TESTING THE CONSEQUENCES OF SPECIFICATIONS IN MODAL μ

Ying Liu, John Aldwinckle, Graham Birtwistle, Ken Stevens,
Department of Computer Science, University of Calgary, Canada.

Abstract

In a companion paper in these proceedings [6], we introduced the CCS notation and explained how to write specifications succinctly in CCS using the composition operator. In this paper we explain how one may associate a process logic with CCS and use it to resolve deadlock, safety, liveness, and fairness properties of specifications by static testing.

1 Introduction

Specifications tell us what a system should do — not how it does it. They are thus simpler and shorter than implementation descriptions. Since equivalent descriptions share exactly the same properties, it pays to investigate properties of a system by testing its specification rather than testing its implementation.

In this paper we couple a process logic to CCS and show how to test for such properties as deadlock, safety, liveness and fairness. See [5, section 2, pages 177–387] for a very readable and full account. [1] gives the intuition and background to minimum and maximum fix points. [7] is an excellent account of process logics and CCS. Ms Liu's thesis [4] applies these techniques to asynchronous hardware.

2 HML — Hennessy-Milner logic

Labelled transition systems. The processes of CCS generate labeled transition systems of the form ($\mathcal{P}$, $\mathcal{A}$, $\mathcal{T}$) where

- $\mathcal{P}$ is a non-empty set of agents
- $\mathcal{A}$ is a set of input actions (α) and output actions ($\overline{\alpha}$)
- $\mathcal{T}$ are the transition relations for each α (or $\overline{\alpha}$) $\in \mathcal{A}$.

E.g. given the system which describes a two place buffer,

$$\begin{array}{lcllll} B_0 & \stackrel{def}{=} & put.B_1 & & \\ B_1 & \stackrel{def}{=} & put.B_2 & + & \overline{get}.B_0 \\ B_2 & \stackrel{def}{=} & & & \overline{get}.B_1 \end{array}$$

then

$$\begin{array}{lcllll} \mathcal{P} & = & \{B_0, B_1, B_2\} \\ \mathcal{A} & = & \{put, \overline{get}\} \\ \mathcal{T} & = & \{B_0 & \stackrel{put}{\rightarrow} & B_1, \\ & & B_1 & \stackrel{put}{\rightarrow} & B_2, \\ & & B_1 & \stackrel{\overline{get}}{\rightarrow} & B_0, \\ & & B_2 & \stackrel{\overline{get}}{\rightarrow} & B_1\} \end{array}$$

HML. The syntax of HML formulae is:

$$A ::= T \mid \neg A \mid A \wedge A \mid < a > A \mid [a]A$$

with interpretation:

- T is the constant true formula
- ¬A is a negated formula
- A ∧ A is a conjunction
- modalised terms:
  <a> A is read as "A holds after some a action"
  [a] A is read as "A holds after all a actions"

We derive F, $\vee$, $\supset$, $\equiv$, ... from the basic operators. Notice that [ ] and <> are duals of each other since $< a > A = \neg[a]\neg A$ — only one need be defined as a primitive.

Since the formulae are parameterised by the action set, each transition system has its own associated HML.

Satisfaction over HML. For a given fixed transition system, we now define when a process E $\in \mathcal{P}$ satisfies the property A (written E $\models$ A):

1	E	$\models$	T	$\forall$ E
2	E	$\models$	¬ A	iff E $\not\models$ A
3	E	$\models$	A ∧ B	iff E $\models$ A and E $\models$ B
4	E	$\models$	<a> A	iff $\exists$ E' $\in \mathcal{P}$, a $\in \mathcal{A}$. E $\stackrel{a}{\rightarrow}$ E' and E' $\models$ A
5	E	$\models$	[a] A	iff $\forall$ E' $\in \mathcal{P}$, a $\in \mathcal{A}$. E $\stackrel{a}{\rightarrow}$ E' $\supset$ E' $\models$ A

with interpretation

1. Every process in $\mathcal{P}$ satisfies property T
2. A process has property ¬A when it fails to have property A
3. A process has property A ∧ B when it has property A and it has property B
4. A process satisfies <a> A if there exists one a action whose resulting process has property A

5. A process satisfies [a] A if after every performance of an a action all the resulting processes have property A

Example: 2 place buffer

1. $B_0 \models$ <*put*> T the buffer can add an item in B_0
2. $B_0 \models \neg$(<$\overline{get}$> T) the buffer cannot remove an item from B_0
3. $B_1 \models$ <$\overline{get}$> T $\wedge$ <*put*> T B_1 can both remove and add an item
4. $B_2 \models$ <$\overline{get}$> T $\wedge$ $\neg$(<*put*> T) B_2 can remove an item, but can not add an item

... and more notation. We make HML more convenient to use by allowing the following notational extensions:

[-]	$\stackrel{def}{=}$	all actions $\mathcal{A}$
[-k,l,m]	$\stackrel{def}{=}$	all actions except k,l,m in $\mathcal{A}$
<a,b,c> S	$\stackrel{def}{=}$	<a> S $\vee$ <b> S $\vee$ <c> S
[a,b,c] S	$\stackrel{def}{=}$	[a] S $\wedge$ [b] S $\wedge$ [c] S

In particular

E $\models$ [a] F	E cannot do an a move
E $\models$ <a> T	E can do an a move
E $\models$ [-] F	E is deadlocked
E $\models$ <-> T	E is live
E $\models$ <a> T $\wedge$ [-a] F	E can only do an a

3 Recursive agents

All interesting agents are recursive. Our buffer has the property that once in state B_1 all we can do is either a *put* followed by a $\overline{get}$, or a $\overline{get}$ followed by a *put*, and then we are back in state B_1 again. I.e. we can keep on doing this forever. An obvious notation in which to express this infinite behaviour is:

$$B_1 = (<\overline{get}> <put> \vee <put> <\overline{get}>)B_1$$

Such a *fix point* equation may have no solutions (e.g. X = $\neg$ X) or several solutions. There will always be at least one solution provided that each fix point variable is within the scope of an even number of negations. From now on, we assume that all our modal formulae pass this simple syntactic test.

Satisfaction via sets of states. We associate with a property the set of states satisfying it:

|| A ||	$\stackrel{def}{=}$	set of all states satisfying A
|| T ||	$\stackrel{def}{=}$	true of all states = $\mathcal{P}$
|| F ||	$\stackrel{def}{=}$	true of no state = $\emptyset$
|| $\neg$A ||	$\stackrel{def}{=}$	$\mathcal{P}$ - || A ||
|| A $\wedge$ B ||	$\stackrel{def}{=}$	|| A || $\bigcap$ || B ||
|| <a> A ||	$\stackrel{def}{=}$	true iff it is possible to do an a and move to a state enjoying A
|| [a] A ||	$\stackrel{def}{=}$	true iff however we do an a we move to a state enjoying A TRUE if we cannot do an a

Example: 2 place buffer (cont)

Here are some satisfaction relations over the 2 place buffer

Property p		Set of states with p
|| T ||	=	{ B_0, B_1, B_2 }
|| <-> T ||	=	{ B_0, B_1, B_2 }
|| <$\overline{get}$> T ||	=	{ B_0, B_1 }
|| <*put*> T ||	=	{ B_1, B_2 }
|| <$\overline{get}$>T$\wedge$[$-\overline{get}$]F ||	=	{ B_0 }
|| <$\overline{get}$>T$\wedge$<$-put$>T ||	=	{ B_1 }
|| <*put*> T $\wedge$ [$-put$] F ||	=	{ B_2 }
|| [-] F ||	=	{ }
|| F ||	=	{ }

Min and max fix points. In general when we look for the fix points of a formula several sets of states might be solutions. And since there may be several solutions, key questions to ask are: how do we find them? are there any of special interest?

If we wish to find all the fix points, we could test all the possible combinations from the empty set, the sets of singletons, two states at a time, ..., all the way up to $\mathcal{P}$. It turns out that the fix points form a lattice and that the "least" and "largest" of solutions are not only unique, but they also have interesting physical interpretations and fast algorithms.

The minimum (least) fixpoint includes only what is necessarily true. It expresses ***liveness***: e.g. a must eventually happen. It is found by iteration: we start from the empty set of states and include what must be there.

The maximum (largest) fix point includes everything except that which is necessarily false. It expresses *safety*: e.g. a holds everywhere. It is found by iteration: start with all possible states and pare out those found wanting.

Raw modal μ. Modal μ extends HML with fix points:

A ::= $HML \mid min(X.A) \mid max(X.A)$

where X is a fix point variable. min and max are dual operations — only one need be defined as a primitive.

Unfortunately properties written in raw modal μ are rather hard to read. As an example, a test for the absence of deadlock is:

$$max(X. <-> T \wedge [-]X)$$

It expresses the set of states X which can themselves make a move ($<-> T$) and from which all moves ($[-]$) takes us to a member of the set of states X which can ... Thus, no member of this set of states is incapable of making a move.

Since this is a relatively simple test, it doesn't take much imagination to to realise that complicated properties can be very hard to for humans to interpret. The same game has been played for many years by temporal logicians who have come up with a few basic operators that may be composed. Amongst these are

ALWAYS	needs all states on all paths as witnesses
PATH	needs all states on a single path to be witnesses
POSSIBLE	needs only a single state on a single path as a witness
EVENTUALLY	needs a single witness on all paths

and modal μ is powerful enough to express them all:

$max(X.P \wedge [-]X)$	$ALWAYS$
$max(X.P \wedge <->X)$	$PATH$
$min(X.P \vee [-]X)$	$EVENTUALLY$
$min(X.P \vee <->X)$	$POSSIBLE$

These operators are easier to understand, and we use them rather than raw modal μ.

4 Property testing in modal μ

DEADLOCK means that a system may reach a state in which it cannot make a move (is stuck). For any system SYS, absence of deadlock may be expressed as:

$$SYS \models \text{ALWAYS} <-> \text{T}$$

read as "in every state (ALWAYS), it is possible to make a move (<-> T)", or by its dual

$$SYS \models \neg(\text{POSSIBLE } [-] \text{ F})$$

read as "it is not true that there exists a path to a state (POSSIBLE) in which every move is impossible ([-] F)".

FAIRNESS means that a system can not "spin" forever without enabling some particular input or output action. For any system SYS, and for a particular action a, this may be expressed as:

$$SYS \models \text{ALWAYS} \neg \text{ PATH} \neg <a> \text{T}$$

read as "from every state (ALWAYS) there does not exist a path ($\neg$ PATH) to a state where action a is never enabled ($\neg$ <a> T)" or by its dual

$$SYS \models \text{ALWAYS EVENTUALLY} <a> \text{T}$$

which reads as "from every state (ALWAYS) for each path (EVENTUALLY) there is a state in which a is enabled (<a> T)".

SAFETY tests to see that bad things cannot happen. Safety tests must be tailored to the system at hand. For the two place buffer, we may want to check that it is never possible to output three times without doing an input.

$$B_0 \models \text{ALWAYS } [\overline{get}][\overline{get}][\overline{get}]\text{F}$$

read as "in every state (ALWAYS) it is not possible to perform three consecutive $\overline{get}$ actions ($[\overline{get}][\overline{get}][\overline{get}]$F).

LIVENESS tests to see that good things may happen (e.g. each request may be accepted)

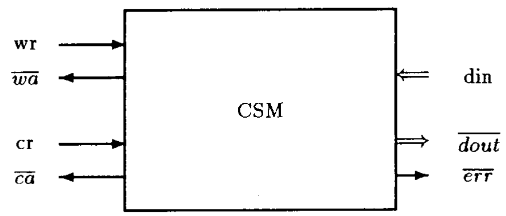
For any system SYS, and a particular action a, liveness of action a may be expressed as:
$$B_0 \models \text{ALWAYS POSSIBLE} <a> \text{T}$$

read as "from every state (ALWAYS) there exists a path (POSSIBLE) to some state where action a is enabled (<a> T)".

Fairness can be seen as a stronger form of liveness.

5 CSM: Dill *et. al*'s memory

In this section we put it all together using a recently published example. [3], Dill *et. al* describe (but do not specify) a FIFO storage management control system using Petri nets.



This CSM has to deal with two types of request:

1. *WRITE* which claims a storage location and then puts data read from *din* into it
2. *CLEAR* which emits the "next" data item (when one is available) on $\overline{dout}$ and then frees that location

The implementation they have in mind uses a circular buffer as basic storage. The storage is guarded by a controller which prevents *din* and $\overline{dout}$ occurring together (mutual exclusion), and also refuses writes when the buffer is full and refuses clears when the buffer is empty.

This specification is given shape by considering W (write) and C (clear) agents running in parallel.

$$W \stackrel{def}{=} wr.din.\overline{wa}.W$$
$$E \stackrel{def}{=} cr.\overline{dout}.\overline{ca}.E$$

$$CSM \stackrel{def}{=} (W \mid E) \setminus \{timing\ constraints\}$$

We now define a counter which is used to keep track of the number of used slots in the system. Every time a slot is given out it counts up and every time a slot is returned it counts down. Since we need to be able to test the state of the counter before actual accesses are made, the counter maintains a number of flags: f (full) and nf (notfull); and e (empty) and ne (notempty).

Here we use a 3-counter — a counter of arbitrary size is defined in the same manner, Notice that this counter fails with a signal on $\overline{err}$ should we attempt to up a full count or down an empty count.

$$\begin{array}{llllll}
C_3 & \stackrel{def}{=} & \overline{down}.C_2 & + \overline{up}.\overline{err}.O & + \overline{f}.C_3 & + \overline{ne}.C_3 \\
C_2 & \stackrel{def}{=} & \overline{down}.C_1 & + \overline{up}.C_3 & + \overline{nf}.C_2 & + \overline{ne}.C_2 \\
C_1 & \stackrel{def}{=} & \overline{down}.C_0 & + \overline{up}.C_2 & + \overline{nf}.C_1 & + \overline{ne}.C_1 \\
C_0 & \stackrel{def}{=} & \overline{down}.\overline{err}.O & + \overline{up}.C_1 & + \overline{nf}.C_0 & + \overline{e}.C_0
\end{array}$$

Our second approximation to the specification is:

$$W \stackrel{def}{=} wr.nf.up.din.\overline{wa}.W$$
$$E \stackrel{def}{=} cr.ne.\overline{dout}.down.\overline{ca}.E$$
$$C_0 \stackrel{def}{=} as\ above$$

$$CSM \stackrel{def}{=} (W \mid E \mid C_0 \mid S) \setminus \{down, up, f, nf, e, ne\}$$

in which W tests the flag nf to ensure a slot before claiming it with an *up* and reading in the data, and E tests the flag *ne* to ensure data is there before writing it out and then returning the slot with a *down*.

Our last step is to ensure that data inputs and outputs are mutually exclusive. All we need add is an extra semaphore:

$$
\begin{array}{lcl}
W & \stackrel{def}{=} & wr.nf.gS.up.din.\overline{pS}.\overline{wa}.W \\
E & \stackrel{def}{=} & cr.ne.gS.\overline{dout}.down.\overline{pS}.\overline{ca}.E \\
C_0 & \stackrel{def}{=} & as\ above \\
S & \stackrel{def}{=} & \overline{gS}.pS.S \\
 & & \\
CSM & \stackrel{def}{=} & (W \mid E \mid C_0 \mid S) \\
 & \backslash & \{down, up, f, nf, e, ne, gS, pS\}
\end{array}
$$

Dill *et. al* state the desirability of testing the specification for mutual exclusion on data input and output and ensuring that data cannot be input when SM is full and output when SM is empty. The Concurrency Work Bench (CWB) [2] has a fully automatic model checker, which makes these and other more searching questions trivial to ask:

- Is it always possible to make some move? i.e. no deadlock.

  $\text{CSM} \models \text{ALWAYS} <\!-\!> \text{T}$

- No bus data contention, i.e. it is never possible to both input and output.

  $\text{CSM} \models \text{ALWAYS} \neg((<\!\text{din}\!>\text{T} \wedge <\!\overline{dout}\!>\text{T})$

- we can neither overflow nor underflow the stack.

  $\text{CSM} \models \neg(\text{POSSIBLE} <\!\overline{err}\!>\text{T})$

- is "din" a live transition?

  $\text{CSM} \models \text{ALWAYS POSSIBLE} <\!\text{din}\!>\text{T}$

- is the system "fair" on din?

  $\text{CSM} \models \text{ALWAYS EVENTUALLY} <\!\text{din}\!>\text{T}$

- is the input protocol respected? i.e. no din without a wr; no $\overline{wa}$ without a din; no $\overline{wa}$ without a wr

  $$
  \begin{array}{lcl}
  \text{CSM} & \models & \text{CYCLE wr din} \\
   & & \wedge\ \text{CYCLE din}\ \overline{wa} \\
   & & \wedge\ \text{CYCLE wr}\ \overline{wa}
  \end{array}
  $$

  where we introduce a new operator called $CYCLE$:

  $$
  \begin{array}{l}
  \text{CYCLE a b} \stackrel{def}{=} \\
  \quad \max(\text{X}.[\text{b}]\text{F} \wedge [\text{-a}]\text{X} \wedge \\
  \quad [\text{a}](\max(\text{Y}.\ [\text{a}]\text{F} \wedge [\text{-b}]\text{Y} \wedge [\text{b}]\text{X})))
  \end{array}
  $$

  This describes the set of states X where no b action is possible, and any action other than a will take us back into this set of states, and an a action will take us to a set of states Y where, no a action is possible, and any action other than a b will take us back into this set of states, and a b action will take us back into the set of states X where ... In essence, it ensures that action a always preceeds action b which always preceeds action a which ...

- is the output protocol respected? i.e. no $\overline{dout}$ without a cr; no $\overline{ca}$ without a $\overline{dout}$; and no $\overline{ca}$ without a cr.

  $$
  \begin{array}{lcl}
  \text{CSM} & \models & \text{CYCLE cr}\ \overline{dout} \\
   & & \wedge\ \text{CYCLE}\ \overline{dout}\ \overline{ca} \\
   & & \wedge\ \text{CYCLE cr}\ \overline{ca}
  \end{array}
  $$

6 Conclusions

If specifications for circuits are written in a formal specification language such as CCS, it is possible to automatically and mechanically check properties of the specification using the modal μ-logic. These methods are complementary, as CCS uses an operational description of circuit behaviour, while the Modal μ logic describes properties of specifications independently from their implementation.

For details of how to specify complex systems in CCS see the companion paper in these proceedings [6].

7 Acknowledgements

This research is supported by Equipment and Operating Grants from CMC and NSERC, studentships from Hewlett Packard (KS) and The Alberta Microelectronic Centre (YL), and AGT/SEED (JA).

References

[1] J. Aldwinckle, R. Nagarajan, and G. Birtwistle. An introduction to Modal Logic and its Applications on the Concurrency Workbench. Technical Report, Computer Science Department, University of Calgary, 1991.

[2] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification fo concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[3] D. Dill, S. Nowick, and C. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in Systems Design*, 1(1):30–60, 1992.

[4] Y. Liu. Reasoning about asynchronous designs in CCS. MSc Thesis, Department of Electrical and Computer Engineering, University of Calgary, 1992.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification.* Springer-Verlag, New York, 1992.

[6] K. Stevens, J. Aldwinckle, G. Birtwistle, and Y. Liu. Designing parallel specifications in CCS. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.

[7] C. Stirling. Modal and Temporal Logics for Processes. Tech Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1992.