Supporting Formal Reasoning for Safety Critical Systems

C.A.Gurr*
Human Communication Research Centre
University of Edinburgh
2 Buccluech Place, Edinburgh EH8 9LW Scotland

April 1994

Abstract

Formal methods can significantly assist in the design and modelling of safety-critical systems. However, formal methods are frequently criticised as being unusable through being too complex and requiring expert knowledge to use. We assert that to make formal methods usable they must be able to be presented in a manner which is readily interpretable. However, we must ensure that the inferences which may be drawn from such a presentation are correct with respect to the formal semantics.

Concurrent systems in which communication occurs between asynchronously operating agents are widely used in safety-critical applications. Unfortunately designing and understanding such systems is made difficult by the interactions between the various concurrent agents.

We present an exercise in the specification and modelling of a safety-critical multiprocessing system fragment. This serves to illustrate three issues which are crucial to the design and modelling of a safety-critical system. These are the advantage of a formal approach, particularly for concurrent systems, the importance of ensuring that a formal model correctly represents the real system and the need to provide a user with a clear understanding (or *visualisation*) of the formal model. For this latter point we propose, with examples, the efficacy of a well-founded graphical representation in supporting such an understanding.

This paper was presented at the British Computer Society Modular Languages SIG conference "Safety Through Quality", Old Windsor (UK), June 6-7 1994.

Keywords: safety, communication, concurrency, CCS, statechart.

1 Introduction

Formal methods can be highly advantageous in safety cases, removing ambiguities in specifications and making explicit assumptions which might otherwise remain hidden. However, such specifications can only be of assistance if they are both correct and usable. A formal model of a safety-critical system must capture all of the relevant properties of that system. In addition, people from a range of disciplines must review and contribute to a safety case. Thus it is essential that: (i) the complex information which the methods involve be presented in a form which can be readily understood by people with different levels of technical expertise; and (ii) such a presentation should not be open to misinterpretation. We assert that a well-founded diagrammatic representation can often support such an understanding. In addition, to assist in the support of current enginerering practice, we are

*email: corin@cogsci.ed.ac.uk

examining the potential for formal methods to ensure consistency of a design specification which may be presented to technical and non-technical users in a variety of representations.

We commonly see multiple representations being used in the specification of a particular system, with differing representations being chosen to emphasise differing *aspects* of the system under consideration. For example, hardware designers mainly use three graphical representations when describing a piece of hardware: circuit diagrams, state diagrams and timing diagrams. In this way several different views may be provided for a given system. Each view will be chosen for its merit in displaying a particular aspect of the system under consideration in a manner which emphasises all pertinent details while abstracting away from all irrelevant details. A key issue in any specification which uses multiple representations is that that information expressed should be consistent across all representations.

In this paper we present an argument for the use of formal methods to provide a strong foundation for a multiple-representation description. Using a formal semantics to provide a "core" representation we may test the consistency of a presentation, potentially with automated verification tools. In this manner we not only provide formal support for current design and specification practice, by supporting the representations currently used, we also remove the problems of the perceived intractability of formal methods by hiding all irrelevant details of the complete model from a technician with particular concern for only certain aspects of the specified system.

To illustrate the above approach we present an example taken from industry in which a formal model of a safety-critical multiprocessing system was derived from its natural language specification. To elucidate the formal model we present a variety of differing, but consistent, representations of it. To demonstrate the usefulness of these differing representations we show how each is particularly useful at highlighting particular aspects, and thus certain potential errors, of the modelled system.

2 Introduction to Example and Formal Notations

Concurrent systems are widely used in safety-critical applications. Concurrent systems are systems which require communication between asynchronously operating agents, such as processors in a multiprocessing system. Unfortunately the design of such systems is made more difficult by the complexity arising from interactions (including potentially undesirable interactions) between the various concurrent agents. For a human reasoner the asynchronous nature of concurrent systems appears to make intuitive models of such systems particularly problematical. We comment that much human reasoning about concurrent systems appears to make strong assumptions about the synchronisation of various events and actions, presumably to make an intuitive modelling of such a system tractable. Unfortunately, as experience of errors in the real world tends to indicate, these assumptions are not well-founded and may lead to assumptions about the correctness or safety of a system which are misplaced.

2.1 An (Informal) Specification of A Multiprocessing System Fragment

One type of data collection system used in safety critical applications generally consists of two components:

1. A Data Acquisition Function. This receives data asynchronously, usually from two or more sources.

2. A Monitoring/Control Function. This function periodically reads a recent data item from each source and performs some calculation or comparison.

The system considered here accepts two parallel streams of data. Each stream is allocated a three-place ring-buffer of the type illustrated by Figure 1 (more places could be used but three is sufficient for this example). When a buffer receives an item of data it is stored in the buffer place currently indicated by the `Read/Write` pointer and this pointer is then moved forward one place. If the pointer is pointing to the last free place (`b2`) when an item of data arrives then the buffer is taken to be full and a "trip" occurs due to a data overflow error.

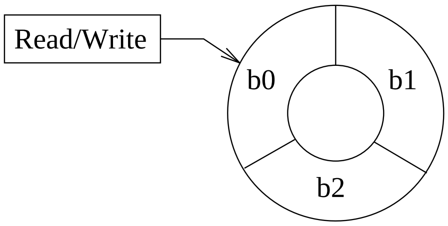

Figure 1: A Three-Place Ring-Buffer

The system may receive an instruction to vote, in which case the item of data pointed to by each buffer's `Read/Write` pointer is read, analysed and each buffer's pointer is then reset to the initial place (`b0`). If a vote request is received when either buffer's pointer points to the initial place then a trip is initiated due to an underflow error as no new data has been received since the last vote.

2.2 An Overview of CCS

A Calculus of Communicating Systems (CCS) was proposed by Milner [2] to provide a theory for communicating systems in terms of a few primitive ideas. An overview of CCS is given in [5] and we provide here a brief description which is sufficient to cover the worked example of the following section.

In CCS an agent, denoted by an upper case name, may communicate with other agents via input and output actions. Actions are denoted by lower case names, output actions are denoted by overbar symbols over names and sequences of actions are connected by the '.' operator. For example, the formula:

$C \stackrel{\mathrm{def}}{=} in.\overline{out}.C$

denotes an agent, C, which continually responds to an in action by outputting an out action.

Agents may be combined via two binary operators $+$ and $|$. The $+$ operator is a 'choice' operator. The agent $P+Q$ behaves either like P or like Q; as soon as one performs its first action the other is discarded. The $|$ operator is a 'composition' operator. Intuitively, the agent $P|\,Q$ is a system in which P and Q may proceed independently and may also interact. This interaction produces a 'silent' (or τ-) action which is a communication between complementary input and output actions such as a and $\overline{a}$.

We emphasise here that the τ-actions described above define the mechanism for communication between concurrent agents in CCS. This point will become important later, when we relate CCS and statecharts.

Agents may be defined as being instances of other agents via a substitution $[a_1/b_1, \ldots, a_n/b_n]$, which indicates that the actions $b_1, \ldots, b_n$ are replaced by the actions $a_1, \ldots, a_n$ respectively. For example, the agent A, with the definition:

$A \stackrel{\text{def}}{=} zap.\overline{ouch}.A$

could equivalently be described by the definition:

$A \stackrel{\text{def}}{=} C[zap/in, ouch/out]$

where the agent C is defined as above.

Finally we introduce the restriction operator \. This operator is used to internalise certain actions in a system. For example, the system $(A|C)\backslash\{out, ouch\}$ will only respond to the external actions in and zap, as the other actions in this system have been internalised.

As a larger example, Figure 2 illustrates a possible CCS specification of a twin-buffer system represented by the agent *Twin*. In this specification the two agents *BufferA* and *BufferB* are instances

$$
\begin{array}{lcl}
Buffer & \stackrel{\text{def}}{=} & read.Buffer + write.Buffer \\
BufferA & \stackrel{\text{def}}{=} & Buffer[voteA/read, writeA/write] \\
BufferB & \stackrel{\text{def}}{=} & Buffer[voteB/read, writeB/write] \\
Voter & \stackrel{\text{def}}{=} & vote.\overline{voteA}.\overline{voteB}.\overline{data}.Voter \\
Twin & \stackrel{\text{def}}{=} & (BufferA \mid BufferB \mid Voter)\backslash\{voteA, voteB\}
\end{array}
$$

Figure 2: Simple CCS Specification of a Twin-Buffer System

of the generic buffer agent *Buffer*. Each can accept either a read message, represented as the actions *voteA* and *voteB* respectively, or a write message, represented as the actions *writeA* and *writeB* respectively. The voter agent, *Voter*, will accept a *vote* message and respond by reading data from the two buffers. This process is represented by the *Voter* agent communicating with *BufferA* and *BufferB* via the output actions $\overline{voteA}$ and $\overline{voteB}$ respectively. Having communicated with the two buffers *Voter* then outputs the message $\overline{data}$. The *Twin* system is comprised of the concurrent composition of these three agents. The restriction of *Twin* to internalise the actions *voteA* and *voteB* ensures that the two buffers in this system will only respond to vote requests from the *Voter* agent.

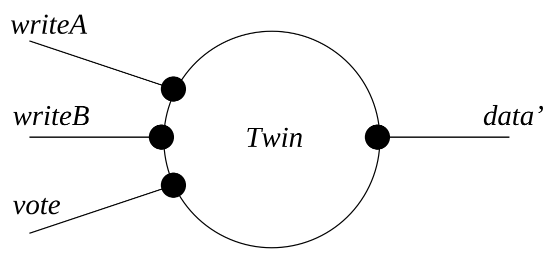

Figure 3: Top Level Flow Graph for CCS Example

Figure 3 gives the CCS *flow graph* for the *Twin* system. A flow graph such as this provides a graphical representation of the actions which a CCS system may communicate with. In this case the above flow graph shows that the *Twin* system may communicate via the actions *writeA*, *writeB*, *vote* and $\overline{data}$. The flow graph of Figure 3 gives no detail of the communication between the agents in the *Twin* system: *BufferA*, *BufferB* and *Voter*. Such a flow graph may be drawn, although we do not show it here.

2.3 An Overview of Statecharts

Statecharts provide a visual formalism for specifying the behaviour of reactive systems (that is, systems which continuously react to internal and external events). They were introduced in [1] and a semantics for them has been proposed in [4]. Statecharts resemble state transition diagrams being essentially a set of states, represented as blocks, and a set of transitions between these states, represented as labelled arcs between these blocks. However, in addition to this statecharts support a number of enhancements which permit various abstractions. One major consequence of these is that statecharts overcome the problem of an explosion of the number of states in a state diagram, provided that there is some regularity in the system being depicted. The extra features of statecharts support the modularisation of state transition diagrams and allow the representation of complex transition conditions and consequences and also the composition of sets of states, either concurrently or hierarchically.

In this paper we shall consider only a restricted subset of the syntax of statecharts which we illustrate with the statechart of Figure 4. In this syntax states are represented as labelled blocks and transitions as arrowed arcs with labels of the form $e/t_1; \ldots; t_n$, which denotes a transition triggered by the event e that generates the actions $t_1; \ldots; t_n$.

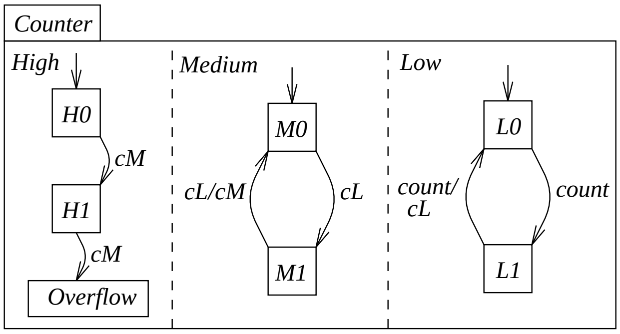

Figure 4: A 3-bit Binary Counter

Figure 4 gives the statechart for a 3-bit binary counter represented by the outer state *Counter*. This system is comprised of three inner states *High*, *Medium* and *Low* which represent the three bits of the counter. The fact that these three states are separated by dotted lines indicates that these are AND-states and *Counter* will be in all of these states concurrently.

The three states *High*, *Medium* and *Low* themselves have inner states which are all OR-states. This means that the state *Low*, for example, will either be in state *L0* or state *L1*, but not both. Each bit in the counter may be either in a ‘0’ state or a ‘1’ state (or the *Overflow* state, in the case of *High*). Initially each will be in the ‘0’ state. This initialisation is represented by the arc pointing to the relevant ‘0’ state (*H0*, *M0* and *L0*) whose tail is anchored in free space rather than a labelled block.

Counting is triggered by the external event *count*. Whenever the event *count* occurs the state *Low* responds by switching between its two internal states. If it moves from *L1* to *L0* it also generates the action *cL*. This generated action is taken as an event by the *Medium* state, triggering a transition between its two internal states. Similarly the transition in state *Medium* between *M1* and *M0* will generate an action triggering a transition in state *High*. Once the counter reaches its maximum value (that is, each inner state in the '1' state) the next *count* event will trigger a set of transitions that will ultimately move *High* into the *Overflow* state.

We emphasise here that the use of generated actions, such as the *cL* action generated by the *count/cL* transition above, as events which trigger other transitions defines the mechanism for communication between concurrently composed states in statecharts. This point will become important later, when we relate CCS and statecharts.

2.4 Representing CCS With Statecharts

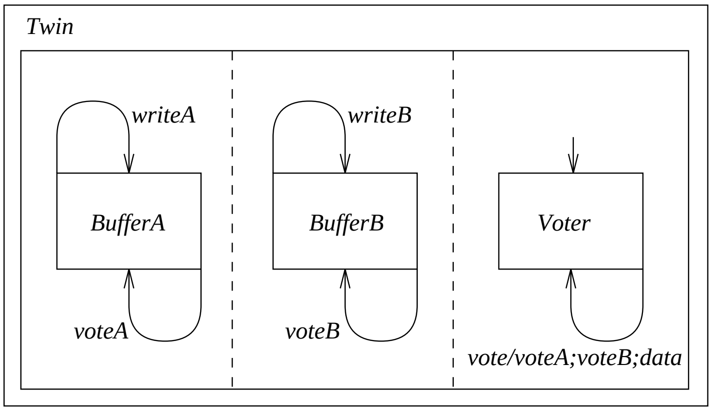

Figure 5: Statechart of the Simple Twin Buffer Specification

Figure 5 is a statechart constructed from the CCS specification in Figure 2. In this diagram the agents such as *BufferA*, *BufferB* and *Voter* are represented by states with these names and actions are represented as transition arcs between states. We have chosen to model input actions in CCS as *events* in statecharts and to model output actions in CCS as *actions* in statecharts (note that in CCS input and output actions are symmetric, so the distinction we make here is artificial). Thus the sequence of CCS actions $vote.\overline{voteA}.\overline{voteB}.\overline{data}$, defined as a part of the *Voter* agent, have been represented by the transition *vote/voteA;voteB;data* in Figure 5.

The manner in which we have modelled the CCS specification of Figure 5 needs to be justified and this justification is as follows. In modelling CCS actions as statechart transitions we state that we operated firstly under the constraint of equating the concurrent communication mechanism of CCS (corresponding input and output actions) with that of statecharts (corresponding generated actions and events). Secondly we wished to define our statechart in a manner which utilised the expressive power of statecharts in a manner which captured a designer's intended interpretation of CCS.

Note that here we are making a distinction between the intended interpretation of a model and its actual semantics. Ideally we would hope that the two were equivalent but, as we show in Section 3.4, this may not always be the case.

3 The Twin-Buffer System

An exercise in the specification and modelling of a safety-critical multiprocessing system fragment was undertaken by an industrial group. A natural language specification was subjected to peer review and then formally modelled in CCS. The designers examination of this formal model uncovered a potential behaviour of the system which they had considered impossible. A further analysis by the designers showed that they had made certain assumptions concerning the relative timings of concurrent agents within the system and that these assumptions were not explicitly recorded, were not necessarily justifiable and had not been identified by peer review. From reports documenting the above exercise, provided by our industrial partner, we have performed an analysis of this communication failure which is currently awaiting commercial clearance.

In this section we present aspects of the formal model derived from the natural language specification. We then proceed to illustrate the use of differing representations of this model by showing three errors which were uncovered by analysis.

3.1 The Twin-Buffer Specification

To simplify the formal modelling of the twin-buffer system the actual contents of any data item held by the buffers were not modelled. Thus while read and vote instructions were modelled, the data transferred by such instructions was abstracted out of the model.

We do not show here the formal (CCS) specification for the twin-buffer system. Instead we present initially the statechart and flow graph associated with this specification. Together these two graphical representations present in (we argue) a clear manner two errors and one key mistaken assumption associated with the formal specification. This specification may be corrected by manipulation of the statechart and flow graph, with the specification being automatically updated to ensure consistency of interpretation.

Figure 6 is the statechart constructed from the formal specification. In this statechart we have

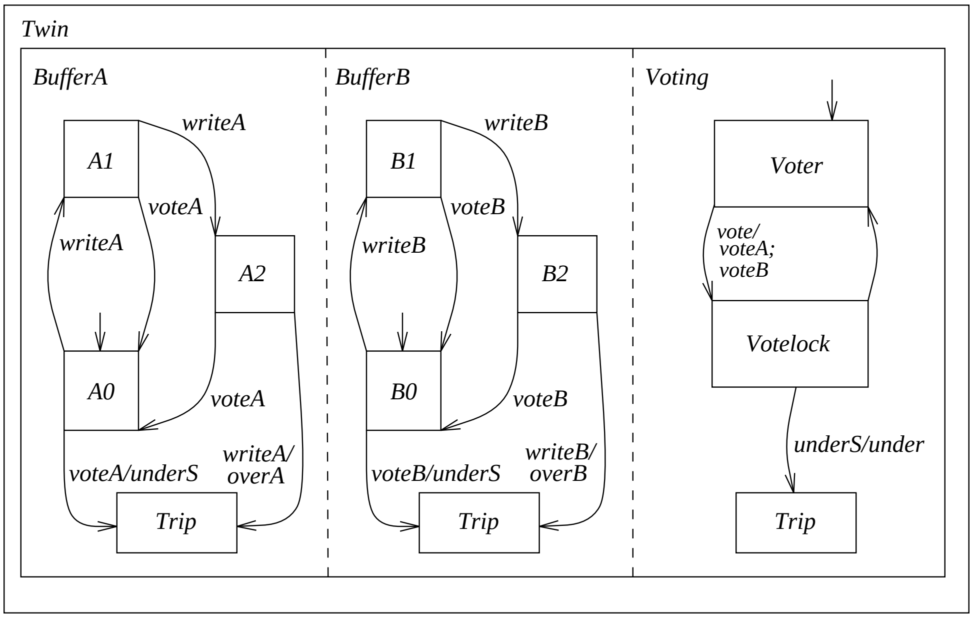

Figure 6: Statechart of the CCS Specification

three main states *BufferA*, *BufferB* and *Voting* which correspond to the three states *BufferA*, *BufferB* and *Voter* in the simpler statechart of Figure 5.

The extra detail in the above statechart reflects the issues which the designers wished to model. These were that the buffers used were three-place ring buffers and that overflow and underflow errors could be modelled. To model these errors the buffers were provided with some measure of "intelligence", in that they would be able to detect for themselves whether they had overflowed or underflowed (a subsequent, more detailed, model was made in which such error states were detected and controlled solely by the voting process).

Each buffer in the specification has three valid states (*A0*, *A1* and *A2*) in the case of *BufferA* and an error state *Trip*. The three valid states correspond to the three buffer places `b0`, `b1` and `b2` in the ring-buffer of Figure 1. The *Trip* state is an error state which will be entered if the buffer either overflows or underflows and the buffer will output an appropriate action ($\overline{overA}$ or $\overline{underS}$ for *BufferA*) if this happens.

The *Voting* state consists of a *Voter* agent which processes *vote* requests and a *Votelock* agent which is intended to inhibit *Voting* from accepting *vote* requests if either buffer has underflowed. If either buffer underflows then it is intended that *Voting* will also enter a *Trip* state.

Three errors may be found in this specification. The first two of these errors were uncovered during our analysis of the formal specification and became apparent when this specification was represented in two differing graphical manners, firstly a flow graph, which we show below, and secondly the statechart above. The third error was identified by the designers during animation of the CCS specification and is due to a mistaken assumption made by them concerning the relative timing of concurrent agents within the system. In Section 3.4 we show how the above statechart may be used to give a formal and readily understandable explanation for this assumption.

3.2 The "No Data" Error

In the formal description of the twin-buffer system we noted that the specification had made several abstractions with respect to the data stored in a ring-buffer. These abstractions had generalised the system so that data was not represented at all, the only aspect of the data which was represented being that *write* and *vote* actions could occur. This led to a peculiarity which becomes obvious when we consider the top-level flow graph for the specification.

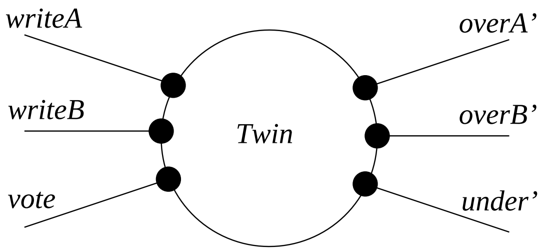

Figure 7: Top Level Flow Graph for CCS Specification

Figure 7 gives the flow graph for the formal specification. It is immediately apparent from this flow graph that, while underflow and overflow error messages are output by the system, there is no communication of data output by the *Twin* system.

This feature of "no data" is not necessarily an error in the specification, as the designers may have intended to make this abstraction in their formal model. However, the absence of any modelling of data has an affect on the modelling of the *Votelock* agent which can be identified as an error.

3.3 The *Votelock* Error

One point of note in the statechart of Figure 6 is that one of the two arcs between the *Voter* and *Votelock* agents has no label. This peculiarity led us to uncover an error in the formal specification.

The *Votelock* agent was designed to "provide a mechanism for deadlocking the *Voter* process when either of the buffers deadlocked" (in the natural language specification it is stated only that this happens for an underflow condition). The intention being that, should a buffer deadlock, an $\overline{underS}$ output action would indicate this and the agent *Votelock* would respond by transferring to the deadlock state *Trip*, incidentally outputting an error message, $\overline{under}$, along the way. However, while the specification models the effect of a failed *write* or *vote* event (that is, an event causing a buffer to deadlock) it does not explicitly model the effect of a successful *write* or, more importantly, *vote* event. Regardless of whether an $\overline{underS}$ action has occurred, it would still be legal for the *Votelock* agent to make the transition to the *Voter* agent. This is a serious problem with the specification as it can lead to an unintended deadlock situation.

Let us describe a state of the statechart in Figure 6 as the set of innermost concurrent states. Thus the initial state of this statechart would be described as $\{A0, Voter, B0\}$. Consider this initial state and the sequence of actions *vote.vote*. A legal sequence of transitions is:

1. $\{vote\}$: triggers a transition in *Voting* which takes the initial state to $\{A0,(/voteA;voteB),B0\}$. Here the term *(/voteA;voteB)* indicates that *Voting* is suspended on the arc *vote/voteA;voteB*, waiting to 'clear' the generated actions *voteA* and *voteB*. This is effected by the two silent τ-actions.

2. $\{\tau\}$: 'clears' the first suspended action in *Voting*. *voteA* is taken in *BufferA* leading to the state $\{(/underS),(/voteB),B0\}$.

3. $\{\tau\}$: 'clears' the second suspended action in *Voting*. *voteB* is taken in *BufferB* leading to the state $\{(/underS), Votelock,(/underS)\}$.

4. $\{vote\}$: may trigger a legal transition in *Voting* which then, instead of communicating with a suspended *underS* action, leads to the state $\{(/underS),(/voteA;voteB),(/underS)\}$. At this point each of the three concurrent states is suspended pending communication with one of the other states. No more actions can be taken and the system has therefore deadlocked.

Note that the above deadlock is a serious error which should not be confused with a deadlock caused by the system entering the *Trip* state. The system is designed to enter the *Trip* state only if an overflow or underflow error has occurred and will always have first broadcast an error message. The deadlock above is potentially far more serious for a safety-critical system as it describes a situation not intended by the designers where the twin-buffer system has deadlocked without broadcasting any form of error message. However, some consolation may be drawn from the fact that this error is a feature of the process by which the natural language specification was formalised, rather than any identifiable error in that specification.

This error has arisen from an *over-abstraction* in the construction of the formal CCS model of the specification. A deliberate decision was made to abstract from a representation of any actual data in the twin-buffer system while maintaining a representation of the situation where the system *failed* to manipulate the data. Unfortunately the CCS designer has abstracted too far by failing to capture any explicit representation of the system *succeeding* in manipulating the data. We may explain the identification of this error as being the discovery of an *under-specification*, which was that in the statechart an arc existed which did not have an associated transition label.

3.4 Identification of an Incorrect Assumption

The designers of the full natural language specification of the twin-buffer system specification suggested eight potentially safety-critical properties which it was desirable to demonstrate held of the system. The last of these properties was that "failure of either input process or the voting process will result in the correct trip position". The above specification of this system was subsequently developed and both the author of the natural language specification and the designer of the CCS specification agreed on the following behaviour of the system:

> Each buffer can receive a *write* instruction directly, but can only receive a *vote* action from the *Voter* agent. Thus we can deduce that either buffer will trip if it performs three *write* actions without the voter performing an intervening *vote* action.

This behaviour was assumed to be a consequence of the above safety-critical property. However, animation of the above specification by an automated verification tool, the Concurrency Workbench [3], showed the following as a legal sequence of actions of the agent *Buffer_Spec*:

vote.writeB.writeA.writeA.writeA.writeA

This sequence of actions was surprising for two reasons:

- The sequence begins with a *vote* action which the designers believed should cause an immediate trip as the buffers are initially empty.
- There are four consecutive *writeA* actions. The designers believed that this should cause a trip.

The legality of the above sequence of actions becomes obvious when one considers the silent τ-action caused by the synchronisation of the actions *voteA* and $\overline{voteA}$. This τ-action could occur after either the first or second *writeA* action in the above sequence, as follows:

1. {*vote*}: triggers a transition in *Voting* which takes the initial state to {*A0,(/voteA;voteB),B0*}.
2. {*writeB*}: triggers a transition in *BufferB* leading to the state {*A0,(/voteA;voteB),B1*}.
3. {*writeA*}: triggers a transition in *BufferA* leading to the state {*A1,(/voteA;voteB),B1*}.
   We may 'clear' the first suspended action in *Voting*, *voteA*, either now or after the next *writeA* event. For the sake of simplicity we shall assume that this action and the second suspended action, *voteB*, are cleared after the following *writeA* action.
4. {*writeA*}: triggers a transition in *BufferA* leading to the state {*A2,(/voteA;voteB),B1*}. The two suspended actions may now be cleared. This is effected by the two silent τ-actions:
   - *voteA* is taken in *BufferA* leading to the state {*A0,(/voteB),B1*}.
   - *voteB* is taken in *BufferB* leading to the state {*A0,Voter,B0*}.
5. {*writeA*}: triggers a transition in *BufferA* leading to the state {*A1,Voter,B0*}.
6. {*writeA*}: triggers a transition in *BufferA* leading to the final (non-trip) state {*A2,Voter,B0*}.

What is of interest to us here is that after examining both the formal specification and the natural language specification in more detail, both designers agreed that this sequence of actions was correct with respect to both specifications and that it was the designers assumption that three consecutive *write* actions with no intervening *vote* action would cause a trip which was incorrect. After review it was decided that the mistaken assumption on the part of the designers was that the time taken for a *vote* action to both initiate and receive a response from the $\overline{voteA}$ and $\overline{voteB}$ actions would be sufficiently small that no intervening *write* action could be performed.

One advantage of using the statechart of Figure 6 to represent the formal specification is that it provides us with a readily understandable formal account for the above missed assumption. The designers assumption can be explained as being a belief that an action, such as *vote*, and all of the actions associated with it (that is, $\overline{voteA}$ and $\overline{voteB}$) would be processed sufficiently quickly that no intervening actions could occur. Examining the statechart of Figure 6 we see that this assumption corresponds precisely to the assumption that transition along an arc will happen sufficiently quickly that no intervening transitions can occur. In the semantics of statecharts [4] this assumption is formally expressed and referred to as the *Synchrony Hypothesis*.

We may manipulate this formalisation of the designers missed assumption to describe the unexpected behaviour they observed. By suggesting that transitions may intervene between the various actions in a single transition (for example, between the three actions in the transition *vote/voteA;voteB*) we are able to clearly indicate how and where the designers missed assumption can be violated.

4 Correcting the Specification

Having identified the "no data" and *Votelock* errors of the previous section we attempted to correct the specification by manipulation of the statechart of Figure 6. After an initial revision of the statechart which, when animated, also deadlocked in a similar and unintended manner we constructed the corrected statechart of Figure 8. The method of construction of both of these revised statecharts

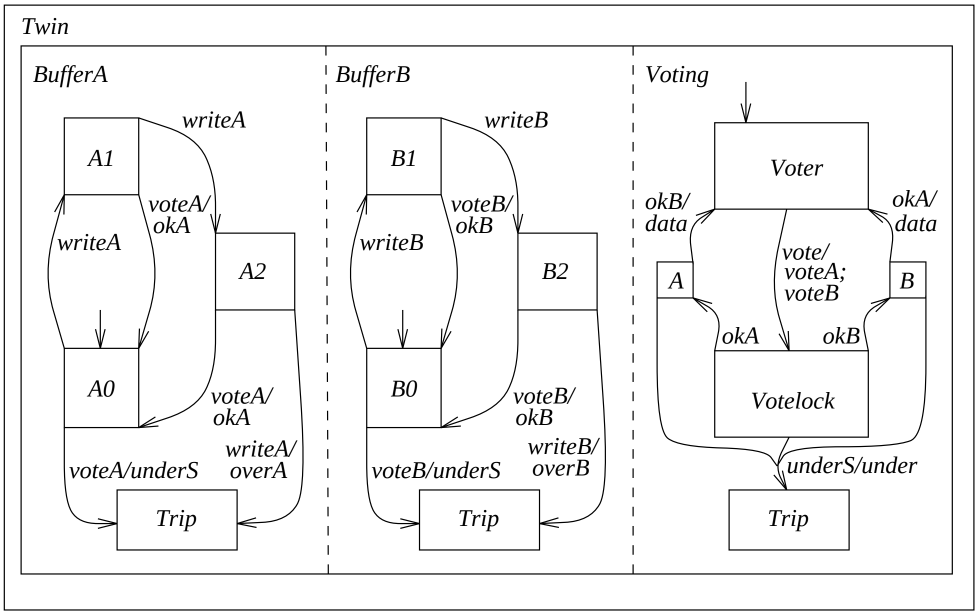

Figure 8: Statechart of Corrected CCS Specification

was to add transitions and any necessary extra states which allowed the unlabelled arc between the *Voter* and *Votelock* agents to have an associated transition. Two new actions were introduced, *okA* and *okB*, which represented data read from buffers A and B respectively (for good measure we added two sets of transitions, indicating that the data could be received from A and B in any order). From this corrected statechart we have reconstructed a corresponding CCS specification.

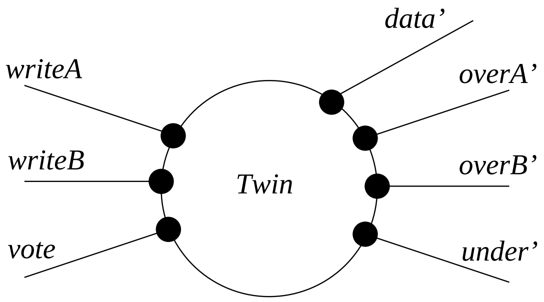

Figure 9: Top Level Flow Graph for Corrected CCS Specification

In addition to the corrections necessary to avoid the above deadlock we also wanted to ensure that the flow graph for the corrected CCS specification indicated that the *Twin* system could communicate via a $\overline{data}$ action. We wished to modify the specification so that it would have the top-level flow graph of Figure 9. To enable this we have appended a *data* action to the appropriate transitions within the *Voting* state, indicating that output data may be communicated by the *Twin* system once both buffers have been successfully read.

5 Conclusions

In this paper we have demonstrated with a worked example that formal methods can provide valuable assistance in the modelling of complex domains, such as concurrent systems, both by forcing many assumptions to be made explicit and by supporting the application of rigorous reasoning techniques. However, a formal model of a system must be able to be presented in a manner which elucidates the inferences which may be drawn from it. We note that such a presentation can provide invaluable support both to an expert, by making aspects of the model more immediate, and also to a non-expert, by providing a more tractable *visualisation* of the formal model.

For a visualisation of a formal model we claim that graphical representations often provide a view of a model which can be both lucid and tractable. However, it must be stressed that a meaningful semantics must be provided for such a representation so that we may verify that the graphical model, and any inferences a user may draw from it, is an accurate reflection of the formal model. These points are clearly demonstrated in this paper where we have provided graphical (flow graph and statechart) representations for a formal (CCS) specification.

We have presented an exercise in the specification and modelling of a safety-critical multiprocessing system fragment which has highlighted two errors and an incorrect assumption. Firstly, by presenting aspects of the formal model using graphical representations, we have identified two errors caused by an over-abstraction in the translation from the natural language specification to the formal model. Secondly the formal modelling has identified an important missed, and mistaken, assumption in the natural language specification.

The two errors were uncovered when using a flow graph and statechart representation of the formal specification respectively. These errors arose from an incorrect translation from the natural language specification to the formal specification and became readily apparent in the new representations. We were able to correct these errors by modifying the flow graph and statechart. By updating the formal model in line with these modifications we were able to ensure their consistency and correctness.

Following examination of the missed assumption we concluded that it was caused by an intuitive (mis-)understanding, on the part of the designers, of aspects of both the specification *and* the formal model relating to timing assumptions. Our statechart representation was originally developed to clarify these missed assumptions and we have shown how it captures the designers (mistaken) intended interpretation of the formal specification. We were thus able to give a formal description of the unrecorded assumption by showing its equivalence to the *Synchrony Hypothesis* of statechart semantics.

6 Acknowledgements

We would like to thank Stuart Anderson, Glenn Bruns, Robert Inder and Keith Stenning for their help and comments on an earlier draft of this paper. This work was supported by SERC/DTI Safety Critical Systems project GR/J58619, "Communication in Safety Cases - A Semantics Approach".

References

[1] D Harel. Statecharts: A Visual Approach to Complex Systems. Technical Report CS 84-05, Dept. of Applied Mathematics. The Weizmann Institute of Science, 1986.

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

[3] F Moller. The Edinburgh Concurrency Workbench (Version 6.0). Technical report, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[4] A Pnueli and M Shalev. What is in a Step: On the Semantics of Statecharts. In *TACS*. LNCS 526, 1992.

[5] D Walker. Introduction to a Calculus of Communicating Systems. Technical Report ECS-LFCS-87-22, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.