Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 137 (2005) 5–22
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Simpler Reasoning About System Properties: a Proof-by-Refinement Technique

D. Atiya[1], S. King[2], and J. C. P. Woodcock[3]

Department of Computer Science, University of York, York, UK

Abstract

Proofs about system specifications are difficult to conduct, particularly for large specifications. Using abstraction and refinement, we propose a proof technique that simplifies these proofs. We apply the technique to *Circus* (a combination of Z and CSP) specifications of different complexities. Interestingly, all the proofs are conducted in Z, even those concerning reactive behaviour.

Keywords: Proofs, Refinement, Circus, Z

1 Introduction

Central to the formal development of systems is the construction of abstract specifications and the refinement of these specifications into more concrete designs. It is often the case that we need to show that the starting specification satisfies certain properties. These could be simple consistency checks (e.g. initialisation theorems) or other general properties (e.g. freedom of deadlock/divergence, in concurrent systems). Should these properties be preserved by the refinement steps, then proving them for the starting specification would mean they hold for the subsequent designs. However, these proofs are often difficult to conduct, particularly for large specifications.

[1] Email: diyaa@cs.york.ac.uk
[2] Email: king@cs.york.ac.uk
[3] Email: jim@cs.york.ac.uk

doi:10.1016/j.entcs.2005.04.022

This paper proposes a proof technique that facilitates reasoning about system specifications. The idea is simple: treat the specification as a "design" of a yet more abstract model in which the proofs are much easier to conduct. To prove that a specification S satisfies a property P:

(i) Propose a more abstract model, say A, of S

(ii) Show that P holds for A

(iii) Show that S is a refinement of A

Of course, the proofs justifying steps ii–iii should be much easier than the direct proof that P holds for S. Also, these steps implicitly assumes that the property P is preserved by the refinement relation in step iii.

In this paper, we apply the above proof-by-refinement technique to *Circus* [12] specifications. Although some of the properties verified are about concurrency, e.g. freedom from deadlock, all the proofs are conducted in Z. This is an interesting result in its own right, as Z and its tool support are traditionally limited to sequential programs. The rest of this paper is organised as follows. Section 2 provides a brief introduction to the *Circus* specification language. Section 3 describes the proof-by-refinement technique and applies it to two examples of different complexities. Finally, Section 4 discusses our experience with the proposed technique and draws the concluding remarks.

2 Circus

Circus is a unified programming language that combines Z and CSP constructs, together with specification statements [9] and guarded commands [6]. With Z and CSP integrated into the language, *Circus* can be used to describe both the state-oriented and the behavioural aspects of concurrent systems. Though there are several other examples of combining Z and CSP in the literature (see, for example, the survey in [7]), *Circus* distinguishes itself by a theory of refinement [4,5,11] for the derivation of programs from their specifications in a calculational style like Morgan's [10]. *Circus* has a well-defined syntax and a formal semantics [12], as well as various refinement laws for refinement [5] of specifications into designs/programs.

A *Circus* program is a sequence of:

- ***Z paragraphs:*** declaring the types, global constants, and other data structures used by the processes defined in the program.
- ***Channel definitions:*** declaring typed channels through which processes can communicate or synchronise.
- ***Process definitions:*** declaring encapsulated state and reactive behaviour.

In its simplest form, a process definition is a sequence of Z paragraphs describing the internal state, and a sequence of operations that describe possible changes in the state and/or interactions between the process and its environment. These operations are called *actions*, and are defined in terms of Z schemas, CSP operators, and guarded commands). In more sophisticated forms, a process may be defined in terms of combinations of other processes using the operators of CSP.

2.1 Example: A Protected Natural Number

We use *Circus* to model a simple process (*ProtNat*) that encapsulates, and provides mutual exclusive access to, a natural number. The process is depicted in Figure 1 and can be described as follows:

Global Definitions

Other processes, drawn from the given set *ProcID*, interact with *ProtNat* through three channels: *write*, *read*, and *leave*. A communication on the channel *write* (*read*) represents the event where a process updates (reads) the data encapsulated in *ProtNat*. A communication on the channel *leave* represents the event when a process has finished its current writing (reading) operation to (from) *ProtNat*.

Process State

There are three components in the state of *ProtNat*.

- The data encapsulated, *data*, of type $\mathbb{N}$.
- The set of *readers*, those processes currently reading from *ProtNat*.
- The set of *writers*, those processes currently writing to *ProtNat*.

As represented by the *ProtNatState* schema, there are two state invariants.

- Reading and writing are mutually exclusive.
- There must be no more than one writer.

Process Actions

The constituent actions of the process are

(i) InitProtNatState: Initially there are no readers or writers, the encapsulated *data* is given an initial value *d*?.

(ii) Update: This action is enabled only if there no current readers or writers. When enabled, the action is signalled by the input communication of a

$[ProcID]$

channel $read, write : ProcID \times \mathbb{N}$
channel $leave : ProcID$

process $ProtNat \mathrel{\widehat=}$ **begin**

ProtNatState

$$data : \mathbb{N}$$
$$readers, writers : \mathbb{F}\, ProcID$$
$$readers \neq \emptyset \Rightarrow writers = \emptyset$$
$$\#writers \leq 1$$

$$InitProtNatState \mathrel{\widehat=} [\, ProtNatState'; \; d? : \mathbb{N} \mid data' = d? \land readers' = writers' = \emptyset \,]$$

$$Update \mathrel{\widehat=} writers \cup readers = \emptyset \;\&\; write?id?d :\rightarrow writers := \{id\}; data := d$$

$$Retrieve \mathrel{\widehat=} writers = \emptyset \;\&\; read?id!data \rightarrow readers = readers \cup \{id\}$$

$$WriterLeave \mathrel{\widehat=} leave?id : writers \rightarrow writers := \emptyset$$

$$ReaderLeave \mathrel{\widehat=} leave?id : readers \rightarrow readers := readers \setminus \{id\}$$

$$ReactiveBehaviour \mathrel{\widehat=} \mu\, X \bullet (\, Update \;\Box\; Retrieve \;\Box\; WriterLeave \;\Box\; ReaderLeave \,);\; X$$

- $InitProtNatState;\; ReactiveBehaviour$

end

Fig. 1. The *Circus* *ProtNat* process

process identifier and the new value to be written [4]. The process becomes the sole writer, and the *data* is updated accordingly.

(iii) Retrieve: This action is enabled only if there is no current writer. When enabled, the action is signalled by the input communication of a process identifier and output of the current *data* value. The process becomes a reader.

[4] For simplicity, we view the update operation as writing new values to *ProtNat*, rather than calculating these values based on current value of *data*.

(iv) WriterLeave: When a process has finished its current writing operation, it leaves *ProtNat*; this is signalled by the communication of the process identifier over the *leave* channel.

(v) ReaderLeave: When a process has finished its current reading operation, it leaves *ProtNat*; again, this is signalled by the communication of the process identifier over the *leave* channel.

(vi) ReactiveBehaviour: Repeatedly offers the external choice between *Update*, *Retrieve*, *WriterLeave* and *ReaderLeave*

The extensional behaviour of the process is given by its main action

$$InitProtNatState;\ ReactiveBehaviour$$

As a useful check on the consistency of the model, we finish this section by showing that an initial state exists for the *ProtNat* process.

Theorem 2.1 $\exists\, ProtNatState' \bullet InitProtNatState$

Proof. Each state component is fixed by an equality in *InitProtNatState*. The expressions in these equalities trivially satisfy *ProtNatState* invariants (by the one-point rule and properties of propositional calculus and set theory). □

3 Proofs By Refinement

To prove that a *Circus* specification S satisfies a property P:

(i) Define an abstraction A that has the same structure and state components of S, with the same types and invariants. This abstraction can be obtained by removing existing guards, changing external choices into internal choices, and/or changing schema operations to unconstrained schemas.

(ii) Show that A satisfies the property P.

(iii) Using the refinement laws of *Circus* show that $A \sqsubseteq S$.

The idea, as shown by the examples below, is that the abstract model defined in step i would have a very simple structure that the proof in step ii would be simple, if not straightforward or obvious. Also, proving the side conditions of *Circus* laws in step iii should be easier than proving the direct conjecture that S satisfies P.

$[ProcID]$

channel $read, write : ProcID \times \mathbb{N}$
channel $leave : ProcID$

process $AProtNat \mathrel{\widehat=}$ **begin**

$$
\begin{array}{|l}
\hline
AProtNatState \\
data : \mathbb{N} \\
readers, writers : \mathbb{F}\, ProcID \\
\hline
readers \neq \emptyset \Rightarrow writers = \emptyset \\
\#writers \leq 1 \\
\hline
\end{array}
$$

$InitAProtNatState \mathrel{\widehat=} [\, AProtNatState'; \; d? : \mathbb{N} \,]$

$AUpdate \mathrel{\widehat=} \prod id : ProcID; \; d : \mathbb{N} \bullet write.id.d :\rightarrow \Delta AProtNatState$

$ARetrieve \mathrel{\widehat=} \prod id : ProcID; \; d : \mathbb{N} \bullet read.id.d \rightarrow \Delta AProtNatState$

$AWriterLeave \mathrel{\widehat=} \prod id : ProcID \bullet leave.id \rightarrow \Delta AProtNatState$

$AReaderLeave \mathrel{\widehat=} \prod id : ProcID \bullet leave.id \rightarrow \Delta AProtNatState$

$ARetrieveBehaviour \mathrel{\widehat=}$
$\quad \mu\, X \bullet (\, AUpdate \sqcap ARetrieve \sqcap AWriterLeave \sqcap AReaderLeave \,); \; X$

$\bullet \; InitAProtNatState; \; AReactiveBehaviour$

end

Fig. 2. The *Circus* *AProtNat* process

3.1 Example 1: ProtNat is deadlock-free and divergence-free

Shown in Figure 2, the abstract model *AProtNat* has the same structure, state components, types and invariants as that of *ProtNat*. The abstract model also uses the same channels, and with similar actions to those defined in *ProtNat*. However, in *AProtNat* there are no guards, and state changes are unconstrained, provided the state invariant is maintained. Finally, the main action of the abstract model is the repeated nondeterministic choice between its actions, following the initialisation of the state.

We now prove that *AProtNat* is deadlock-free and divergence-free. This implies that the actions of *AProtNat* maintain the state invariants, otherwise

the process would not be divergence-free.

Theorem 3.1 *If ProcID is nonempty, then the abstraction AProtNat is both deadlock-free and divergence-free.*

Proof. By inspection of *Circus* syntax and semantics, there are eight conditions that are *sufficient* for a *Circus* process to be both deadlock and divergence-free:

(i) It is sequential.
(ii) It is free from hiding.
(iii) It doesn't mention *Stop* or *Chaos*.
(iv) All internal and external choices are over non-empty sets.
(v) Its channel types are non-empty.
(vi) Its local definitions are satisfiable.
(vii) Its main action's initial state exists.
(viii) Its actions are all total on the state.

Conditions (i)–(iii) are satisfied syntactically. Conditions (iv) and (v) are guaranteed by the proviso of the theorem. Condition (vi) is trivially satisfied, since there are no local definitions. Condition (vii) can be stated as

$$\forall\, d? : \mathbb{N} \bullet \exists\, AProtNatState' \bullet InitAProtNatState$$

Expanding the schemas, we must prove that

$$\begin{array}{l} \forall\, d? : \mathbb{N} \bullet \\ \quad \exists\, data' : \mathbb{N};\ readers', writers' : \mathbb{F}\, ProcID \bullet \\ \quad\quad (readers' \neq \emptyset \Rightarrow writers' = \emptyset) \land \#writers' \leq 1 \end{array}$$

which is true, since both $\mathbb{N}$ and *ProcID* are non-empty. Finally, condition (viii) follows trivially from the construction of the actions from the total, but arbitrary state change $\Delta AProtNatState$: all actions have true guards and never abort. □

Now, if we can prove that *ProtNat* is a refinement of *AProtNat*, then we can conclude that *ProtNat* is also deadlock-free and divergence-free. Moreover, the main action of *ProtNat* must also preserve the state invariants, otherwise *AProtNat* would not be divergence-free. We state and prove that *ProtNat* is a refinement of *AProtNat* in Theorem 3.2, which will make use of the following three laws [5].

[5] The proofs of these laws are direct consequences of the work reported in [2].

Law 1 is about the refinement of internal choices over a number of prefixed actions. Using this law, the internal choice can be transformed to an external choice over a number of guarded actions.

Law 1 *Suppose, for $i \in I$, that c_i is a channel, that S_i and T_i are subsets of the communicable values over c_i, that T_i is non-empty, that A_i and B_i are actions over a common state, that g_i is a boolean-valued expression over the state, and that pre is an assertion about the state. Then*

$$\{pre\}; \prod i : I \bullet (\prod x : T_i \bullet c_i.x \rightarrow A_i) \quad \sqsubseteq \quad \Box\, i : I \bullet g_i \;\&\; c_i?x : S_i \rightarrow B_i$$

provided

(i) $pre \Rightarrow \bigvee i : I \bullet g_i \wedge S_i \neq \emptyset$

(ii) $\forall i : I \bullet S_i \subseteq T_i$

(iii) $A_i \sqsubseteq B_i$, *for all* $i : I$

The assumption $\{pre\}$ is used to record the abstract action's precondition; it does nothing if pre is true, and aborts otherwise. □

Law 2 applies to guarded prefixed actions. Simply, the law states that if the action does engage in a communication with its environment, then the guard (g) and the communicated value (x) are in scope for that part of the action which follows the communication.

Law 2 *Suppose that A is an action, g is a guard over A's state, c is a channel, and S is a subset of c's communicable values.*

$$g \;\&\; c?x : S \rightarrow A \quad = \quad g \;\&\; c?x : S \rightarrow \{g \wedge x \in S\}\; A$$

□

Law 3 states the necessary conditions for the refinement of a schema operation into an assignment statement.

Law 3 *Suppose that Op is a schema operation over a state with variables x and w, that e is an expression with the same type as x, and that pre is an assertion over the variables in scope.*

$$Op \quad \sqsubseteq \quad \{pre\}\; x := e$$

provided $\quad pre \wedge \text{pre } Op \Rightarrow Op\,[\,x', w' := e, w\,]$

The notation $S\,[\,y := f\,]$ denotes the predicate S, with f systematically substituted for y. □

Using Laws 1–3, we can now show that *ProtNat* is indeed a refinement of *AProtNat*.

Theorem 3.2 $ProcID \neq \emptyset \Rightarrow AProtNat \sqsubseteq ProtNat$

Proof. From [5], and since *AProtNat* and *ProtNat* have the same state, this refinement holds provided that

(a) $InitAProtNatState \sqsubseteq InitProtNatState$

(b) $AReactiveBehaviour \sqsubseteq ReactiveBehaviour$

Proviso (a) follows from Theorem 2.1. We also know that $\sqsubseteq$ distributes through recursion. Thus, to prove Proviso (b), it is sufficient to show that

$$(AUpdate \sqcap .. \sqcap AReaderLeave) \sqsubseteq (Update \Box .. \Box ReaderLeave)$$

This, in turn, is a direct consequence of applying Law 1 to the nondeterministic choice over *AProtNat* actions.
Thus, all we have to do now is prove that provisos 1–3 (of Law 1) hold for *AProtNat* and *ProtNat* actions. To prove Proviso (i), and since $ProcID \neq \emptyset$, it sufficient to prove that
$pre \Rightarrow ((writers \cup readers = \emptyset) \vee (writers \neq \emptyset) \vee (writers = \emptyset) \vee (readers \neq \emptyset))$
which is trivially satisfied, since the consequent is always true. Also, Proviso (ii) is trivially satisfied, since the abstract sets are all types.
Using Law 2, Proviso (iii) follows from the following proof obligations:

(i) *Update*

$$[\Delta AProtNatState;\ id? : ProcID;\ d? : \mathbb{N}] \sqsubseteq \left\{ readers \cup writers = \emptyset \wedge id \in ProcID \wedge d \in \mathbb{N} \right\}; writers := \{id\};\ data := d$$

(ii) *Retrieve*

$$[\Delta AProtNatState;\ id? : ProcID] \sqsubseteq \left\{ writers = \emptyset \wedge id \in ProcID \right\};\ readers := readers \cup \{id\}$$

(iii) *WriterLeave*

$$[\Delta AProtNatState;\ id? : ProcID] \sqsubseteq \left\{ id \in writers \right\};\ writers := \emptyset$$

(iv) *ReaderLeave*

$$[\Delta AProtNatState;\ id? : ProcID] \sqsubseteq \left\{ id \in readersf \right\};\ readers := readers \setminus \{t\}$$

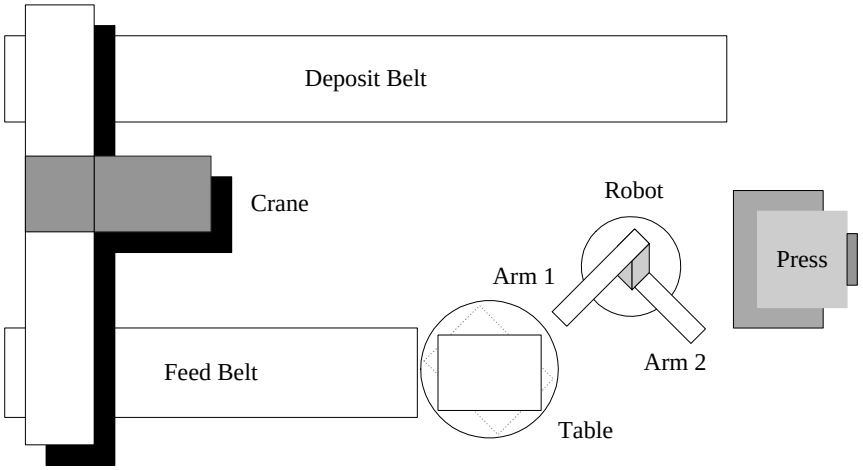

Fig. 3. A top view of the Production Cell

Proof obligation (*Update*) follows directly from Lemma A.1, which is given in Appendix A. The other obligations have similar proofs. □

Finally, we can now achieve our main goal for this example and show that the process *ProtNat* is free from deadlock and divergence.

Theorem 3.3 *If $ProcID \neq \emptyset$, then ProtNat is deadlock- and divergence-free.*

Proof. This is a direct consequence of Theorems 3.1–3.2 □

3.2 Example 2: Production Cell

The Production Cell is a realistic case study from industry, representing an actual installation used in a metal factory. The system is of moderate complexity and can modelled by a finite automaton with a number of states in the order of 10^{12} [8]. As illustrated in Figure 3, the Production Cell system comprises six machines, working concurrently: a *feed belt*, an *elevating table*, a *robot*, a *press*, a *deposit belt*, and a *crane*.

From the perspective of a metal plate, the general sequence of production is:

– The feed belt transports a metal plate to the table.
– The table elevates and rotates so that the robot can pick up the plate.
– The robot picks up the plate with its first arm, then turns anticlockwise and feeds the metal plate into the press.
– The press forges the plate and returns to bottom position in order to unload.

$BeltStatus ::= no\_plate \mid plate\_before\_end \mid plate\_at\_end$

channel $start, drop$

process $FeedBelt \mathrel{\widehat=}$ **begin**

$FeedBeltState \mathrel{\widehat=} [\, fb : BeltStatus \,]$

$InitFeedBeltState \mathrel{\widehat=} [\, FeedBeltState' \mid fb' = no\_plate \,]$

$LoadFeedBelt \mathrel{\widehat=} fb = no\_plate \;\&\; start \rightarrow fb : [\, true, fb' = plate\_before\_end \,]$

$TransferPlate \mathrel{\widehat=} fb = plate\_before\_end \;\&\; fb : [\, true, fb' = plate\_at\_end \,]$

$UnloadFeedBelt \mathrel{\widehat=} fb = plate\_at\_end \;\&\; drop \rightarrow fb : [\, true, fb' = no\_plate \,]$

$\bullet\ InitFeedBeltState;$
$\mu\, X \bullet (LoadFeedBelt;\ TransferPlate;\ UnloadFeedBelt);\ X$

end

Fig. 4. The *FeedBelt* process

– The robot picks up the plate from the press with its second arm, then rotates further to unload the plate on the deposit belt.
– The deposit belt transports the plate to the travelling crane.
– The crane picks up the metal plate from the deposit belt, moves to the feed belt, and then unloads the metal plate; a new cycle begins.

This sequence is further complicated by the fact that the robot can go back to the table and pick up a metal plate while the press is forging another one.

In [1] we have provided a formal model of the Production Cell in *Circus*. We have also used the above proof-by-refinement technique to prove that the proposed model is divergence-free. Due to space limitations, we cannot detail the full *Circus* model of the Production Cell or the formal proof conducted. Nonetheless, as another example of using the proof-by-refinement technique, we here present the *Circus* model of the *feed belt* machine, and prove that it is free from deadlock and divergence. The process is depicted in Figure 4 and can be described as follows:

Global Definitions

At any moment of time, either there is no plate on the belt, there is a plate on the belt but it has not reached the extreme end of the belt, or there is a

plate exactly at the extreme end of the belt [6].
The feed belt receives a metal plate from the crane, an event signalled by synchronisation on the channel *start*, and transports it to the other end, where it can *drop* it to elevating table.

Process State

The state of the feed belt is described with only one variable, *fb*.

Process Actions

The constituent actions of the process are

(i) InitFeedBeltState: Initially the feed belt is carrying no plates.

(ii) LoadFeedBelt: This action is enabled only when there is no plate on the belt. The arrival of a new plate to the belt is indicated by the synchronisation event on the channel *start*. The effect of the action is to change the status of the feed belt accordingly.

(iii) TransferPlate: This action is enabled only when there is a plate that is not at the extreme end of the feed belt. The effect of the action is to transfer that plate to the extreme end of the belt.

(iv) UnloadFeedBelt: The second action is unloading a plate to the table, an event signalled by a communication on the channel *drop*. The action is enabled only when there is a plate at the extreme end of the belt. The effect of this action is to change the state variable to indicate that there is no longer a plate on the belt.

Thus, the processing sequence of the feed belt is: load a plate, transfer it to the extreme end, then unload the plate to the table. This sequence is repeated indefinitely.

An interesting remark on the *FeedBelt* process, is that can be expressed as a sequential composition where the postcondition of each action in that sequential composition guarantees the guards of the following action. Also, the guards of each action guarantee the preconditions of that action. We formalise these remarks as a lemma [7], as we will need it in a later stage of this paper.

[6] For simplicity, we assume that there can be at most one plate on the feed belt at any moment of time. Even with this assumption the Production Cell can still process up to 5 plates at the same time.

[7] There is a generalized version of this lemma in [1], as similar remarks applied to all processes in our *Circus* model of the Production Cell.

Lemma 3.4 *Let Main be the main action of the FeedBelt process. Then, Main can be expressed as:* $Main = A_0;\ \mu X \bullet (g_1 \ \& \ A_1;\ \ldots;\ g_n \ \& \ A_n);\ X$ *Where*

- $A_0 = [\, S' \mid post'_0 \,]$
- *For all* $1 \leq i \leq n$, A_i *is a (prefixed) specification statement*

  $$A_i = c_i \rightarrow \alpha S : [\, pre_i, post'_i \,]$$
- *For all* i, pre_i *contains no dashed variables, and* $post'_i$ *contains no undashed variables.*

Furthermore, the following properties hold for g_i, pre_i, *and* $post'_i$

(i) $\forall\, i \mid 0 < i \leq n \bullet g_i \Rightarrow pre_i$

(ii) $\forall\, i \mid 0 \leq i < n \bullet post_i \Rightarrow g_{i+1}$

(iii) $post_n \Rightarrow g_1$

Proof. Straightforward, by inspection of the process' actions. □

Now, to show that the *FeedBelt* is deadlock-free and divergence-free, we start by defining an abstract model (*AFeedBelt*) of the process. Again, the abstract model (Figure 5) has the same structure, state components, types and invariants. It also uses the same channels, and with similar actions, except that this time there are no guards.

Theorem 3.5 *The abstraction AFeedBelt is both deadlock-free and divergence-free.*

Proof. By inspection of *Circus* syntax and semantics, there are eight conditions that are *sufficient* for a *Circus* process to be both deadlock and divergence-free:

(i) It is sequential.

(ii) It is free from hiding.

(iii) It doesn't mention *Stop* or *Chaos*.

(iv) All internal and external choices are over non-empty sets.

(v) Its channel types are non-empty.

(vi) It local definitions are satisfiable.

(vii) Its main action's initial state exists.

(viii) Its actions are all total on the state.

Conditions (i)–(iv) are satisfied syntactically. Condition (v) is trivially satisfied, since "start" and "drop" are synchronisation channels. Condition (vi) is also trivially satisfied, since and there are no local definitions. Condition (vii)

$BeltStatus ::= no\_plate \mid plate\_before\_end \mid plate\_at\_end$

channel $start, drop$

process $AFeedBelt \mathrel{\widehat=}$ **begin**

$AFeedBeltState \mathrel{\widehat=} [\, fb : BeltStatus \,]$

$InitAFeedBeltState \mathrel{\widehat=} [\, fb' : BeltStatus \,]$

$ALoadFeedBelt \mathrel{\widehat=} start \rightarrow fb : [\, true, fb' = plate\_before\_end \,]$

$ATransferPlate \mathrel{\widehat=} \Delta fb : [\, true, fb' = plate\_at\_end \,]$

$AUnloadFeedBelt \mathrel{\widehat=} drop \rightarrow fb : [\, true, fb' = no\_plate \,]$

$\bullet\ InitAFeedBelt;$
$\mu\, X \bullet (ALoadFeedBelt;\ ATransferPlate;\ AUnloadFeedBelt);\ X$

end

Fig. 5. The *AFeedBelt* process

can be stated as

$$\exists\, AFeedBeltState' \bullet InitAFeedBeltState$$

Expanding the schemas, we must prove that

$$\exists\, fb' : BeltStatus \bullet (fb' \in BeltStatus)$$

which is true, since *BeltStatus* is non-empty. Finally, condition (viii) follows trivially since all actions have no guards and *true* preconditions. □

We state and prove that *FeedBelt* is a refinement of *AFeedBelt* in Theorem 3.6 – consequently, this means that the *FeedBelt* process is also deadlock-free and divergence-free. The theorem makes use of the following laws:

Law 4 *Suppose that A is an action, g is a guard over A's state, and pre is an assumption over that state.*

$$\{pre\};\ A \quad = \quad \{pre\};\ g\ \&\ A$$

provided $pre \Rightarrow g$

□

If *pre* does not hold, both actions above abort. If *pre* does hold then the introduction of the guard has no effect, as g is guaranteed by *pre*.

Law 5 *Suppose that A is an action, g is a guard over A's state, and pre_1 and pre_2 are assumptions over that state.*

$$\{pre_1\};\ \mu X \bullet (A;\ \{pre_2\});\ X \quad = \quad \{pre_1\};\ \mu X \bullet (g\ \&\ A;\ \{pre_2\});\ X$$

provided

(i) $pre_1 \Rightarrow g$

(ii) $pre_2 \Rightarrow g$

□

If pre_1 does not hold, both actions above abort. If pre_1 does hold then the μ expression is guaranteed to be entered at least once; pre_1 guarantees g. Now, if pre_2 holds then the cycle defined by the μ expression is repeated; again the introduction of the guard has no effect as g is guaranteed by pre_2. If pre_2 does not hold at the end of any cycle, then both of the μ expressions abort.

Law 6 *Suppose that w is a list of variables in the state schema S. Also, suppose that pre, post′, and assump′ are predicates.*

$$w : [\, pre, post' \wedge assump' \,] \quad = \quad w : [pre, post' \wedge assump' \,];\ \{assump\}$$

Syntactic Restrictions: *$assump'$ contains no undashed variables*

□

That is, the updates on the state can be expressed as an *assumption* for the following actions. We now show that the *FeedBelt* process is free from the risks of deadlock and divergence.

Theorem 3.6 *FeedBelt is deadlock- and divergence-free.*

Proof. It is sufficient to show that: $AFeedBelt \sqsubseteq FeedBelt$
Let *Main* and *AMain* be the main actions of *FeedBelt* and *AFeedBelt*, respectively.

$$AMain$$

$$= \{\ \textit{By Lemma 3.4 and definition of AFeedBelt}\ \}$$

$$A_0;\ \mu X \bullet (A_1;\ \ldots;\ A_n);\ X$$

$$= \{\ \textit{By Law 6}\ \}$$

$$A_0;\ \{post_0\};\ \mu X \bullet (A_1;\ \{post_1\};\ \ldots;\ A_n;\ \{post_n\});\ X$$

$= \{$ *By Laws 4–5 and ii–iii in Lemma 3.4* $\}$

$A_0;\ \{post_0\};\ \mu\, X \bullet (g_1\ \&\ A_1;\ \{post_1\};\ \ldots;\ g_n\ \&\ A_n;\ \{post_n\});\ X$

$= \{$ *By Law 6 and definition of* A_i*'s* $\}$

$A_0;\ \mu\, X \bullet (g_1\ \&\ A_1;\ \ldots;\ g_n\ \&\ A_n);\ X$

$= \{$ *By Lemma 3.4* $\}$

Main

Therefore, $AFeedBelt \sqsubseteq FeedBelt$ □

4 Conclusions

Formal proofs that a specification satisfies a certain property are valuable yet challenging. Using "abstraction" and "refinement" as the guiding principles, we have introduced a proof-by-refinement technique that can facilitate reasoning about properties of system specifications. The key idea is to show that the specification is a refinement of a more abstract model, in which the proofs are much easier to conduct. Of course, it is important here that the properties considered are preserved be the refinement relation. Otherwise, proving that an abstract model A satisfies a property P, and that $A \sqsubseteq S$, does not necessitate that S also satisfies P.

Similar ideas for proof-by-refinement can be traced in the literature. In CSP, for example, a process P can be proved to be deadlock free if it is a refinement of the process $NeverDeadlock = \mu\, X \bullet \prod b : \alpha P \rightarrow X$. Our approach is different in the sense that it can consider both state and behaviour refinement and it does not explicitly appeal to the formal semantics of the specification language concerned. Instead, the details of the semantic model are hidden away through the use of refinement laws. We are then left with the side conditions for these laws, which need to be discharged. These side conditions are in the form of Z or, indeed, first order logic. Thus, in effect, the logical framework of the proofs is independent of the semantic model of the specification language. This allows for elegant proofs that are more readable, and hence understandable; try, for example, to do prove the same properties of *ProtNat* using the formal semantics [12] of *Circus*[8]. Also, Z is already an established notation in both academia and industry, with no lack of tool support. Thus, Z tools can now be applied in a domain where

[8] Alternatively, refer to the proofs reported [2] and [3] where the former appeals to the semantics of *Circus* and the latter uses the proof-by-refinement technique

they previously have been thought to be ineffective; that is, reasoning about concurrent systems.

Here, and in [1,3], we have successfully applied the proof-by-refinement technique to different examples, ranging from simple processes like *ProtNat* to complex systems like the Production Cell. Though all our experiments are with *Circus* specifications, it should be clear that the technique is not limited to a particular specification language. Steps i–iii in Section 1 are general and can be employed for other specification languages and refinement techniques. Nonetheless, we acknowledge that more experience is needed, in order to provide a deeper understanding of the technique and the conditions that control its application.

A Proofs

Lemma A.1 *In Theorem 3.2, proving that AProtNat is refined by ProtNat requires us to prove that:*

$$[\,\Delta AProtNatState;\ id? : ProcID;\ d? : \mathbb{N}\,] \sqsubseteq \left\{ readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \right\}; writers := \{id\};\ data := d$$

Proof *Since $\theta AProtNatState = \theta ProtNatState$, the required follows directly from applying Law 3, provided that we can prove:*

$$readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \land ProtNatState \Rightarrow ProtNatState'\,[\,data', readers', writers' := d, readers, \{id\}\,]$$

$$= \{\text{ by definition of } ProtNatState' \}$$

$$readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \land ProtNatState \Rightarrow (\ data' \in \mathbb{N} \ \land\ readers' \in \mathbb{F}\, ProcID \ \land\ writers' \in \mathbb{F}\, ProcID \ \land\ (readers' \neq \emptyset \Rightarrow writers' = \emptyset) \ \land\ \#writers' \leq 1\ ) [\,data', readers', writers' := d, readers, \{id\}\,]$$

$$= \{\text{ by substitution }\}$$

$$readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \land ProtNatState \Rightarrow (\ d \in \mathbb{N} \ \land\ readers \in \mathbb{F}\, ProcID \ \land\ \{id\} \in \mathbb{F}\, ProcID \ \land\ (readers \neq \emptyset \Rightarrow \{id\} = \emptyset) \ \land\ \#\{id\} \leq 1\ )$$

$$= \{\text{ by assumption and from } ProtNatState \}$$

$$readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \land ProtNatState \Rightarrow (\ (readers \neq \emptyset \Rightarrow \{id\} = \emptyset) \ \land\ \#\{id\} \leq 1\ )$$

$$= \{ \text{by set theory, propositional calculus, and using } readers = \emptyset \}$$

$$readers \cup writers = \emptyset \land id \in ProcID \land d \in \mathbb{N} \land ProtNatState \Rightarrow true$$

Hence, the application of Law 3 is valid, and the result follows. □

References

[1] D. Atiya. *Verification of Concurrent Safety–critical Systems: The Compliance Notation Approach*. PhD thesis, University of York. Submitted in October 2004.

[2] D. Atiya, S. King, and J. C. P. Woodcock. Ravenscar protected objects: a *Circus* semantics. Technical Report YCS-2003-356, Department of Computer Science, University of York, UK, June 2003.

[3] D. M. Atiya, S. King, and J. C. P. Woodcock. A *Circus* semantics for Ravenscar protected objects. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 617–635. Springer-Verlag, 2003.

[4] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in *Circus*. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002.

[5] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2–3):146–181, 2003.

[6] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[7] C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: the Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23, Germany, 1998. Springer-Verlag.

[8] C. Lewerentz and T. Lindner, editors. *Formal development of reactive systems: case study Production Cell*, volume 891. Springer-Verlag, 1995.

[9] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[10] C. C. Morgan. *Programming from specifications*. Prentice Hall International, 2nd edition, 1994.

[11] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002 — Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

[12] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.