A Verified Implementation of a Control System

Alistair A. McEwan\textsuperscript{1} and J. C. P. Woodcock\textsuperscript{2}

\textsuperscript{1} Department of Computing, University of Surrey, Guildford, U. K.
\textsuperscript{2} Department of Computer Science, University of York, York, U. K.

Abstract. In this paper, a case study consisting of a plant, and associated control laws, is presented. An abstract specification of a control system governing operation of the plant is given in Hoare’s Communicating Sequential Processes (CSP). The control system is known to respect the safety properties required of the plant. Via a series of calculated, verified refinements, an implementation is developed on a Field Programmable Gate Array using Handel-C. Verification is performed using the model-checker for CSP, FDR. The result is a verified system on a chip.

"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 implementation of the control system is calculated via a full, verified, top down procedure, from an existing abstract specification. The implementation of the control system is used to control a simulation of the plant. Proof obligations in the development process are discharged using a model-checker.

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. Operation of the valves and the pumps is the responsibility of a control system—the boiler should never be allowed to run dry, or to flood. The combination of the boiler and the control system gives rise to an autonomous system, and as such, assurances about the correctness of the implementation are required.

An instantiation of an abstract control system is developed. The correctness of the implementation is assured through the development process, and can be verified using a model-checker. This instantiation is then transformed into an executable refinement—where only those operators available in the target programming language are used—and from this, a verified system on a chip is
produced using the Handel-C programming language for Field Programmable Gate Arrays.

Development of the implementation is done in a top-down manner. The existing abstract specification 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. Furthermore, as verification is performed at the specification stage, expensive development phases testing and correcting implementations are redundant. This verification phase can be automated using a model-checker. 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 verified system on a chip, where confidence in the behaviour and correctness of the implementation is assured.

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 3. In section 4, 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 5. The executable model is translated into Handel-C in section 6, resulting in a verified implementation of the abstract model. Some conclusions are drawn in section 7.

CSP and FDR

The process algebra Communicating Sequential Processes (CSP)[3, 13] 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. CSP is a process algebra: descriptions of processes can be re-written and manipulated in accordance with sets of algebraic laws; the correctness of these laws being justified within the semantic models associated with CSP.

Example 1. Process refinement in CSP: $A \sqsubseteq_{FD} B$

$$A \sqsubseteq a \rightarrow A \sqcap b \rightarrow A$$

$$B \sqsubseteq b \rightarrow B$$

Integral to CSP is the notion of refinement. A process $B$ is said to refine a process $A$ if all the observations possible of $B$ are also possible of $A$. This is shown in example 1. The process $A$ offers to engage in either the event $a$, or the event $b$, after which it recurses. Which of the two it chooses to offer is non-deterministic.
For instance, one possibility is that it will always choose to refuse the event $a$. This is precisely the behaviour described by the process $B$, and therefore $B$ can be regarded as a refinement of $A$, written $A \sqsubseteq_{FD} B$. The process $A$ is therefore an abstract specification of the process $B$: a customer who requested $A$ would be content if $B$ were delivered, and be none the wiser.

An important property of refinement is that if a behaviour is not possible of $A$, then it is also not possible of $B$. For instance, if the abstract specification can be shown to be deadlock-free—perhaps by model-checking—then the implementation is known to be also.

Failures Divergences Refinement (FDR) [4] is a tool for model-checking networks of CSP processes, checking refinements between processes, and allowing the proving or refuting of assertions about those processes. If assertions are incorrect, FDR presents a counter-example.

**Handel-C**

Handel-C[2] is a programming language, targeting Field Programmable Gate Arrays, reminiscent of occam[5]: co-routines, parallelism, and communication are all primitive to the language; the communication discipline is value passing with synchronous handshaking. Handel-C differs from occam in that parallel assignments to state variables all take place in synchrony on the leading edge of each clock cycle.

The clean, intuitive semantics of Handel-C mean that its programs may be modelled using CSP, augmented with a technique for describing abstract data types, such as the Z notation[22, 15]. Circus[7, 18, 14, 17, 21] is one such language combining the two notations—and currently, much interest is being shown in Circus as a language suitable for this style of hardware development. In this case study, the use of abstract data types is minimal, allowing us to regard the CSP model as equivalent to the Circus one.

The same communication constraints expected of an 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—these are implemented by passing arbitrary values across typed channels. Although Handel-C 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. Several extensions to classical occam exist: typically, these are specific to FPGAs, and are not considered in this development technique.

**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

---

3 The subscript $_{FD}$ refers to a particular refinement ordering. These are discussed in [13].
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 [10], even though 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 [19], or a quiescent, stable system state [16, 20]. 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 [7].

Example 2. A multi-way synchronisation

A ≜ a → A ⊔ x → A
B ≜ b → B ⊔ x → A
C ≜ c → C ⊔ x → A

System ≜ A || {x} || B || {x} || C

Example 2 is an example of one such synchronisation. Each of the processes A, B, and C, can engage independently in the events a, b, or c respectively. Alternatively, they are willing to engage in the event x—and for x to occur, all three processes must agree. The possibilities are that all three will engage in the event, or that one, or more, may instead engage in the alternative. The protocols used to replace this multi-way synchronisation are required to ensure that participants can negotiate whether or not all are willing, or if it should be refused.

The term output guard refers to the situation where an output communication is one of a possible set of choices—a more familiar description in CSP is an external choice where one of the possible choices is an output on a channel, as in example 3. Detection of output guards in CSP is a delicate affair: the output and input operators do not appear in normal form, therefore refinement checks will not reveal a problem. Concurrent programming languages such as Handel-C do not permit programs of this form. Techniques for removing output guards, by inspection and the application of verified laws, are given in [7].

Example 3 contains a simple output guard. The process A is willing to output the value 1 on the channel a, or to engage in the synchronisation b—precisely the situation not permitted. Inspection shows that the corresponding input in the process B is not guarded, therefore this guard can be removed simply by prefixing the communication in A with an input, and an output in B.
Example 3. An output guard, and a removed guard

\[
A \equiv a!1 \rightarrow SKIP \sqcap b \rightarrow SKIP \\
B \equiv a?any \rightarrow SKIP
\]

\[
A' \equiv (a?any \rightarrow a!1 \rightarrow SKIP) \sqcap b \rightarrow SKIP \\
B \equiv a!1 \rightarrow a?any \rightarrow SKIP
\]

In refining an abstract specification into a program that can be written in Handel-C, the designer must remove multi-way synchronisations, and output guards.

2 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).

Fig. 1. The steam boiler

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 state in which the water level lies between \( N_1 \) and \( N_2 \), the steam outlet is open, the overflow is closed, and one pump is open and the other closed.

**Definition 1. Basic control system requirements**

- If the water level falls below \( N_1 \), open closed pumps;
- If the water level rises above \( N_2 \), close open pumps.

<table>
<thead>
<tr>
<th>Sense</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( LtM_1 )</td>
<td>Water level critically low</td>
</tr>
<tr>
<td>( LtN_1 )</td>
<td>Water level dropped below safe level</td>
</tr>
<tr>
<td>Between( N_1 ) ( N_2 )</td>
<td>Water level is between normal operating parameters</td>
</tr>
<tr>
<td>( GtN_2 )</td>
<td>Water level above safe level</td>
</tr>
<tr>
<td>( GtM_2 )</td>
<td>Water level critically high</td>
</tr>
</tbody>
</table>

**Table 1. Senses in the system**

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. 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.\(^4\) 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.

**3 The Control System**

In [11], an inference engine is presented, which takes a 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 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.

\(^4\) The control laws have therefore been simplified accordingly. Each hypothesis contains the fact \( normal \), which is assumed to be true, and disappears through simple propositional logic.
The control system for the boiler is expressed in this manner. Rules are 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.

**Definition 2.** The structure of a rule

\[
\text{hypothesis} \equiv \text{fact}_1 \land \text{fact}_2 \land \ldots \land \text{fact}_n
\]

\[
\text{rule} \equiv \text{hypothesis} \Rightarrow \text{conclusion}
\]

Senses are emitted by the boiler, and the control system must react to those senses. The control system has one process for each sense emitted by the boiler. When a sense is detected, the relevant process attempts to synchronise on the channels corresponding to rules containing that sense in the hypothesis. This is as if the process were asserting the truth of that fact in the hypothesis of all those rules.

The CSP model of this is given in definition 3. The process YetToSense is initially willing to either engage in a **tack**—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.

**Definition 3.** Abstract YetToSense and Sensed

\[
\text{YetToSense}(i) \equiv
\]

\[
\begin{align*}
 i & \rightarrow \text{Sensed}(i) \\
 \Box & \rightarrow \text{YetToSense}(i) \\
 \text{tack} & \rightarrow \text{YetToSense}(i)
\end{align*}
\]

\[
\text{Sensed}(i) \equiv
\]

\[
\begin{align*}
 \Box (H, c) : X \bullet \text{infer}(H, c) & \rightarrow \text{Sensed}(i) \\
 \Box & \rightarrow \text{YetToSense}(i) \\
 \text{tack} & \rightarrow \text{YetToSense}(i)
\end{align*}
\]

The set \( X \) in the process Sensed consists of all of the rules in the system where the sense \( i \) 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 **tack** 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.

Table 2 contains rules in the control system intended to prevent it from repeatedly asserting known facts about the state of the system. In, for instance, rule\(_0\), if the level is below \( N1 \) and Pump1 is closed, then it is inferred that the ability to close Pump1 should be disabled, as the boiler is in a state where closing a pump would have an undesirable effect.

Complementary to these are the rules for inferring actuations, given in table 3. In, for instance, rule\(_4\), if it asserted that closing Pump1 has been disabled,
Table 2. Disable inappropriate assertions

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: rule0 asserted, that when the water level is critically low, closing the pump should be disabled; and rule4 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.

Table 3. Performing actuates

The CSP model of this is given in definition 4. The set \( X \) 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 \( c \), this can be performed, and subsequent behaviour is the process Inferred.

**Definition 4. Abstract YetToInfer**

\[
\text{YetToInfer}(i) \triangleq \\
\Box (H, c) : X \bullet \text{infer.}(H, c) \rightarrow (i \in \text{Actuate} \& c \rightarrow \text{Inferred}(i) \\
\quad \Box \\
\quad i \notin \text{Actuate} \& \text{Inferred}(i))
\]

\[
\text{Inferred}(i) \triangleq \\
\Box (H, c) : \text{Applicable} \bullet \text{infer.}(H, c) \rightarrow \text{Inferred}(i) \\
\quad \Box \\
\quad \Box (H, c) : \text{Forget} \bullet \text{infer.}(H, c) \rightarrow \text{YetToInfer}(i)
\]

Once the fact \( i \) has been inferred, the process attempts to synchronise on all of the rules containing \( i \) in the hypothesis. Two possibilities exist: these rules may
require the process to then forget $i$ had been received, or they may not. If a rule
fires causing the process to forget $i$, it returns to a state where it is required to
assert it again.

The complete control system is the instantiation of these processes, synchron-
ising 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 based on the synchronisations which have occurred.

4 Calculating Process Instantiations

In this section, the network of processes corresponding to the concrete instanti-
ation of the inference engine is given. In doing so, several possible observations
and optimisations are made.

Five senses exist in the control system—therefore five instantiations of the
process $YetToSense$ are required, drawn from table 1. A static analysis of the
rules, given in table 4, reveals only $LtN1$ and $GtN2$ have consequential rules. Two
parameterised versions of this process can be specified to reflect this, given in
definition 6 and definition 5. 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, either may fire. The complete network of senses
are these instantiations, all synchronising on clock ticks.

<table>
<thead>
<tr>
<th>Sense</th>
<th>Consequent rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LtM1$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$LtN1$</td>
<td>$rule_0, rule_1$</td>
</tr>
<tr>
<td>$BetweenN1N2$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$GtN2$</td>
<td>$rule_2, rule_3$</td>
</tr>
<tr>
<td>$GtM2$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Table 4. Sense dependencies

Definition 5. A sense with two consequents, $A$ and $B$.

\[
YetToSense(i, A, B) \equiv \\
i \rightarrow Sensed(i, A, B) \\
\square \\
tock \rightarrow YetToSense(i, A, B)
\]

\[
Sensed(i, A, B) \equiv \\
A \rightarrow Sensed(i, A, B) \\
\square \\
B \rightarrow Sensed(i, A, B) \\
\square \\
tock \rightarrow YetToSense(i, A, B)
\]
**Definition 6.** A sense with no consequents

\[
\text{Sense}'(i) \triangleq \\
i \rightarrow \text{tock} \rightarrow \text{Sense}'(i) \\
\square \\
\text{tock} \rightarrow \text{Sense}'(i)
\]

A similar analysis can be performed for the eight conclusions, given in table 5. The first four facts correspond to actuates, the second four do not, meaning that two different versions of \textit{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 of consequential rules can be calculated for each process; the results of this calculation are given in table 5 and the concrete, parameterised versions of the processes given in definition 7 and definition 8 respectively.

<table>
<thead>
<tr>
<th>Actuation</th>
<th>Concluded by</th>
<th>Applicable</th>
<th>Forget</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{OpenPump}.\textit{Pump1}</td>
<td>rule\textsubscript{4}</td>
<td>rule\textsubscript{6}</td>
<td>rule\textsubscript{2}</td>
</tr>
<tr>
<td>\textit{OpenPump}.\textit{Pump2}</td>
<td>rule\textsubscript{5}</td>
<td>rule\textsubscript{7}</td>
<td>rule\textsubscript{3}</td>
</tr>
<tr>
<td>\textit{ClosePump}.\textit{Pump1}</td>
<td>rule\textsubscript{6}</td>
<td>rule\textsubscript{4}</td>
<td>rule\textsubscript{0}</td>
</tr>
<tr>
<td>\textit{ClosePump}.\textit{Pump2}</td>
<td>rule\textsubscript{7}</td>
<td>rule\textsubscript{5}</td>
<td>rule\textsubscript{1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-actuate fact</th>
<th>Concluded by</th>
<th>Applicable</th>
<th>Forget</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Disable}.\textit{OpenPump}.\textit{Pump1}</td>
<td>rule\textsubscript{2}</td>
<td>\varnothing</td>
<td>rule\textsubscript{6}</td>
</tr>
<tr>
<td>\textit{Disable}.\textit{OpenPump}.\textit{Pump2}</td>
<td>rule\textsubscript{3}</td>
<td>\varnothing</td>
<td>rule\textsubscript{7}</td>
</tr>
<tr>
<td>\textit{Disable}.\textit{ClosePump}.\textit{Pump1}</td>
<td>rule\textsubscript{0}</td>
<td>\varnothing</td>
<td>rule\textsubscript{4}</td>
</tr>
<tr>
<td>\textit{Disable}.\textit{ClosePump}.\textit{Pump2}</td>
<td>rule\textsubscript{1}</td>
<td>\varnothing</td>
<td>rule\textsubscript{5}</td>
</tr>
</tbody>
</table>

Table 5. Inference dependencies

It can be observed that each actuate assertion has precisely one rule in \textit{Applicable}, and one in \textit{Forget}, while non-actuate assertions only have a single rule in \textit{Forget}. This gives rise to two versions of the process \textit{YetToInfer}. In definition 7, the process offers either of the consequent rules, while in definition 8 there is no choice.

**Definition 7.** An inference with an actuate

\[
\textit{YetToInfer}(i, A, B, C) \triangleq \\
A \rightarrow i \rightarrow \text{Inferred}(i, A, B, C) \\
\square \\
C \rightarrow \text{YetToInfer}(i, A, B, C)
\]

**Definition 8.** An inference without an actuate

\[
\textit{YetToInfer}'(A, B) \triangleq \\
A \rightarrow B \rightarrow \text{YetToInfer}'(A, B)
\]
The full control system is assertion inferences, 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; although the calculational nature of building this concrete version of the system means that such a check should be unnecessary.

**Theorem 1.** The instantiation is equivalent to the abstract model in section 3.

*Proof.* Model-check using FDR, or by the calculation nature of the development.

## 5 Removing Specification Constructs

We observe that three processes synchronise on the channel corresponding to any one given rule. For instance, $\text{YetToSense}(\text{GtN2})$, $\text{YetToInfer}(\text{OpenPump}, \text{Pump1})$, and $\text{YetToInfer'}(\text{Disable}, \text{OpenPump}, \text{Pump1})$ all synchronise on rule 2. Furthermore, the clock tick 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 [7] is used to eliminate these multi-way synchronisations. The precise details of this protocol, and its verification, are beyond the scope of this paper—for these, the reader is referred to [7].

The elimination involves introducing a controller process corresponding to each rule, and set of new channels to implement each multi-way synchronisation. The correctness of this development step is assured through the verification of the protocol in [7]; the resulting CSP is typically too large to model-check.

**Theorem 2.** Removing the multi-way synchronisations has produced an equivalent system.

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

The remaining task is to remove output guards. The abstract model leaves the directions of communications unspecified: in CSP terms, this means that channel communications were of the form $\text{chan}.value$, and not $\text{chan}!value$ or $\text{chan}?value$. However, these directions must be specified: for instance, effecting an actuation is actually a communication from the control system to the boiler.

In some cases, the direction of communication may be intuitive and obvious, in others, it may be irrelevant. For instance, in Handel-C a communication that is intended solely to achieve synchronisation between two processes is often implemented as a channel passing an arbitrary value. Such a situation exists where the boiler emits senses. As these directions are exposed, inevitably, output guards arise. A family of laws is presented in [7], allowing them to be transformed into equivalent processes not containing guards. The details of these laws is outwith the scope of this paper: for these, the reader is referred to [7].

The CSP produced as a result of applying the protocol, and the laws, is rather long: an example of one process is included in appendix A.
6 A Handel-C implementation

In this section, a description of the translation into a Handel-C program suitable for compilation on an FPGA is given. The discussion in this section is brief: only the operators necessary to implement the control system are covered—concurrency, communication, and recursion.

Typically, CSP processes are tail recursive. Handel-C has no direct support for recursion; but does for iteration. The translation from a tail recursion to an iteration is well understood, and normally processes will be required to be translated into an iterative form for translation into Handel-C—this may be achieved by replicated sequential composition, or by introducing a pseudo-code combining the two syntaxes.

Example 4. Tail recursion in pseudo-code

\[ P \triangleq a \rightarrow P \]

\[ P= \text{while}(\text{true}) \{ \ a \rightarrow \text{SKIP} \} \]

Variables in CSP are usually introduced using parameterised recursions. Handel-C is stricter: variables must be declared before use, and the width of those variables, in bits, must be specified. In this case study, the use of user state is minimal, and therefore is not considered.

Channels and channel types

Processes in Handel-C, as in CSP, communicate via synchronous channels. These channels are declared before use, and have a type associated with them: this is the type of data that will be passed across the channel. In the abstract control system, channels are simple synchronisations, and do not have a type associated with the communication. However, such a declaration is not possible in Handel-C: instead, channels must pass arbitrary values. This is given in example 5; and this style of declaration can be used to implement all of the synchronisations in the control system.

Example 5. Typed, and untyped, channels

```
channel x : B
channel y

chan x 1; // A channel communicating a boolean
chan y 1; // A synchronisation
```
Processes

A process in Handel-C is a compound statement which is part of a larger par block. Intuitively, therefore, there is some correspondence between the parallel operators of CSP, and a Handel-C par—although there is a semantic gap. They differ in the treatment of alphabets, or channels upon which the processes communicate. In CSP, the alphabet of the operator indicates which channels processes should synchronise on; whereas in Handel-C channels are strictly one-to-one communications between processes that have been explicitly passed them.

Example 6. Combining processes

\[
\begin{align*}
(P_1 \parallel [x] \parallel P_2) \setminus \{x\} & \quad (P'_1 \parallel P'_2) \setminus \{x\} \\
\text{par} & \{ P1(x); P2(x) \}
\end{align*}
\]

In example 6 the CSP processes \(P_1\) and \(P_2\) are placed in parallel, and synchronise on the event \(x\). To ensure the communication is one to one, the channel is then hidden. However, \(P'_1\) and \(P'_2\) do not synchronise on this event. The former corresponds to the Handel-C processes \(P1\) and \(P2\), who have each been parameterised by a channel end.

In a specification, a definition of a process may be used multiple times by renaming its component channels. To achieve this effect in Handel-C, the process can be declared as a macro procedure, and parameterised by the renamed channels where it is used. Example 7 shows this for the process \(P1\).

Example 7. A macro procedure

\[
P1 \equiv ...
\]

\[
\text{macro proc } P1(channel) \{ // .. Process definition ... \}
\]

\[
\text{par} \{ P1(x); P1(y); \}
\]

Sequential composition and termination

Example 8. Atomic communications and simple prefix

\[
a \rightarrow b \rightarrow SKIP
\]

\[
\{ a!any; b!any; \}
\]

Processes in CSP can be sequentially composed using the ; operator: the same is true of Handel-C. Often, within a CSP process is a number of implicit sequential compositions as a result of the simple prefix operator. For instance, a process which engages in a communication \(a\), then a communication \(b\), and then terminates could be written either as \(a \rightarrow b \rightarrow SKIP\), or as \(a \rightarrow SKIP; b \rightarrow SKIP\).
The latter simply represents the sequential composition of a number of simple
prefixes: these prefixes correspond to atomic communications in *Handel-C*. The
compound statement terminates when the final atomic statement terminates.

Care must be taken with use of *SKIP*. Unit laws of CSP apply—for instance
*SKIP; P = P = P; SKIP*. However, a process which starts, does nothing, and
terminates, may do one of two things in *Handel-C*—it may occupy a time cycle,
or not. Example 9 highlights this. While all three programs exhibit the behaviour
of *P* and only of *P*, the different timing properties of these three programs show
that a *delay* cannot be regarded as an implementation of *SKIP*. A closer analogy
would be *P**== P;*.

**Example 9.** Three very different *Handel-C* programs

\{ P \} != \{ delay; P \} != \{ P; delay \}

An example implementation of a sense process, using these translations, is given
in appendix B.

7 Summary

In this paper, a full, top down derivation of an executable program was calculated
from an abstract CSP specification. The result was a hardware implementation
of a control system, and simulation of the plant it was intended to control, that
operated without apparent error, and without need for an experiment and test
cycle normally expected for a highly concurrent program, or hardware develop-
ment project. The process allowed design errors, faults, and flaws, to be detected,
investigated, and corrected, at the specification phase.

Despite this notable success, there are several limitations. Firstly, the final
stage of development, in moving from CSP to *Handel-C* reveals a semantic gap.
Although the semantics of concurrency and communication in *Handel-C* reflect
those of CSP, there was no direct application of a refinement calculus to justify
the existence of each program statement in the style of [9]. However, there is
a clear one-to-one correspondence between concurrency and communication in
CSP and in *Handel-C*, so confidence can be earned from the simplicity of the
process.

Secondly, this case study was based on concurrency and communication—
and not on user state. A case study where more user variables were involved
may require a more sophisticated technique for reasoning about that user state,
and translating it into *Handel-C* variables. A promising approach in this area is
*Circus*, and this is discussed in [7].

Thirdly, the implementation of the *Handel-C* compiler itself has not been
verified—this development process does not guarantee that the implementation
at the level of hardware exactly conforms to the semantic intention at the level
of *Handel-C*. A Grand Challenge being embarked upon in the U. K. at present
is to build a verifying compiler: to look at how the *Handel-C* compiler could offer
assurances at the level of the FPGA could form an interesting research topic in this area.

In this paper, the correctness of the control system with respect to the safety properties required of the boiler was not proven. Other work on this case study, in [7], shows how a safety specification can be constructed in CSP, and the model-checker FDR used to confirm, or refute, the fact that the control system is correct with respect to these safety properties. Other similar works in this area are [20]. A possible approach to calculating requirements for a control system such as this is discussed in [8]. A software version of this case study, implemented in a Java based library for CSP, JCSP[12] has been presented in [6].

Further work continuing in this area is to show that the laws being applied to remove guarded outputs are jointly complete—that is, to show any deterministic CSP specification, comprising of a restricted signature, can be transformed into one that does not contain output guards. This work may lead to a theory allowing a tool to be built, automating the process from abstract CSP specification to verified Handel-C style hardware implementation.

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

Acknowledgements: The authors wish to thank QinetiQ and AWE for their role in funding this work; and Celoxica for providing the DK Toolkit and RC2000 FPGA board, and for funding dissemination.

References


A  An Example CSP Implementation of a Sense Inference

Definition 9. Senses not in a hypothesis, with multi-way synchronisations removed

\[ \text{Sense}'(\text{to}, \text{from}, i, s) \triangleq \]
\[ \text{to!} i \rightarrow \text{Sense}''(\text{to}, \text{from}, i, s) \]

\[ \text{Sense}''(\text{to}, \text{from}, i, s) \triangleq \]
\[ s \rightarrow \text{Withdraw}(\text{to}, \text{from}, i); \]
\[ GS\text{ync}(\text{to}, \text{from}, i); \]
\[ \text{Sense}'(\text{to}, \text{from}, i, s) \]
\[ \square \]
\[ \text{from?any } \rightarrow \]
\[ \text{to!} i \rightarrow \]
\[ \text{from?sync } \rightarrow \]
\[ \text{sync } \& \text{ Sense}'(\text{to}, \text{from}, i, s) \]
\[ \square \]
\[ \neg \text{ sync } \& \text{ Sense}''(\text{to}, \text{from}, i, s) \]

Definition 10. Senses in a hypothesis, with multi-way synchronisations removed

\[ \text{Sense}(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \triangleq \]
\[ \text{to!} i \rightarrow \text{Sense}'(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \]

\[ \text{Sense}'(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \triangleq \]
\[ \text{sense } \rightarrow \text{Withdraw}(\text{to}, \text{from}, i); \]
\[ \text{Sensed}(\text{to}, \text{from}, i, \text{toA}, a, \text{toB}, \text{fromB}, b, s) \]
\[ \square \]
\[ \text{from?any } \rightarrow \]
\[ \text{to!} i \rightarrow \]
\[ \text{from?sync } \rightarrow \]
\[ \text{sync } \& \text{ Sense}(\text{to}, \text{from}, i, \text{toA}, a, \text{toB}, \text{fromB}, b, s) \]
\[ \square \]
\[ \neg \text{ sync } \& \text{ Sense}'(\text{to}, \text{from}, i, \text{toA}, a, \text{toB}, \text{fromB}, b, s) \]
\[ \text{Sensed}(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \subseteq \]
\[ (\text{toli} \to \text{SKIP} \ || \ \text{toA}a \to \text{SKIP} \ || \ \text{toB}b \to \text{SKIP}); \]
\[ \text{Sensed}'(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \]
\[ \text{Sensed}'(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \subseteq \]
\[ \text{from}? \any \to \]
\[ \text{to}!i \to \]
\[ \text{from}? \sync \to \]
\[ \sync \& (\text{Withdraw(\text{toA, fromA, a}) \ || \ Withdraw(\text{toB, fromB, b})}); \]
\[ \text{YetToSense(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s)} \]
\[ \neg \sync \& \text{Sensed}'(\text{to}, \text{from}, i, \text{toA}, \text{fromA}, a, \text{toB}, \text{fromB}, b, s) \]
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
\[ \)
B An Example *Handel-C* implementation of a Sense Inference

To follow!