DIVISION OF COMPUTER SCIENCE

Using Testing Semantics to Show Safety and Liveness in the Development of CCS Specifications

Jean Baillie

Technical Report No.199

May 1994

Using Testing Semantics to Show Safety and Liveness in the Development of CCS Specifications

Jean Baillie

1 Introduction

Where formal methods are used in industrial software engineering, it is primarily as notation or language—an aid to the software engineer in thinking about and understanding the problem. What we address in this paper is the question of whether the laws of CCS [10, 11] can be used to *prove* important properties of a system, namely, safety and liveness. This begs questions about what precisely we mean by safety and liveness and how we model these properties within the constraints of the calculus. Ideally we should like to be able to prove an appropriate congruence between specification and design that would enable us to conclude that a property holding for the specification will also hold for the design. (We interpret 'design' to mean the refinement of a specification by decomposition.)

The problems created by τ are well known; the difficulty of proving a design bisimilar to its specification was demonstrated in [2], (amongst others). In that case study certain safety signals which formed part of the design but were not part of the interface were therefore restricted in the expanded composite system, their presence at the lower level being indicated by leading τ's in the expansion. This precluded bisimilarity with the specification and gave the impression of nondeterminism in what was arguably a deterministic system. All τ's look the same, even those which are the result of responsible signalling and which, it may be argued, actually *enable* the required determinism.

Although bisimulation is the standard model of equivalence in CCS, we may also give CCS agents *testing* semantics [5, 6], and indeed the testing equivalences and preorders are implemented in the Concurrency Workbench[4]. This semantics has a pleasing intuition and an elegant theory and enables us to demonstrate safety and liveness properties as these are associated with the congruences. (*Testing* equivalence is the conjunction of may and $must$ equivalences; may coincides with Hoare's trace model and, for convergent agents, $must$ coincides with failures [7].)

We present a small case study which will be used as a vehicle for illustrating the ideas presented. The example is that of a simple level crossing where all communications are synchronization signals; no data is passed. This is for two main reasons: first, we wish to use the Concurrency Workbench to test the behaviour of our specifications and, at the time of writing, parameter passing is not implemented: second, a large, complex system is not needed to illustrate the issues; the problems we will be addressing occur in quite simple examples. In addition, by considering a safety-critical system, attention is focussed on the importance of safety and liveness.

We begin in section 2 by looking briefly at what we mean by *safety* and *liveness* before moving on to

our case study. In section 3 the system is specified at a very high level of abstraction then, in section 4, taken through several stages of refinement. Each refinement is compared with its predecessor (of which it is a decomposition) for either equivalence or ordering, and the implications for safety and liveness are discussed. The orderings are the *testing* preorders, reflexive and transitive relations whose symmetric cases give the *kernels* of the preorders, that is, the associated equivalence classes. The Concurrency Workbench is used to construct these relations.

2 Safety and liveness

Lamport [8] suggests that concurrent systems may be specified in terms of their *safety* and *liveness* properties. Informally, safety is to do with the bad things a process *may not* do, and liveness with the good things it *must* do.

We consider these properties of specifications as they may be interpreted in CCS (rather than in temporal logic, as Lamport). There is no facility in CCS for defining, for example, invariant properties of a process that will guarantee its safety throughout its execution, or for translating the temporal logic assertion that a particular action must eventually occur, thereby guaranteeing liveness. Instead, we shall discuss the safety of a process in terms of the sequences of actions in which it may engage—that is, its traces—and liveness in terms of the sequences of actions in which it $must$ engage.

2.1 Safety

What is meant by the safety of a system is very much context-dependent. In industrial process control, for example, critical values of temperature and pressure may need to be included in the safety specification. In the case of the level crossing, however, where safety amounts to ensuring that cars and trains do not occupy the critical region of the crossing at the same time, we can model safety in CCS as the prohibiting of unsafe traces. For example, we can specify a system in which the only visible actions are a and b and where all sequences are safe except for those containing 2 or more a's together:

$$Spec \stackrel{def}{=} a.b.Spec + b.Spec$$

We can state that may equivalence between any design and this specification is a proof of this particular safety property. If it has been shown that a specification contains no unsafe traces and also that a design is may equivalent to that specification then we can be sure that the design also is safe. Clearly any design D of which we can write $D \sqsubset_{may} Spec$ will also be safe *vis-à-vis* $Spec$ since D's traces will be a subset of those of $Spec$; and since safety does not require that a process does anything at all then theoretically, at least, this is satisfactory. To the software engineer, however—and certainly to his customer—this is quite *un*satisfactory; $\perp$ may be safe, but it has few practical uses. We can think of may equivalence as a maximal safety condition.

2.2 Liveness

In general, liveness is the property of a system whereby every action which is required to occur does eventually do so. Given that in CCS we have no guarantee of fairness, we interpret this in its negative sense: a system is defined to be live if it is not unnecessarily prevented from making progress. Nondeterminism may block such progress; broadly, $must$ semantics equates or orders processes on the basis of nondeterminism (and divergence), making it an appropriate measure for the liveness property.

The liveness condition will be stated in terms of the required behaviour (that is, successful $must$ testing) of a specification under restriction; the question we shall ask in order to establish liveness is whether, when certain actions are prevented from occurring, progress $must$ be made by the remaining visible actions. Nondeterminism may preclude this. For example, the process

$$P \stackrel{def}{=} a.P + \tau.b.P$$

is not live with respect to a; if we restrict by b the leading τ may still prevent a. It is, though, live with respect to b.

If a specification can be shown to possess a particular liveness property—that is to say, it is not unnecessarily prevented from behaving in a required manner—and $must$ equivalence can be shown to hold between that (possibly restricted) specification and a (similarly restricted) design, then that design will enjoy the same liveness property.

3 The Specification

The level crossing is viewed as a shared resource to which cars and trains have mutually exclusive access. It admits one car or one train at a time (i.e., the car or train must leave the crossing before anything else is admitted). The signals car_app, $train\_app$ are sent by approaching vehicles; car_cross, $train\_cross$ are sent by vehicles which have been allowed to cross or, in other words, are inside a mutual exclusion zone. Note that in this model, trains always have priority; cars may be held up to allow a train (or trains) to cross, but never the other way around. This specification implicitly states the safety condition, namely, that cars and trains cannot interleave on the crossing.

$$\begin{aligned} Spec \stackrel{def}{=} \quad & car\_app.(car\_cross\,.Spec + train\_app.train\_cross\,.Spec') \\ & + \\ & train\_app.(train\_cross\,.Spec + car\_app.train\_cross\,.Spec') \end{aligned}$$

$$Spec' \stackrel{def}{=} \quad car\_cross\,.Spec + train\_app.train\_cross\,.Spec'$$

Making the substitution c for car_app, d for car_cross, u for $train\_app$ and v for $train\_cross$, we

have

$$\begin{aligned} Spec &\stackrel{def}{=} \quad c.(d.Spec + u.v.Spec') \\ &\quad\quad + \\ &\quad\quad u.(v.Spec + c.v.Spec') \end{aligned}$$

$$Spec' \stackrel{def}{=} \quad d.Spec + u.v.Spec'$$

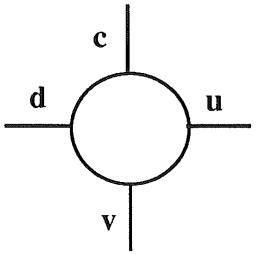

Figure 1: The Level Crossing: $Spec$

The *safety* condition may be defined in terms of legal and illegal traces; for example, $cduv$, $uvcd$, $cuvd$, $ucvd$ are all legal traces. The last two may be interpreted to mean that if a car is seen to approach the crossing either immediately before or immediately after a train has been observed, the train must be allowed to proceed *before* the car. Theoretically, any number of trains may be permitted through the crossing, causing the car to wait indefinitely: $cuv(uv)^*d$ and $ucv(uv)^*d$ are all legal traces. This may be unlikely in practice but is nevertheless safe behaviour. Examples of illegal traces are $cudv$ and $ucdv$; the implication of each of these is that a car has been allowed to enter the mutual exclusion zone before the train has sent the signal that it has left.

We also require that in the absence of a train, the crossing behaves like a road and similarly, in the absence of cars, like a track. That is to say, we wish that

$$Spec \setminus \{u, v\} = Road \text{ where } Road \stackrel{def}{=} c.d.Road$$

and

$$Spec \setminus \{c, d\} = Track \text{ where } Track \stackrel{def}{=} u.v.Track$$

We find that $Spec \setminus \{u, v\} = c.d\,.Spec \setminus \{u, v\}$ and that $Spec \setminus \{c, d\} = u.v\,.Spec \setminus \{c, d\}$, which is what is required. This is the *liveness* property.

4 The Design

4.1 First Refinement

We first define two agents, a road $Light$ and a R_Sensor; the light shows $green$ if no train is approaching, i.e., it also acts implicitly as a train sensor. The $gone$ signal is received when the car

has crossed safely. (This would seem to imply that the train is physically stopped if a car gets stuck, though we need not be concerned with that at this level of abstraction.)

$$Light1 \stackrel{def}{=} \overline{green}.gone.Light1 + u.v.Light1$$

$$R\_Sensor \stackrel{def}{=} c.green.d.\overline{gone}.R\_Sensor$$

Then

$$R1 \stackrel{def}{=} Light1 | R\_Sensor \setminus \{gone, green\}$$

The expansion law gives

$$\begin{aligned} R1 &= c.(\tau.d.R1 + u.v.R1'_1) \\ &\quad + \\ &\quad u.(v\,R1 + c.v.R1'_1) \\ R1'_1 &= \tau.d.R1 + u.v.R1'_1 \end{aligned}$$

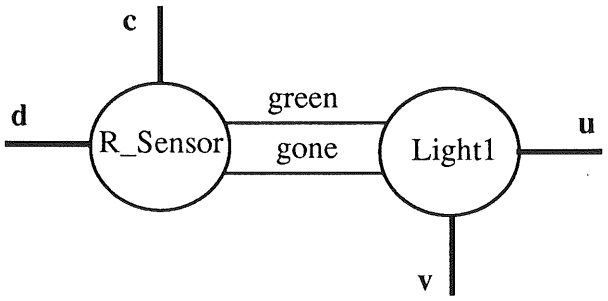

Figure 2: First refinement: R1

The Concurrency Workbench gives $R1 \approx_{may} Spec$; this guarantees that $R1$ cannot admit illegal traces such as $ucdv$ or $cudv$ (allowing cars to interleave trains) so safety is assured. This equivalence is essential; we need to be sure that the implementation allows no illegal traces. In addition, by insisting on $\approx_{may}$ we preclude a chain degenerating to the undefined process $\perp$ (denoted by Ω in [6] and @ in the Concurrency Workbench).

We also find that

$$R1 \setminus \{u, v\} \approx_{must} c.d.R1 \setminus \{u, v\}$$

and

$$R1 \setminus \{c, d\} \approx_{must} u.v.R1 \setminus \{c, d\}$$

that is, the liveness property is preserved.

4.2 Second Refinement

An agent $Train$ is introduced such that $Train \mid Light2$ is a decomposition of $Light1$. $Train$ sends the visible signals u and v to the environment, and the hidden signals $sensin$ and $sensout$ to the agent $Light2$.

$$Train \stackrel{def}{=} \overline{sensin}.u.v.\overline{sensout}.Train$$

$$Light2 \stackrel{def}{=} \overline{green}.gone.Light2 + sensin.sensout.Light2$$

with $TL2 = Train \mid Light2 \setminus \{sensin, sensout\}$, that is, $TL2$ is a refinement of $Light1$

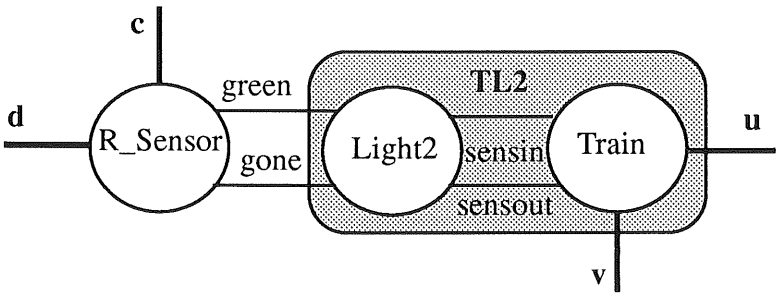

Figure 3: Second refinement: R2

We find that $TL2 \approx_{may} Light1$ and that $TL2 \sqsubset_{\sim must} Light1$.

So, then, if we define

$$R2 \stackrel{def}{=} TL2 \mid R\_Sensor \setminus \{green, gone\}$$

we find, from the concurrency workbench, that

$$R2 \approx_{may} R1$$

that is, $R2$ is safe. The liveness property is weakened, however, i.e.,

$$R2 \setminus \{c, d\} \approx_{must} Track$$

but

$$R2 \setminus \{u, v\} \sqsubset_{\sim must} Road$$

This implies that even in the absence of trains, cars might still be prevented from using the crossing. Though this might not agree with our intuition about our design it is unavoidable in the CCS model because of a leading τ in the composition. In the absence of cars, though, the crossing is live for trains.

4.3 Third refinement

We now add an agent $Control$ such that $Control \mid Light3$ is a decomposition of $Light2$. $Control$ receives the signals $sensin$, $sensout$ from the Train. Where no train is approaching, ok is sent by $Control$ to $Light3$ signalling that it is safe to show $green$, and the signal $gone$ is now sent (by R_Sensor) to $Control$ and not to $Light3$.

$$Control \stackrel{def}{=} \overline{ok}.okt.gone.Control + sensin.sensout.Control$$

$$Light3 \stackrel{def}{=} ok.\overline{green}.\overline{okt}.Light3$$

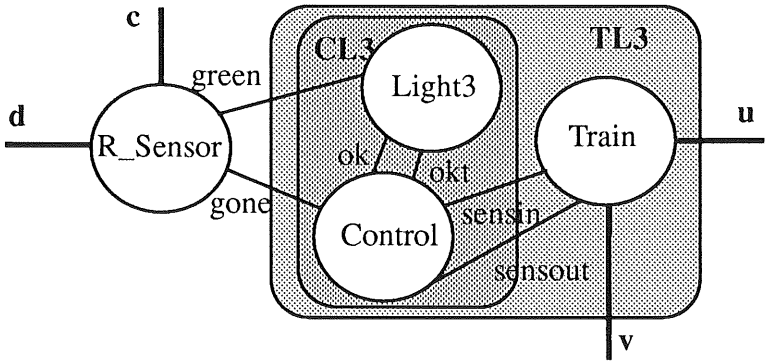

Figure 4: Third refinement: R3

Then

$$CL3 \stackrel{def}{=} Control \mid Light3 \setminus \{ok, okt\}$$

and

$$TL3 \stackrel{def}{=} CL3 \mid Train \setminus \{sensin, sensout\}$$

We define

$$R3 \stackrel{def}{=} TL3 | R\_Sensor \setminus \{green, gone\}$$

From the Concurrency Workbench we have

$$R3 \approx_{may} R2$$

that is, $R3$ is safe, but the liveness property has been weakened still further. We now find that

$$R3 \setminus \{c, d\} \sqsubset_{must} Track \text{ and } R3 \setminus \{u, v\} \sqsubset_{must} Road$$

Equality has been lost through internal nondeterminism.

Summarizing, the orderings are

$$R3 \approx_{may} R2 \approx_{may} R1 \approx_{may} Spec$$

and

$$R3 \setminus \{c,d\} \sqsubset_{\sim must} R2 \setminus \{c,d\} \approx_{must} R1 \setminus \{c,d\} \approx_{must} Track$$

and

$$R3 \setminus \{u,v\} \sqsubset_{\sim must} R2 \setminus \{u,v\} \sqsubset_{\sim must} R1 \setminus \{u,v\} \approx_{must} Road$$

5 Conclusion

The safety property (may equivalence with $Spec$) has been relatively easy to maintain throughout the refinement process; liveness—characterized by $must$ equivalence with either $Road$ or $Track$—has not.

Where may equivalence with the top level specification is provable we can say with confidence that the model is safe—or at least, as safe as its specification. But the liveness condition is weakened (within the limits of the expressive power of the calculus) by internal nondeterminism. This does not reflect our intuition about the behaviours of the systems we have designed; whilst we may have confidence in our designs, their safety, liveness and so on, there is no way around the fact that once actions have been restricted in a CCS expansion, the only indicator of their existence is τ and all τ's look the same; in the design, the restricted actions may be *enabling* liveness, whereas after expansion they may seem to be precluding it.

Whilst this case study does not formally prove the contention that $must$ equivalence between specifications and implementations cannot generally be established, it is nonetheless clear that the problem is a general one; only by either contrivance or luck can it be eliminated in any particular case. Leading τ's in expanded systems are the rule rather than the exception, so any attempt to prove an equivalence stronger than may between a full behavioural specification and its behavioural implementation is likely to run into the same problem.

There are a number of possible ways forward; one of these might be through *partial* specifications [1], possibly expressed as a set of modal formulae, stating required properties of the system and identifying proof obligations (a full modal specification only gives an alternative bisimulation semantics). An alternative approach is to build a degree of nondeterminism into the specification, intended (in this case study) to mean that we don't mind whether cars or trains use the crossing first, though the semantics of unstable agents together with the fact that it is not possible to restrict the scope of τ make the required behaviour difficult to model.

The work of Glenn Bruns in collaboration with Praxis Sytems plc [3] is also relevant. In that paper, Modal Process Logic [9] was used to verify properties of a failure recovery protocol using the notions of *admissable* and *necessary* transitions; an implementation is deemed to be a refinement of a specification if every necessary action of the specification is matched by a necessary action of

the implementation, and every admissable action of the implementation is matched by an admissable action of the specification. It may be that properties of safety and liveness of an implementation *vis-à-vis* a specification can be captured in this way; this is a subject for future work.

References

[1] E. J. Baillie. *Towards a Satisfaction Relation between CCS Specifications and their Refinements.* PhD thesis, University of Hertfordshire, 1992.

[2] Jean Baillie. A CCS case study: a safety-critical system. *Software Engineering Journal*, 6(4), 1991.

[3] Glenn Bruns. Applying process refinement to a safety-relevant system. Technical Report ECS-LFCS-94-287, LFCS, Department of Computer Science, University of Edinburgh, 1994.

[4] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench. In *Automatic Verification methods for Finite State Systems*, pages 24–37, 1989.

[5] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[6] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall International, 1985.

[8] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming languages and systems*, 5(2):190–222, 1983.

[9] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Third Annual Symposium on Logic in Computer Science*, 1988.

[10] R. Milner. *A Calculus of Communicating systems.* Springer Verlag, 1980. LNCS 92.

[11] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.