstration of a full top-down development technique, from a high level specification to a dependable, verified, executable program.

The paper begins by presenting some relevant background material, including the case study in question. This is followed by a detailed description of the control laws in section 2. In section 3, a refinement of this model is produced, involving concrete instantiations of the processes necessary to implement the laws, and is further refined into an executable model in section 4. The executable model is translated into JCSP in section 5, resulting in a verified implementation of the abstract model. Some conclusions are drawn in section 6.

1.1 CSP and FDR

The process algebra Communicating Sequential Processes (CSP)[3, 11] is a mathematical approach to the study of concurrency and communication. It is suited to the specification, design, and implementation of systems that continuously act and interact with their environment. CSP is a state-based approach to modelling, where systems are characterised by the events in which they are willing to participate in their lifetime. The collection and interaction of these events form processes, which can be combined using the operators of CSP, to describe more complex systems. *Failures Divergences Refinement* (FDR) [4] is a tool for model-checking networks of CSP processes, checking the containment of processes, and allowing the proving or refuting of assertions about those processes.

1.2 Multi-way Synchronisations

The term *multi-way synchronisation* refers to the situation where three or more processes simultaneously engage in a common event. In CSP, this situation arises from synchronising processes in a network on a single event, and can be used, for instance, as a technique for composing different units of a specification [7], even thought the designer may never intend to implement the specification in this manner. Another use is for modelling common events in a system: for instance a system clock [13], or a quiescent, stable system state [12, 14]. Concurrent programming languages such as *Handel-C*[2] and occam[5] support one-to-one communication between processes: only two processes may read to, or write from, a channel. There is no direct support for implementing multi-way synchronisations. To produce executable code multi-way synchronisations must be replaced with constructs supported by these languages; and a family of protocols allowing this is presented in [6].

1.3 JCSP

JCSP[9, 10] is a Java class library providing a CSP-style interface to concurrent Java programming. A JCSP program consists of a network of communicating, independent processes, interacting with each other via synchronous channel communications. Although JCSP offers several constructs that do not have a direct, one-to-one correspondence with CSP primitives, the code developed in this case study utilises only those that do.

The JCSP model of concurrent programming corresponds to that of occam: processes interact via synchronous channel communication. The same communication constraints that are expected of a *Handel-C* or occam program exist: channel communications are one-to-one, and a process may not have an output on a channel as a possibility in a guard. Simple synchronisations do not exist—as in *Handel-C* and occam, these are implemented by passing arbitrary values across typed channels. Several extensions to classical occam exist: for instance, processes can be created and destroyed dynamically; and object-oriented concepts such as inheritance are commonly exploited in a JCSP program.

1.4 The Steam Boiler

The steam boiler case study is formally presented in [1], and is an example of a class of system where control in the presence of non-manifest failures is a fundamental issue. The boiler itself consists of a tank of water, a set of pumps that supply water to the tank, a valve allowing steam to escape, an emergency outlet, and sensors reporting the water level in the tank (table 1).

Pumps can be either open or closed—an open pump supplies water to the boiler. Four water levels are pre-defined: levels $N1$ and $N2$ depict the minimum and maximum normal, safe, operating levels respectively; while $M1$ and $M2$ represent minimum and maximum critical levels. If there is too much water in the boiler, the emergency overflow may be opened; and if pressure is too great, the steam outlet may be opened. Figure 1 shows a boiler

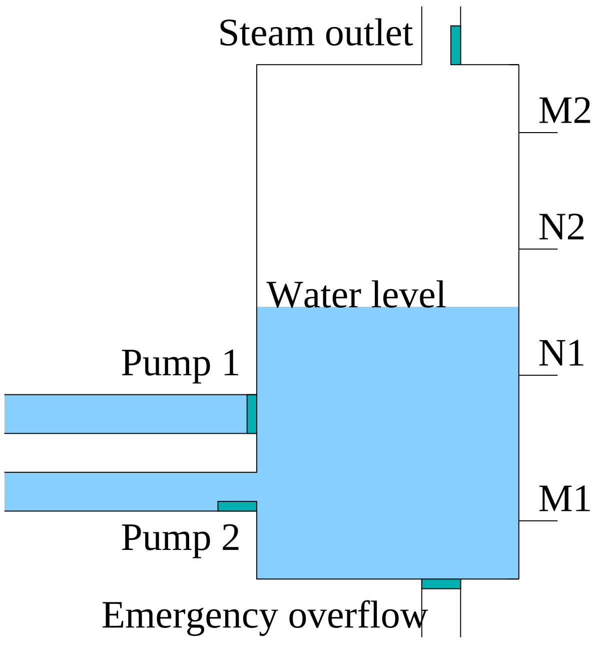

Figure 1: The steam boiler

Sense	Meaning
$LtM1$	Water level critically low
$LtN1$	Water level dropped below safe level
$BetweenN1N2$	Water level is between normal operating parameters
$GtN2$	Water level above safe level
$GtM2$	Water level critically high

Table 1: Senses in the system

state in which the water level lies between $N1$ and $N2$, the steam outlet is open, the overflow is closed, and one pump is open and the other closed.

Definition 1.1 *Basic control system requirements*

- *If the water level falls below N1, open closed pumps;*
- *If the water level rises above N2, close open pumps.*

It is the job of the control system to ensure that the quantity of water in the boiler remains at a safe level, summarised by definition 1.1. In the full model of the boiler, the control system may shut down the boiler if the water level becomes critical. Additionally, failures may occur non-deterministically in any of the components, and these failures must be mitigated. In this paper, only the normal operating mode is considered. Laws for initialisation, shut down, and failure mitigation exist, but as the derivation technique for their implementation is identical, they are not considered in this paper.

In [8], an *inference engine* is presented, which takes this set of rules, and instantiates a control system. The model of the inference engine uses an idiom that relies on concurrency and synchronisation, but not on data flow. For this reason, it is highly suitable to implementation in *Handel-C* on an FPGA, as concurrency can be exploited without the additional

$rule_0 \mathrel{\widehat{=}} Normal \wedge Level.LtN1 \wedge ClosePump.Pump1 \Rightarrow Disable.ClosePump.Pump1$	
$rule_1 \mathrel{\widehat{=}} Normal \wedge Level.LtN1 \wedge ClosePump.Pump2 \Rightarrow Disable.ClosePump.Pump2$	
$rule_2 \mathrel{\widehat{=}} Normal \wedge Level.GtN2 \wedge OpenPump.Pump1 \Rightarrow Disable.OpenPump.Pump1$	
$rule_3 \mathrel{\widehat{=}} Normal \wedge Level.GtN2 \wedge OpenPump.Pump2 \Rightarrow Disable.OpenPump.Pump2$	

Table 2: Assertion disablers

overhead of complex abstract data types being implemented directly into hardware. Each rule is declared as a CSP channel; and processes monitoring senses, inferring facts, and effecting actuates by synchronising on these channels are declared. These processes are given in the following sections.

2 The Control System

The control laws governing operation of the steam boiler are represented as a set of rules. In normal operating mode, rules can be divided into two distinct categories: those that react to senses, and those that infer facts about the plant—some of which are actuates to be performed on the plant. Rules are expressed as implications: the hypothesis of a rule is the conjunction of a set of facts which imply the conclusion—when all the facts in the hypothesis have been asserted, the conclusion can be inferred. All the rules used in this paper are taken from [8].

Definition 2.1 *The structure of a rule*

$$hypothesis \mathrel{\widehat{=}} fact_1 \wedge fact2 \wedge \ldots \wedge fact_n$$

$$rule \mathrel{\widehat{=}} hypothesis \Rightarrow conclusion$$

2.1 Assertion Disablers

An *assertion disabler* is a rule that prevents the control system from inferring facts that are incorrect, or from repeatedly asserting a known fact. The complete set of assertion disablers for normal, unfailed operating mode is given in table 2. In, for instance, $rule_0$, if the boiler is in *Normal* operating mode, and the level is below $N1$ and $Pump1$ is closed, then it is inferred that the ability to close $Pump1$ should be disabled.

In the case of a process monitoring a sense, when that sense is detected, the process attempts to synchronise on the channels corresponding to rules containing that sense in the hypothesis. This is shown in figure 2. This process monitors the occurrence of sense x; and when detected, it offers to synchronise on two different channels.

The CSP model of this is given in definition 2.2. The process *YetToSense* is initially willing to either engage in a *tock*—an event that delimits units of time in the boiler, after which it returns to its initial state. Alternatively, it may engage in a specific sense. If the sense occurs, subsequent behaviour is the process *Sensed*.

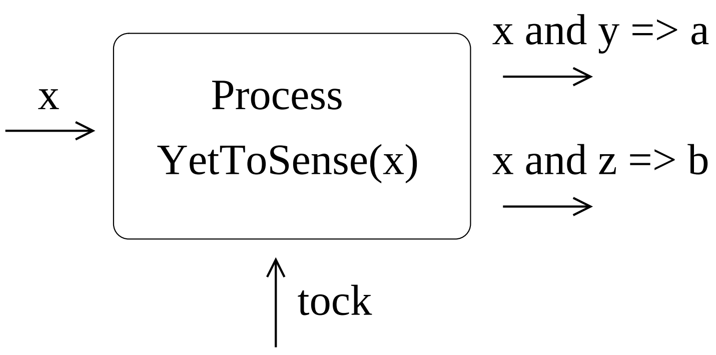

Figure 2: A sense process

Definition 2.2 *YetToSense and Sensed*

$$\begin{array}{l}
YetToSense(i) \mathrel{\widehat=} \\
\quad i \rightarrow Sensed(i) \\
\quad \Box \\
\quad tock \rightarrow YetToSense(i)
\end{array}$$

$$\begin{array}{l}
Sensed(i) \mathrel{\widehat=} \\
\quad let \\
\qquad use\_i\_to\_infer \mathrel{\widehat=} \{(A,b) \mid (A,b) \leftarrow deductions \cup ddeductions, i \in A)\} \\
\quad within \\
\qquad \Box (H,c) : use\_i\_to\_infer \bullet infer.(H,c) \rightarrow Sensed(i) \\
\qquad \Box \\
\qquad tock \rightarrow YetToSense(i)
\end{array}$$

The set *use_i_to_infer* in the process *Sensed* consists of all of the rules in the system where the sense event *i* appears in the hypothesis. Initially, this process is willing to synchronise on all of the channels corresponding to rules in this set. Alternatively, the time-slice may end with a *tock* event, and subsequent behaviour is the process *YetToSense*. The consequence of this is that the sense is no longer current and it is forgotten that it had been received—none of the rules requiring its assertion in the hypothesis are enabled.

2.2 Inference Assertions

Complementary to the assertion disablers are the rules for inferring assertions, given in table 3. In, for instance, $rule_4$, if it asserted that closing $Pump1$ has been disabled, and $Pump1$ is closed, then it can be inferred that $Pump1$ should be opened. In this way, it can be seen how these rules are complementary: $rule_0$ asserted, that when the water level is critically low, closing the pump should be disabled; and $rule_4$ asserted that if the pump were closed, and the ability to close the pump were disabled, then it should be opened—the fact that this is because of the water level dropping is left implicit.

The CSP model of this is given in definition 2.3. The set *i_inferred_from* corresponds to all the rules concluding the fact *i*, and initially, the process is willing to synchronise on any of the channels corresponding to these rules. Should one such synchronisation occur, then it must be the case that all processes attempting to assert a fact in the hypothesis of that rule have been successful. If its consequent is an actuation event, this can be performed, and subsequent behaviour is the process *Inferred*.

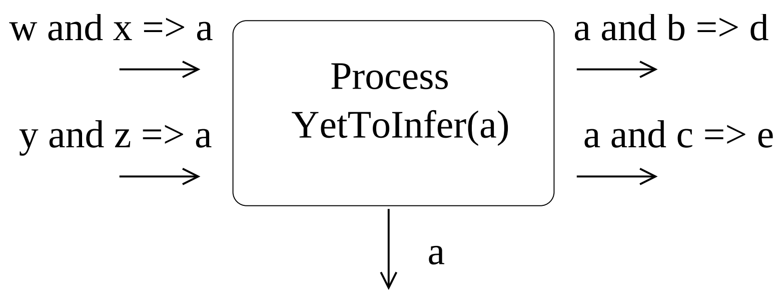

Figure 3: Inferring a fact from two possible hypothesis

$rule_4 \mathrel{\widehat=} Disable.ClosePump.Pump1 \wedge ClosePump.Pump1 \Rightarrow OpenPump.Pump1$
$rule_5 \mathrel{\widehat=} Disable.ClosePump.Pump2 \wedge ClosePump.Pump2 \Rightarrow OpenPump.Pump2$
$rule_6 \mathrel{\widehat=} Disable.OpenPump.Pump1 \wedge OpenPump.Pump1 \Rightarrow ClosePump.Pump1$
$rule_7 \mathrel{\widehat=} Disable.OpenPump.Pump2 \wedge OpenPump.Pump2 \Rightarrow ClosePump.Pump2$

Table 3: Performing actuates

Definition 2.3 *YetToInfer*

```
$YetToInfer(i) \mathrel{\widehat=}$
        $let$
                $i\_inferred\_from = \{(A,b) \mid (A,b) \leftarrow deductions \cup ddeductions, b = i\}$
        $within$
                $\Box(H,c) : i\_inferred\_from \bullet infer.(H,c) \rightarrow ($
                        $i \in Actuate \ \& \ c \rightarrow Inferred(i)$
                        $\Box$
                        $i \notin Actuate \ \& \ Inferred(i)$
                $)$

$Inferred(i) \mathrel{\widehat=}$
        $let$
                $Applicable \mathrel{\widehat=} \{(A,b) \mid (A,b) \leftarrow deductions, i \in A\}$
                $Forget \mathrel{\widehat=} \{(A,b) \mid (A,b) \leftarrow ddeductions, i \in A\}$
        $within$
                $\Box(H,c) : Applicable \bullet infer.(H,c) \rightarrow Inferred(i)$
                $\Box$
                $\Box(H,c) : Forget \bullet infer.(H,c) \rightarrow YetToInfer(i)$
```

In the process *Inferred*, the set *Applicable* comprises all of the assertion disablers where the event *i* appears in the hypothesis. Similarly, the set *Forget* consists of all the inference assertions where the event *i* appears in the hypothesis. Initially, *Inferred* is willing to synchronise on any channel in these sets. Should a synchronisation drawn from *Forget* occur, the process returns to a state where it needs to assert the hypothesis once more.

The complete control system is the instantiation of these processes, synchronising on the channels corresponding to rules. By constructing the system in this way there is no explicit data flow in the system: processes do not contain any data structures recording system state. Knowledge is implicitly held in the overall state of the system.

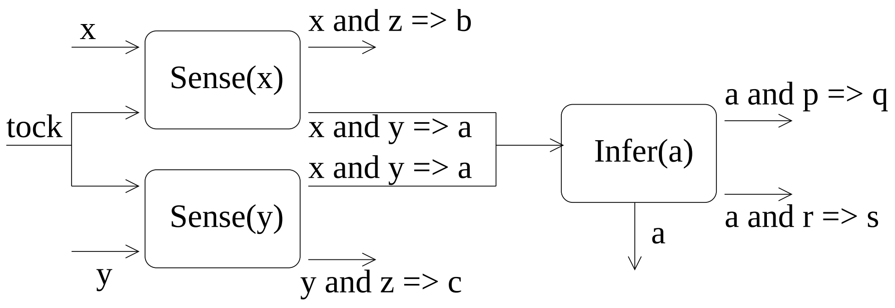

Figure 4: Connecting senses and inference

$$
\begin{array}{l}
rule_0 \mathrel{\widehat{=}} Level.LtN1 \land ClosePump.Pump1 \Rightarrow Disable.ClosePump.Pump1 \\
rule_1 \mathrel{\widehat{=}} Level.LtN1 \land ClosePump.Pump2 \Rightarrow Disable.ClosePump.Pump2 \\
rule_2 \mathrel{\widehat{=}} Level.GtN2 \land OpenPump.Pump1 \Rightarrow Disable.OpenPump.Pump1 \\
rule_3 \mathrel{\widehat{=}} Level.GtN2 \land OpenPump.Pump2 \Rightarrow Disable.OpenPump.Pump2
\end{array}
$$

Table 4: Simplified assertion disablers

3 Calculating Process Instantiations

In this section, the network of processes corresponding to the concrete instantiation of the inference engine is given. In doing so, several possible observations and optimisations are made, and operators used in the abstract description, but not available in the target programming environment, are removed.

3.1 Sense Dependencies

In the previous section, it was stated that only the rules for normal operating mode were being considered. Therefore, by propositional logic, a simpler set of assertion disablers may be used (table 4). Consequently, a process attempting to establish *Normal* is redundant, thereby reducing the costs of implementation.

Five instantiations of the process *YetToSense* are required, drawn from table 1. A static analysis of the rules reveals the set *use_i_to_infer* in each; the results are given in table 5—only *LtN*1 and *GtN*2 have consequences. Two parameterised versions of this process can be specified to reflect this, given in definition 3.2 and definition 3.1. When there are no consequents, the only possibility in the process *Sensed* is that the clock ticks after the sense has been received. In the case of two consequents, wither may fire. The complete network of senses are these instantiations, all synchronising on clock ticks.

Sense	*use_i_to_infer*
*LtM*1	$\emptyset$
*LtN*1	$rule_0$, $rule_1$
*BetweenN*1*N*2	$\emptyset$
*GtN*2	$rule_2$, $rule_3$
*GtM*2	$\emptyset$

Table 5: Sense dependencies

Actuate fact	*i_inferred_from*	*Applicable*	*Forget*
$OpenPump.Pump1$	$rule_4$	$rule_6$	$rule_2$
$OpenPump.Pump2$	$rule_5$	$rule_7$	$rule_3$
$ClosePump.Pump1$	$rule_6$	$rule_4$	$rule_0$
$ClosePump.Pump2$	$rule_7$	$rule_5$	$rule_1$
Non-actuate fact	***i_inferred_from***	***Applicable***	***Forget***
$Disable.OpenPump.Pump1$	$rule_2$	$\emptyset$	$rule_6$
$Disable.OpenPump.Pump2$	$rule_3$	$\emptyset$	$rule_7$
$Disable.ClosePump.Pump1$	$rule_0$	$\emptyset$	$rule_4$
$Disable.ClosePump.Pump2$	$rule_1$	$\emptyset$	$rule_5$

Table 6: Inference dependencies

Definition 3.1 *A sense with two consequents, A and B.*

$$\begin{aligned}&YetToSense(i,A,B) \mathrel{\widehat=}\\ &\quad i \rightarrow Sensed(i,A,B)\\ &\quad \Box\\ &\quad tock \rightarrow YetToSense(i,A,B)\end{aligned}$$

$$\begin{aligned}&Sensed(i,A,B) \mathrel{\widehat=}\\ &\quad A \rightarrow Sensed(i,A,B)\\ &\quad \Box\\ &\quad B \rightarrow Sensed(i,A,B)\\ &\quad \Box\\ &\quad tock \rightarrow YetToSense(i,A,B)\end{aligned}$$

Definition 3.2 *A sense with no consequents*

$$\begin{aligned}&Sense'(i) \mathrel{\widehat=}\\ &\quad i \rightarrow tock \rightarrow Sense'(i)\\ &\quad \Box\\ &\quad tock \rightarrow Sense'(i)\end{aligned}$$

3.2 Assertion Dependencies

A similar analysis can be performed for the eight facts involving assertion enablers (table 6). The first four facts correspond to actuates, the second four do not, meaning that two different versions of *YetToInfer* are required. Furthermore, a static analysis of the rules reveals that each of these facts appears precisely once in the hypothesis of the other rules—the set *i_inferred_from* can be calculated for each process; the results of this calculation are given in table 6 and the concrete, parameterised versions of the processes given in definition 3.3 and definition 3.4 respectively.

Definition 3.3 *An inference with an actuate*

$$\begin{aligned}&YetToInfer(i,A,B,C) \mathrel{\widehat=}\\ &\quad A \rightarrow i \rightarrow Inferred(i,A,B,C)\end{aligned}$$

$$\begin{aligned}&Inferred(i,A,B,C) \mathrel{\widehat=}\\ &\quad B \rightarrow Inferred(i,A,B,C)\\ &\quad \Box\\ &\quad C \rightarrow YetToInfer(i,A,B,C)\end{aligned}$$

Definition 3.4 *An inference without an actuate*

$$\begin{aligned}&\mathit{YetToInfer}'(A, B) \mathrel{\widehat=}\\&\quad A \rightarrow B \rightarrow \mathit{YetToInfer}'(A, B)\end{aligned}$$

Further analysis reveals that each actuate assertion has precisely one rule in *Applicable*, and one in *Forget*, while non-actuate assertions only have a single rule in *Forget* (table 6).This gives rise to two versions of the process *YetToInfer*. In definition 3.3, the process offers either of the consequent rules, while in definition 3.4 there is no choice, and laws of CSP permit this simpler representation.

Definition 3.5 *The network of actuate assertions*

$$\begin{aligned}&\mathit{ActuateAssertions} \mathrel{\widehat=}\\&\quad(\mathit{YetToInfer}(\mathit{OpenPump}.\mathit{Pump}1, \mathit{rule}4, \mathit{rule}2, \mathit{rule}6)\\&\quad\ [|\{|\mathit{rule}4, \mathit{rule}6|\}|]\\&\quad\ \mathit{YetToInfer}(\mathit{ClosePump}.\mathit{Pump}1, \mathit{rule}6, \mathit{rule}0, \mathit{rule}4))\\&\quad|||\\&\quad(\mathit{YetToInfer}(\mathit{OpenPump}.\mathit{Pump}2, \mathit{rule}5, \mathit{rule}3, \mathit{rule}7)\\&\quad\ [|\{|\mathit{rule}5, \mathit{rule}7|\}|]\\&\quad\ \mathit{YetToInfer}(\mathit{ClosePump}.\mathit{Pump}2, \mathit{rule}7, \mathit{rule}1, \mathit{rule}5))\end{aligned}$$

Definition 3.6 *The network of non-actuate assertions*

$$\begin{aligned}&\mathit{NonActuateAssertions} \mathrel{\widehat=}\\&\quad(\mathit{YetToInfer}'(\mathit{rule}0, \mathit{rule}4) \ |||\ \mathit{YetToInfer}'(\mathit{rule}1, \mathit{rule}5)\\&\quad\ |||\ \mathit{YetToInfer}'(\mathit{rule}2, \mathit{rule}6) \ |||\ \mathit{YetToInfer}'(\mathit{rule}3, \mathit{rule}7))\end{aligned}$$

The system of assertion dependencies is given by the parallel composition of the processes in each of definition 3.5 and definition 3.6, synchronising on the full set of rules. A full instantiation of the control system is this resulting process, in parallel with the sense dependencies, synchronising on the clock event and the set of rules. Rules are internal to the system, and are then hidden. The resulting system of processes can be verified equivalent to the original abstract model by model-checking using FDR. [1]

Theorem 3.1 *The instantiation is equivalent to the abstract model in section 2.*

Proof Model-check using FDR □

[1]This implementation assumes the boiler has initialised—establishing pumps are closed. Therefore this fact must be established manually: processes parameterised by $\mathit{ClosePump}.\mathit{Pump}1$ and $\mathit{ClosePump}.\mathit{Pump}2$ must be prefixed with their inferences firing: ($\mathit{rule}0 \rightarrow \mathit{YetToInfer}(\mathit{ClosePump}.\mathit{Pump}1, \ldots)$ and $\mathit{rule}1 \rightarrow \mathit{YetToInfer}(\mathit{ClosePump}.\mathit{Pump}2, \ldots)$ respectively).

4 Implementing Rules as Multi-way Synchronisations

As discussed in the previous section, each rule is a implemented as a channel. Further inspection of the complete system reveals that, for each rule, there are three processes whose agreement is required to enable any one given rule—this can be seen from table 5 and table 6. For instance, each of the processes $YetToSense(GtN2)$, $YetToInfer(OpenPump.Pump1, ...)$, and $YetToInfer'(Disable.OpenPump.Pump1, ...)$ all synchronise on $rule_2$. Furthermore, the clock ticking is a synchronisation between all of the sense processes and the boiler. To derive an executable implementation, these multi-way synchronisations must be removed. A straight application of the protocol presented in [6] is used to eliminate these multi-way synchronisations.

The elimination involves introducing a controller process, and set of channels to implement each multi-way synchronisation. As each controller is independent, they are interleaved. The protocol is applied to all of the processes developed in the previous section for each rule. The correctness of this development step is assured through the verification of the protocol in [6]; it is also possible, although unnecessary, to model check the resulting network using FDR. The CSP produced as a result of applying the protocol is rather long, and is included in appendix A. [2]

Theorem 4.1 *Removing the multi-way synchronisations has produced an equivalent system.*

Proof By the correctness of the multi-way synchronisation protocol. □

5 A JCSP Implementation

In this section, a description of the translation into an executable JCSP program is given. This simulation is only a demonstration of the derivation to executable code: while every step in the derivation to the executable model has been verified, the JCSP interface has not. No claims are made, therefore, about the behaviour of the JCSP library. JCSP was chosen to allow demonstrations of the results of this case study to be run on commodity personal computer environments.

Each channel in the CSP is declared as a JCSP channel. Many channels in the CSP specification utilise a process index and the value to be communicated. In the executable code, these indexes are statically determined by tagging the name of each process onto the channel name. For instance, example 5.1 gives the JCSP declaration of the channels corresponding to pumps in the original specification; all channels are declared in this manner.

Example 5.1 *Declaring the pump channels*

```
private static final int pumps = 2;
private static final int WIDTH = 2;
private One2OneChannelInt[][] open =
  new One2OneChannelInt[pumps][WIDTH];
private One2OneChannelInt[][] close =
  new One2OneChannelInt[pumps][WIDTH];
```

JCSP processes are classes that implement the library interface `CSProcess`. Each process instantiation in the complete CSP system is declared to implement this interface. Local private state, corresponding to the process parameters and the channels upon which that process

[2]For details of the protocol, the reader is referred to [6].

synchronises, are declared; and references to these passed to the object constructor. For instance, definition 5.1 contains the definition of a JCSP process implementing a non-actuate assertion disabler of definition 3.4. In this definition, local state corresponding to the channels used in the multi-way synchronisation (and the processes index in the synchronisation) are declared, and references to the global channels supplied on construction. From this example, it can be seen that the translation from a CSP process definition to a JCSP process definition is relatively straightforward, providing the specification has been refined to an executable subset.

Definition 5.1 *A process class for an inference without an actuate*

```
public class NonActuateAssertion implements CSProcess {

  ChannelOutputInt toA, toB;
  ChannelInputInt fromA, fromB;
  int a, b;
  Fact fact;

  public NonActuateAssertion (
    ChannelOutputInt toA, ChannelInputInt fromA, int a,
    ChannelOutputInt toB, ChannelInputInt fromB, int b,
    Fact fact
  ) {
    this.toA = toA;  this.fromA = fromA;  this.a = a;
    ...  etc
  }
}
```

Every process must implement the abstract method `run`, which is analogous to a main method in a program, and specifies what the process does when it is given a thread of control. In the run method for the non-actuate inference of definition 5.2, the process enters an infinite loop, engaging in the multi-way synchronisation corresponding to establishing its fact, and then the multi-way synchronisation corresponding to alerting other processes of the truth of this fact.

Definition 5.2 *The run method for an inference without an actuate*

```
public void run () {
  while (true) {

    toA.write (a);
    do {
      fromA.read ();
      toA.write (a);
    } while (fromA.read () != 1);

    toB.write (b);
    do {
      fromB.read ();
      toB.write (b);
    } while (fromB.read () != 1);

  }
}
```

A full JCSP program consists of a network of processes, grouped together in an array structure allowing for their parallel instantiation and execution, along with the relevant global declarations of common channels. Instruction as to how this is implemented is covered in the documentation for JCSP, and is omitted from this paper for brevity. However, a complete JCSP implementation of the steam boiler, the control system, and a graphical interface depicting the state of the system, has been developed and can be run on commodity personal computers.

6 Summary

In this paper, a full, top down derivation of an executable program was calculated from an abstract CSP specification. The result was a simulation of a control system, and associated plant, in a Java library for CSP that compiled, and ran without apparent error, and without need for an experiment/test cycle normally expected for a highly concurrent program.

Despite this notable success, there are several limitations. Firstly, the final stage of development, in moving from CSP to JCSP, is largely an approximation. There was no direct application of a refinement calculus to guide the translation from CSP to JCSP, so it cannot be completely justified. However, the library offering implementations of the primitives used has a clear one-to-one correspondence, so confidence can be earned from the simplicity of the process.

Secondly, although the JCSP library claims to implement these primitives, the majority of library itself has not been verified—therefore it cannot be guaranteed that the executable code behaves precisely as the specification intended. For instance, a non-terminating, mutually (infinitely) recursive pair of processes is a common appearance in CSP; but if translated directly into JCSP, stack overflow errors occur, leading to programs crashing. Such an error is not readily detectable by model-checking in FDR as it is a property of the implementation of the target executable language that does not exist in the mathematical model. Areas such as this need to be addressed if justified claims that the executable JCSP code was equivalent to the abstract CSP specification are to be made.

Despite these limitations, the production of the simulation is a definite success. The intention was to demonstrate that techniques exist allowing for the accurate, calculated production of executable concurrent code; and to produce a simple example of this which could be run on commodity personal computers for demonstration purposes—and this has been achieved. However, clearly, for the production of real control systems, applying the refinement calculus to a verified subset of, for instance, *Handel-C* on an FPGA, is necessary.

Acknowledgements

The author wishes to thank QinetiQ Malvern and Bedford for their role in funding this work, and Jim Woodcock and Peter Welch for extensive technical discussion and assistance.

References

[1] J. R. Abrial, E. Borger, and H. Laangmack, editors. *Formal methods for industrial applications: specifying and programming the steam boiler control*, volume 1165 of *LNCS*. Springer–Verlag, 1996.

[2] Celoxica. Handel-C reference manual. Technical report, Celoxica, 1999.

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

[4] Formal Systems (Europe) Ltd. FDR: User manual and tutorial, version 2.28. Technical report, Formal Systems (Europe) Ltd., 1999.

[5] INMOS Ltd. occam *Programming manual*. International Series In Computer Science. Prentice-Hall, 1984.

[6] Alistair A. McEwan. *Concurrent Program Development*. DPhil thesis, The University of Oxford, Submitted Trinity Term, 2004.

[7] Carroll Morgan and J. C. P Woodcock. What is a specification? In Dan Craigen and Karen Summerskill, editors, *Formal Methods for Trustworthy Computer Systems*, Workshops in Computing, pages 38–43. Springer-Verlag, 1989.

[8] Colin O'Halloran. Identifying critical requirements. Internal report of work in progress, Qinetiq, 2003.

[9] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[10] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.

[11] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[12] J. C. P. Woodcock. Montigel's Dwarf, a treatment of the Dwarf Signal problem using CSP/FDR. In *Proceedings of the 5th FMERail Workshop*, Toulouse, France, 1999.

[13] J. C. P Woodcock and Alistair A. McEwan. An overview of the verification of a *Handel-C* program. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume V, page 3003. CSREA Press, 2000.

[14] J. C. P. Woodcock and Alistair A. McEwan. Verifying the safety of a railway signalling device. In H. Ehrig, B. J. Kramer, and A. Ertas, editors, *Proceedings of IDPT 2002*, volume 1. The 6th Biennial World Conference on Integrated Design and Process Technology, Society for Design and Process Science, 2002. Winner of the best paper award.

A Control system processes implementations

Auxiliary definitions

Definition A.1 *Withdrawing from a synchronisation*

```
$Withdraw(to, from, i) \mathrel{\widehat=}$
        $(to!flip(i) \rightarrow SKIP)$
        $|||$
        $(from?invite \rightarrow invite \;\&\; from?any \rightarrow SKIP \;\Box\; \neg\; invite \;\&\; SKIP)$
```

Definition A.2 *A guaranteed synchronisation*

```
$GSync(to, from, i) \mathrel{\widehat=}$
        $to!i \rightarrow GSync'$

$GSync' \mathrel{\widehat=}$
        $from?any \rightarrow$
                $to!i \rightarrow from?sync \rightarrow$
                        $(sync \;\&\; SKIP \;\Box\; \neg\; sync \;\&\; GSync')$
```

Sense inferences

Definition A.3 *Senses not in a hypothesis, with multi-way synchronisations removed*

```
$Sense'(to, from, i, s) \mathrel{\widehat=}$
        $to!i \rightarrow Sense''(to, from, i, s)$

$Sense''(to, from, i, s) \mathrel{\widehat=}$
        $s \rightarrow Withdraw(to, from, i);$
                $GSync(to, from, i);$
                        $Sense'(to, from, i, s)$
        $\Box$
        $from?any \rightarrow$
                $to!i \rightarrow$
                        $from?sync \rightarrow$
                                $sync \;\&\; Sense'(to, from, i, s)$
                                $\Box$
                                $\neg\; sync \;\&\; Sense''(to, from, i, s)$
```

Definition A.4 *Senses in a hypothesis, with multi-way synchronisations removed*

```
$Sense(to, from, i, toA, fromA, a, toB, fromB, b, s) \mathrel{\widehat=}$
        $to!i \rightarrow Sense'(to, from, i, toA, fromA, a, toB, fromB, b, s)$

$Sense'(to, from, i, toA, fromA, a, toB, fromB, b, s) \mathrel{\widehat=}$
        $sense \rightarrow Withdraw(to, from, i);$
                $Sensed(to, from, i, toA, fromA, a, toB, fromB, b, s)$
        $\Box$
        $from?any \rightarrow$
                $to!i \rightarrow$
                        $from?sync \rightarrow$
                                $sync \;\&\; Sense(to, from, i, toA, fromA, a, toB, fromB, b, s)$
                                $\Box$
                                $\neg\; sync \;\&\; Sense'(to, from, i, toA, fromA, a, toB, fromB, b, s)$
```

```
$Sensed(to, from, i, toA, fromA, a, toB, fromB, b, s) \mathrel{\widehat=}$
    $(to!i \rightarrow SKIP \interleave toA!a \rightarrow SKIP \interleave toB!b \rightarrow SKIP);$
        $Sensed'(to, from, i, toA, fromA, a, toB, fromB, b, s)$

$Sensed'(to, from, i, toA, fromA, a, toB, fromB, b, s) \mathrel{\widehat=}$
    $from?any \rightarrow$
        $to!i \rightarrow$
            $from?sync \rightarrow$
                $sync \;\&\; (Withdraw(toA, fromA, a) \interleave Withdraw(toB, fromB, b));$
                        $YetToSense(to, from, i, toA, fromA, a, toB, fromB, b, s)$
                $\Box$
                $\neg\, sync \;\&\; Sensed'(to, from, i, toA, fromA, a, toB, fromB, b, s)$
    $\Box$
    $fromA?any \rightarrow$
        $toA!a \rightarrow$
            $fromA?sync \rightarrow$
                $sync \;\&\; (Withdraw(to, from, i) \interleave Withdraw(toB, fromB, b));$
                    $Sensed(to, from, i, toA, fromA, a, toB, fromB, b, s)$
                $\Box$
                $\neg\, sync \;\&\; Sensed'(to, from, i, toA, fromA, a, toB, fromB, b, s)$
    $\Box$
    $fromB?any \rightarrow$
        $toB!a \rightarrow$
            $fromB?sync \rightarrow$
                $sync \;\&\; (Withdraw(to, from, i) \interleave Withdraw(toA, fromA, a));$
                    $Sensed(to, from, i, toA, fromA, a, toB, fromB, b, s)$
                $\Box$
                $\neg\, sync \;\&\; Sensed'(to, from, i, toA, fromA, a, toB, fromB, b, s)$
```

Assertion inferences

Definition A.5 *An inference without an actuate, with multi-way synchronisations removed*

```
$YetToInfer'(toA, fromA, a, toB, fromB, b) \mathrel{\widehat=}$
    $GSync(toA, fromA, a);$
        $GSync(toB, fromB, b);$
            $YetToInfer'(toA, fromA, a, toB, fromB, b)$
```

Definition A.6 *An inference with an actuate, with multi-way synchronisations removed*

```
$YetToInfer(toA, fromA, a, toB, fromB, b, toC, fromC, c, i) \mathrel{\widehat=}$
    $GSync(toA, fromA, a);$
        $i \rightarrow Inferred(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$

$Inferred(toA, fromA, a, toB, fromB, b, toC, fromC, c, i) \mathrel{\widehat=}$
    $(toB!b \rightarrow SKIP \interleave toC!c \rightarrow SKIP);$
        $Inferred'(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$
```

```
$Inferred'(toA, fromA, a, toB, fromB, b, toC, fromC, c, i) \mathrel{\widehat=}$
    $fromB?any \rightarrow$
            $toB!a \rightarrow$
                $fromB?sync \rightarrow$
                    $sync \;\&\; Withdraw(toC, fromC, c);$
                        $Inferred(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$
                    $\Box$
                    $\neg\, sync \;\&$
                        $Inferred'(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$
    $\Box$
    $fromC?any \rightarrow$
        $toC!a \rightarrow$
            $fromC?sync \rightarrow$
                $sync \;\&\; Withdraw(toB, fromB, b);$
                    $YetToInfer(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$
                $\Box$
                $\neg\, sync \;\&\; Inferred'(toA, fromA, a, toB, fromB, b, toC, fromC, c, i)$
```