Applying Formal Methods to Prove Correctness of Surgical Robot Software

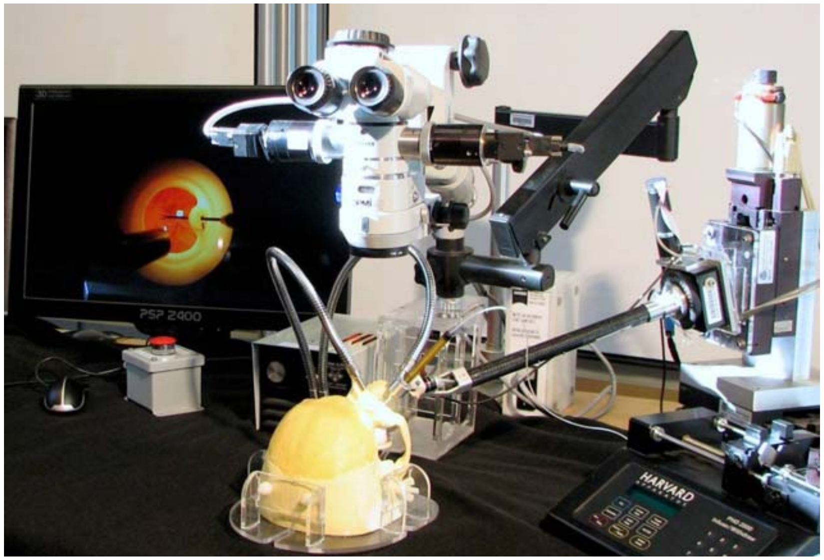
Peter Kazanzides
Computer Science
Johns Hopkins Univ.

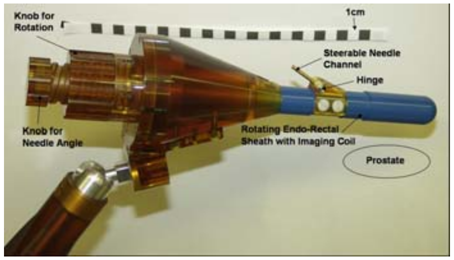
Yanni Kouskoulas
Applied Physics Lab
Johns Hopkins Univ.

Zhong Shao
Computer Science
Yale Univ.

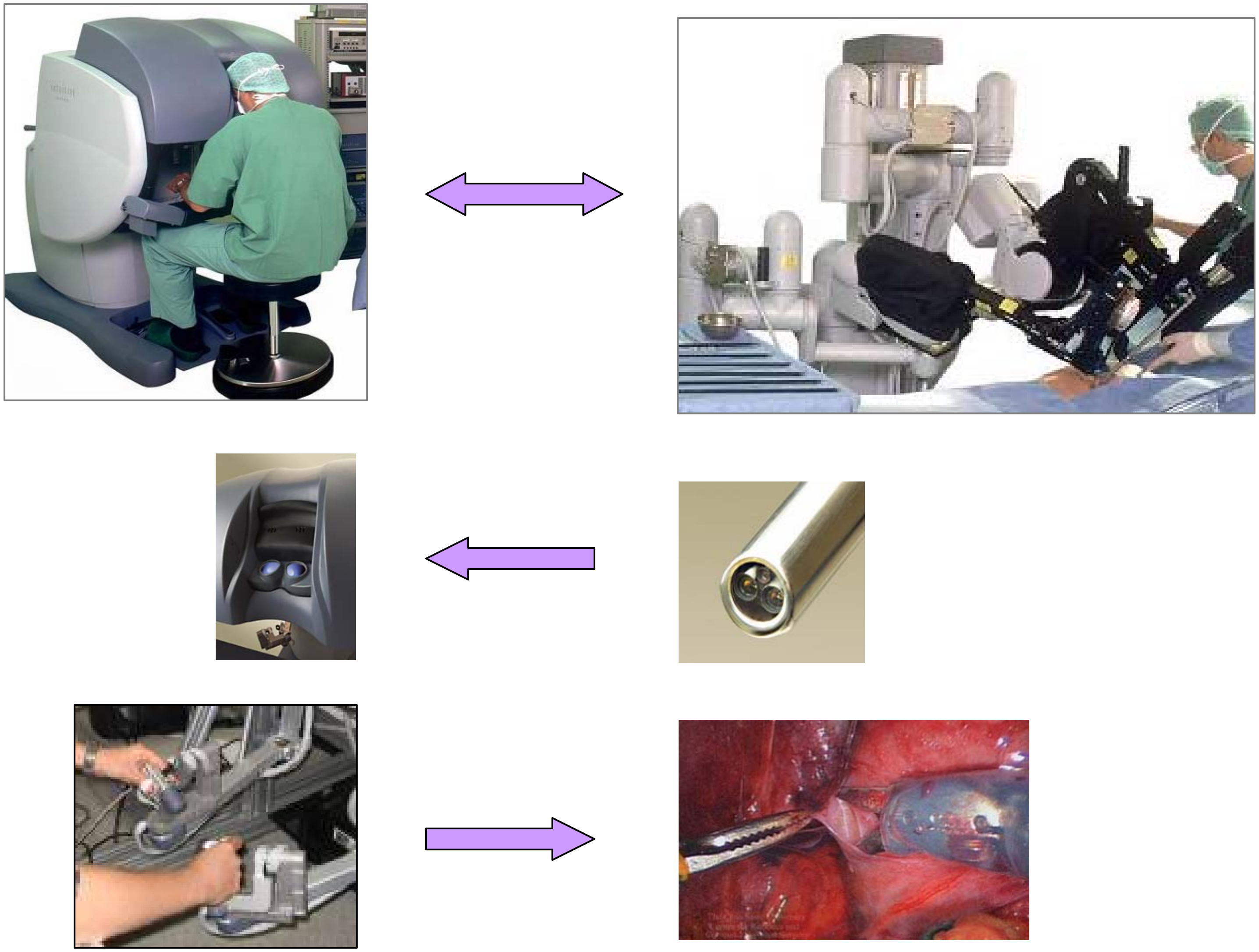
HCSS, Annapolis MD, May 4, 2011

Surgical Robotics (Cyber-Physical Systems)





da Vinci Surgical Robot



Figures courtesy of Intuitive Surgical, Inc.

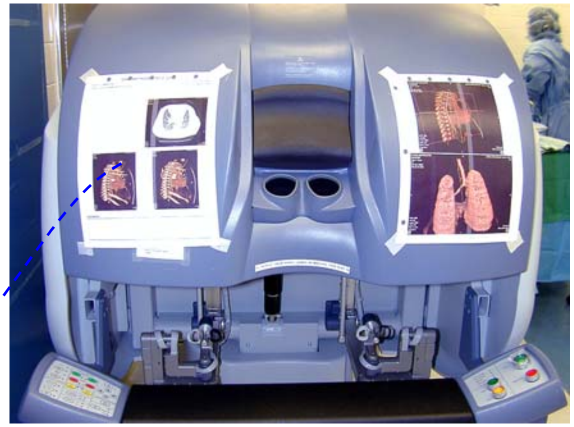
da Vinci research opportunities

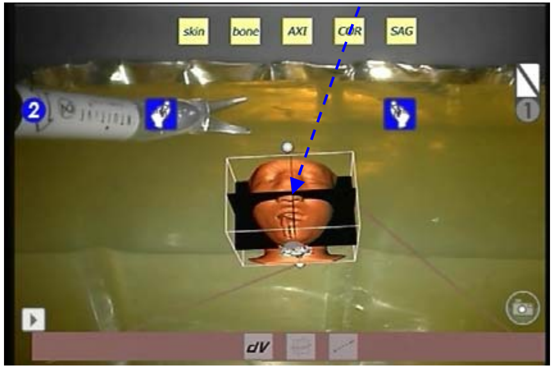
Augmented Reality

- Preoperative images
- Intraoperative data

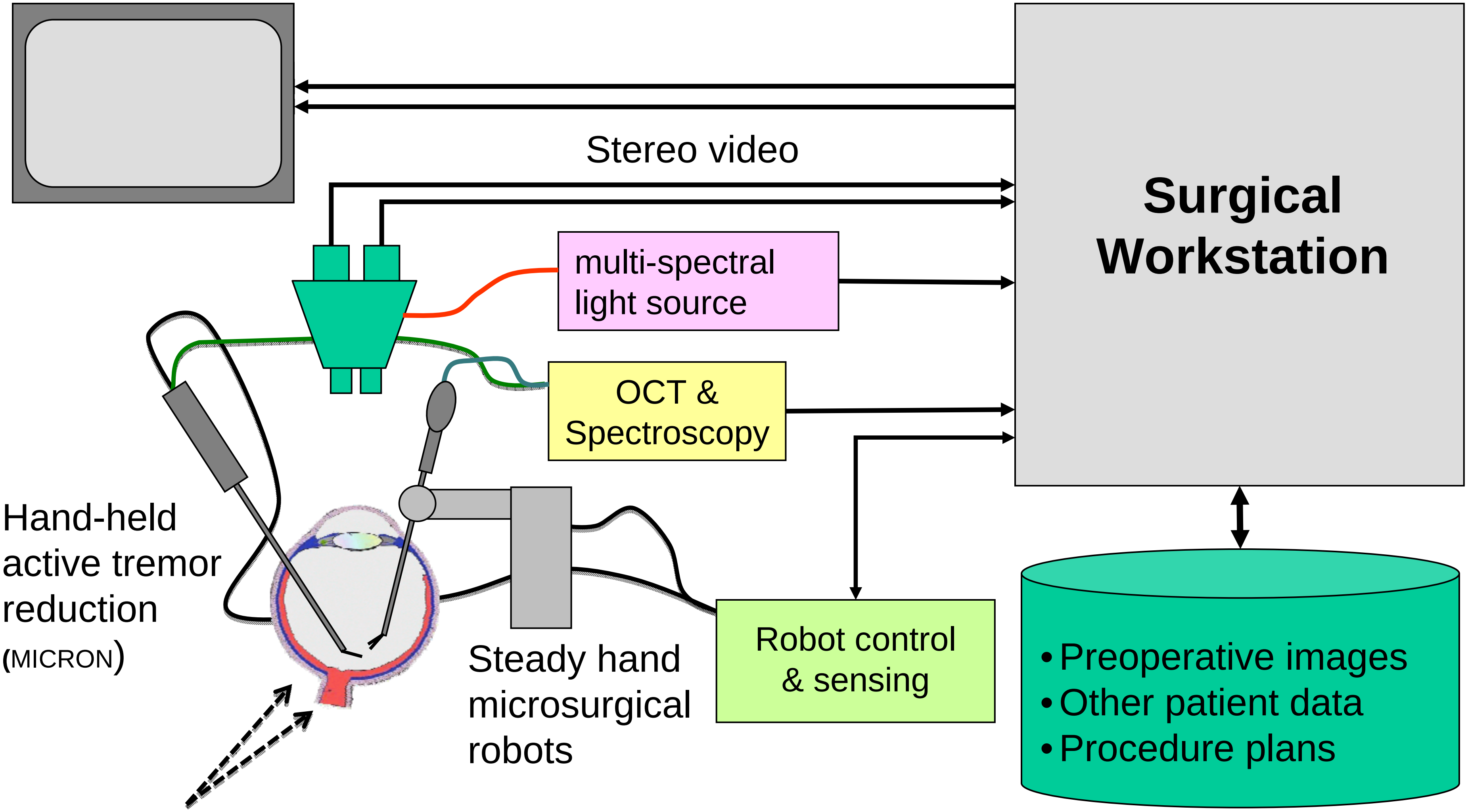
Mechanical Assistance

- Virtual fixtures
- Motion primitives





Retinal Microsurgery System



Stereo video

Surgical Workstation

multi-spectral light source

OCT & Spectroscopy

Hand-held active tremor reduction (MICRON)

Steady hand microsurgical robots

Robot control & sensing

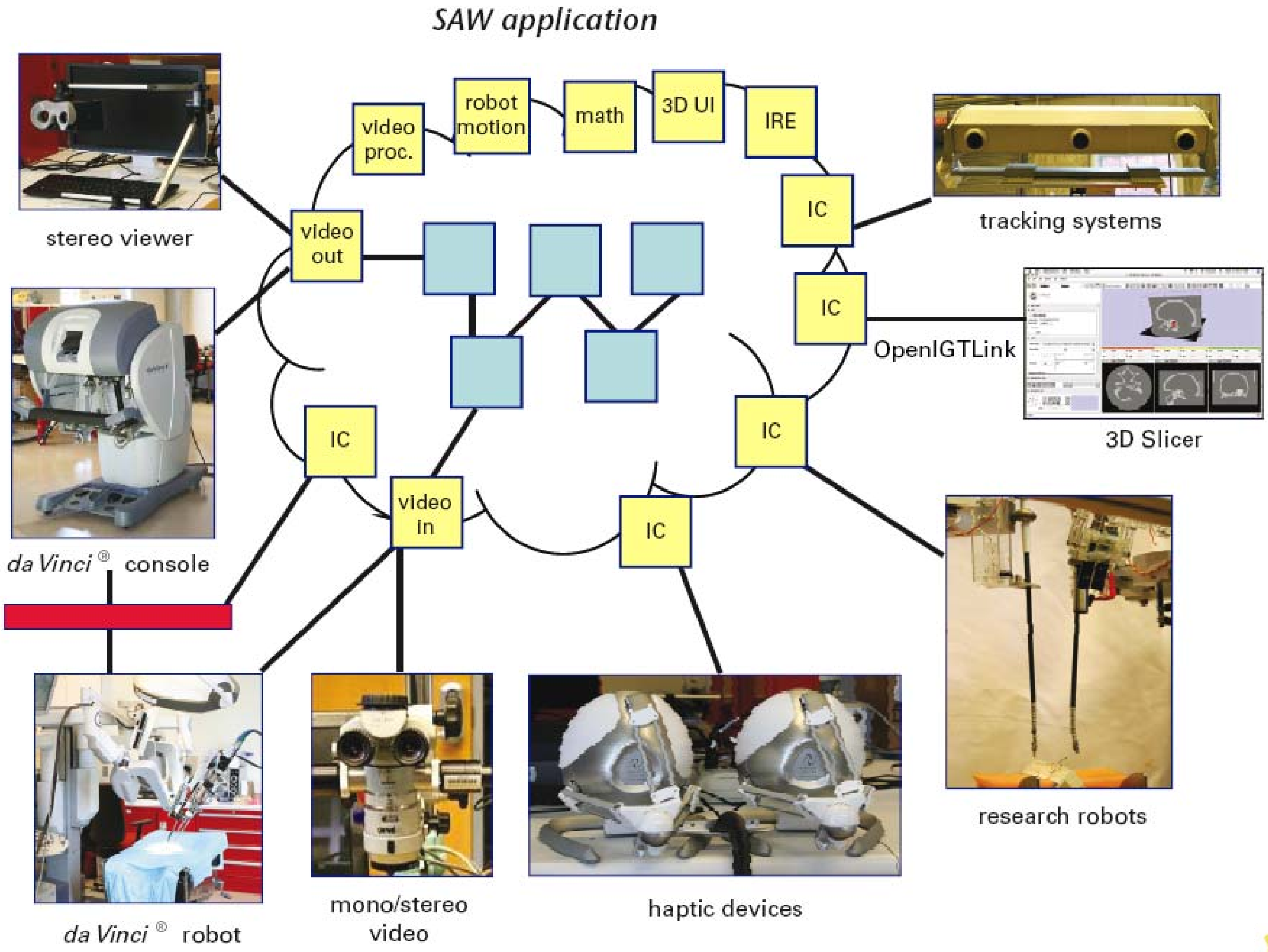
- Preoperative images
- Other patient data
- Procedure plans

20-25 gauge tools & sensors (proximity, force, ischemia, OCT)

NIH EB 007969

Credit: Russell Taylor

Surgical Assistant Workstation (SAW)

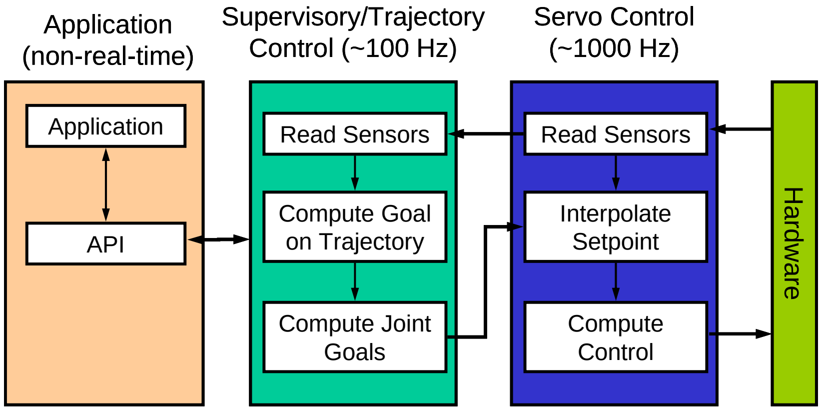


Joint development with Intuitive Surgical, Inc.

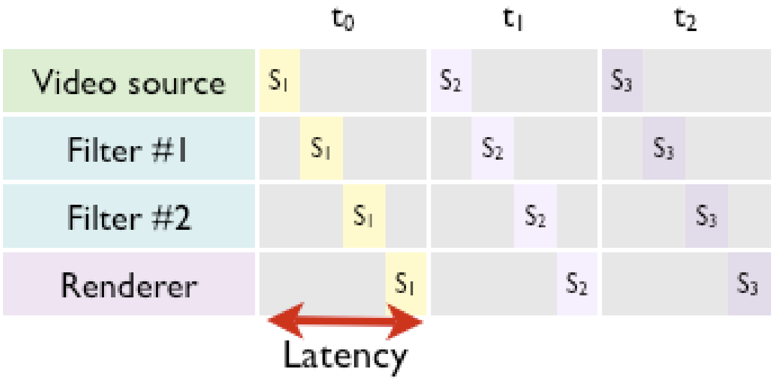
NSF EEC 9731748, EEC 0646678, MRI 0722943

SAW Requirements

- Support robot control and real-time image processing
- Concurrent execution within a process (multi-threading) and between processes
- Plug & play of devices
- Safety for clinical testing



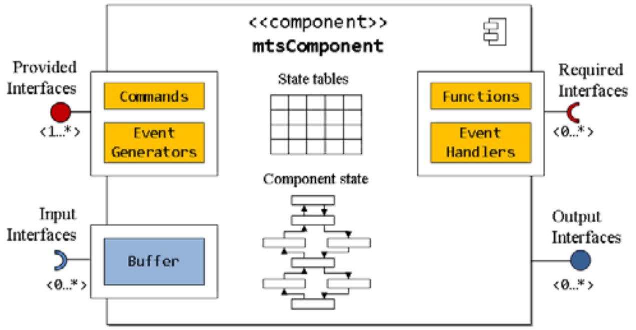
Hierarchical multi-rate robot control



Real-time video stream

SAW Design

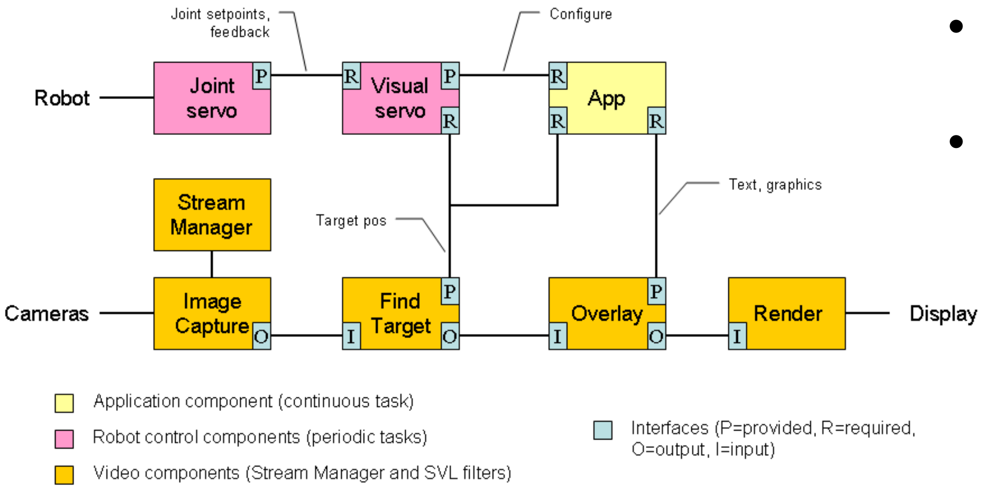
- Component-based software architecture
  - Within process and between processes
  - Uses *cisst* C++ libraries



ICE for data exchange between components in different processes

Efficient, lock-free data exchange between components in same process

- State table (single writer, multiple readers)
- Mailboxes (single reader/writer FIFO)



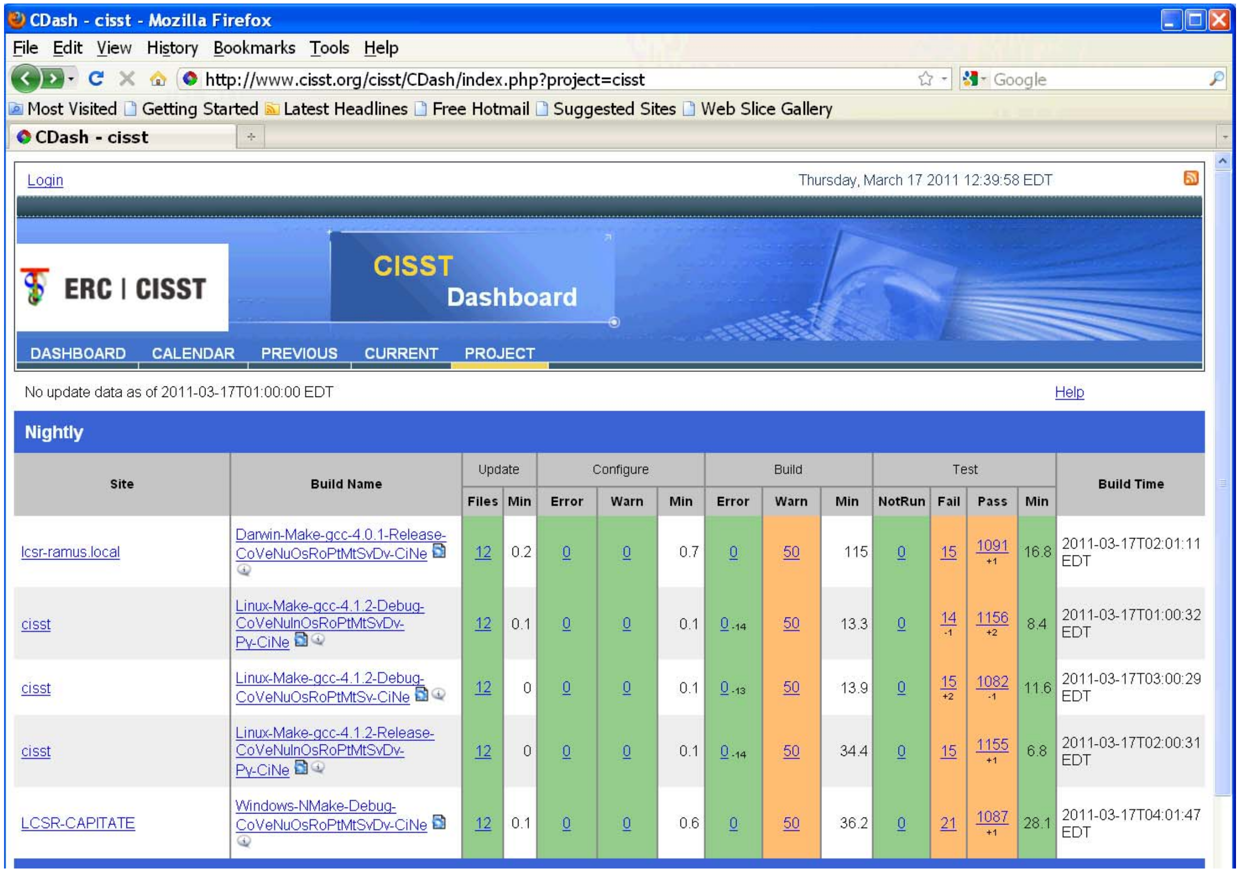
Medical Device Safety

- Medical device safety: IEC 60601
- Medical device SW life cycle: IEC 62304
- Risk management: ISO 14971
  - Failure Modes Effects and Criticality Analysis (FMECA), IEC 60812

Focus on process, traceability, and testing

SAW Testing

Automated unit testing framework (uses CDash)



Thursday, March 17 2011 12:39:58 EDT

No update data as of 2011-03-17T01:00:00 EDT

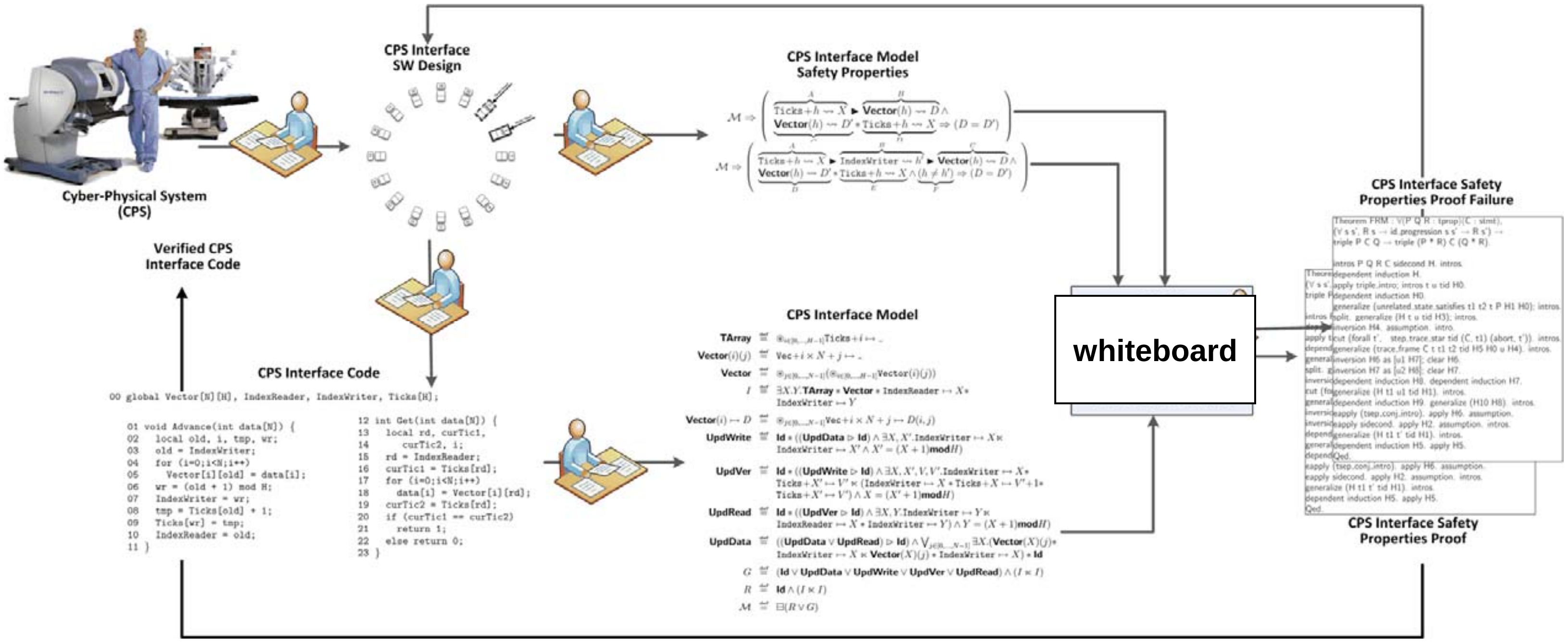
Nightly

Site	Build Name	Update		Configure			Build			Test				Build Time
		Files	Min	Error	Warn	Min	Error	Warn	Min	NotRun	Fail	Pass	Min	
lcsr-ramus.local	Darwin-Make-gcc-4.0.1-Release-CoVeNuOsRoPtMtSvDv-CiNe	12	0.2	0	0	0.7	0	50	115	0	15	1091 +1	16.8	2011-03-17T02:01:11 EDT
cisst	Linux-Make-gcc-4.1.2-Debug-CoVeNuInOsRoPtMtSvDv-Py-CiNe	12	0.1	0	0	0.1	0 -14	50	13.3	0	14 -1	1156 +2	8.4	2011-03-17T01:00:32 EDT
cisst	Linux-Make-gcc-4.1.2-Debug-CoVeNuOsRoPtMtSv-CiNe	12	0	0	0	0.1	0 -13	50	13.9	0	15 +2	1082 -1	11.6	2011-03-17T03:00:29 EDT
cisst	Linux-Make-gcc-4.1.2-Release-CoVeNuInOsRoPtMtSvDv-Py-CiNe	12	0	0	0	0.1	0 -14	50	34.4	0	15	1155 +1	6.8	2011-03-17T02:00:31 EDT
LCSR-CAPITATE	Windows-NMake-Debug-CoVeNuOsRoPtMtSvDv-CiNe	12	0.1	0	0	0.6	0	50	36.2	0	21	1087 +1	28.1	2011-03-17T04:01:47 EDT

What about lock-free mechanisms for data exchange between concurrent threads?

Formal Methods to the Rescue!

- Use FM to validate a small subset of SAW (data exchange primitives)
  - Most difficult to validate by testing
  - Incremental introduction of FM



CPS Interface Model Safety Properties

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{B} \wedge \\ \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{C} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{D} \Rightarrow (D = D') \end{array} \right)$$

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\texttt{IndexWriter} \rightsquigarrow h'}^{B} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{C} \wedge \\ \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{D} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{E} \wedge \underbrace{(h \neq h')}_{F} \Rightarrow (D = D') \end{array} \right)$$

CPS Interface Code

```
00 global Vector[N][H], IndexReader, IndexWriter, Ticks[H];

void Advance(int data[N]) {
  local old, i, tmp, wr;
  old = IndexWriter;
  for (i=0;i<N;i++)
    Vector[i][old] = data[i];
  wr = (old + 1) mod H;
  IndexWriter = wr;
  tmp = Ticks[old] + 1;
  Ticks[wr] = tmp;
  IndexReader = old;
}

int Get(int data[N]) {
  local rd, curTic1,
     curTic2, i;
  rd = IndexReader;
  curTic1 = Ticks[rd];
  for (i=0;i<N;i++)
    data[i] = Vector[i][rd];
  curTic2 = Ticks[rd];
  if (curTic1 == curTic2)
    return 1;
  else return 0;
}
```

CPS Interface Model

$$\begin{array}{rcl}
\textbf{TArray} & \stackrel{\text{def}}{=} & \circledast_{i\in[0,\dots,H-1]}\texttt{Ticks}+i \mapsto \_ \\
\textbf{Vector}(i)(j) & \stackrel{\text{def}}{=} & \texttt{Vec}+i\times N+j \mapsto \_ \\
\textbf{Vector} & \stackrel{\text{def}}{=} & \circledast_{j\in[0,\dots,N-1]}(\circledast_{i\in[0,\dots,H-1]}\texttt{Vector}(i)(j)) \\
I & \stackrel{\text{def}}{=} & \exists X, Y.\textbf{TArray} * \textbf{Vector} * \texttt{IndexReader} \mapsto X * \\
 & & \texttt{IndexWriter} \mapsto Y \\
\textbf{Vector}(i) \mapsto D & \stackrel{\text{def}}{=} & \circledast_{j\in[0,\dots,N-1]}\texttt{Vec}+i\times N+j \mapsto D(i,j) \\
\textbf{UpdWrite} & \stackrel{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdData} \rhd \textbf{Id}) \wedge \exists X, X'.\texttt{IndexWriter} \mapsto X \ltimes \\
 & & \texttt{IndexWriter} \mapsto X' \wedge X' = (X+1)\textbf{mod}H) \\
\textbf{UpdVer} & \stackrel{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdWrite} \rhd \textbf{Id}) \wedge \exists X, X', V, V'.\texttt{IndexWriter} \mapsto X * \\
 & & \texttt{Ticks}+X' \mapsto V' \ltimes (\texttt{IndexWriter} \mapsto X * \texttt{Ticks}+X \mapsto V'+1 * \\
 & & \texttt{Ticks}+X' \mapsto V') \wedge X = (X'+1)\textbf{mod}H) \\
\textbf{UpdRead} & \stackrel{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdVer} \rhd \textbf{Id}) \wedge \exists X, Y.\texttt{IndexWriter} \mapsto Y \ltimes \\
 & & \texttt{IndexReader} \mapsto X * \texttt{IndexWriter} \mapsto Y) \wedge Y = (X+1)\textbf{mod}H) \\
\textbf{UpdData} & \stackrel{\text{def}}{=} & ((\textbf{UpdData} \vee \textbf{UpdRead}) \rhd \textbf{Id}) \wedge \bigvee_{j\in[0,\dots,N-1]} \exists X.(\textbf{Vector}(X)(j) * \\
 & & \texttt{IndexWriter} \mapsto X \ltimes \textbf{Vector}(X)(j) * \texttt{IndexWriter} \mapsto X) * \textbf{Id} \\
G & \stackrel{\text{def}}{=} & (\textbf{Id} \vee \textbf{UpdData} \vee \textbf{UpdWrite} \vee \textbf{UpdVer} \vee \textbf{UpdRead}) \wedge (I \ltimes I) \\
R & \stackrel{\text{def}}{=} & \textbf{Id} \wedge (I \ltimes I) \\
\mathcal{M} & \stackrel{\text{def}}{=} & \boxminus(R \vee G)
\end{array}$$

CPS Interface Safety Properties Proof Failure

```
Theorem FRM : ∀(P Q R : tprop)(C : stmt),
(∀ s s', R s → id_progression s s' → R s') →
triple P C Q → triple (P * R) C (Q * R).

intros P Q R C sidecond H. intros.
dependent induction H.
apply triple_intro; intros t u tid H0.
dependent induction H0.
generalize (unrelated_state_satisfies t1 t2 t P H1 H0); intros.
split. generalize (H t u tid H3); intros.
inversion H4. assumption. intro.
cut (forall t',  step_trace_star tid (C, t1) (abort, t')). intros.
generalize (trace_frame C t t1 t2 tid H5 H0 u H4). intros.
inversion H6 as [u1 H7]; clear H6.
inversion H7 as [u2 H8]; clear H7.
dependent induction H8. dependent induction H7.
generalize (H t1 u1 tid H1). intros.
dependent induction H9. generalize (H10 H8). intros.
eapply (tsep_conj_intro). apply H6. assumption.
eapply sidecond. apply H2. assumption. intros.
generalize (H t1 t' tid H1). intros.
dependent induction H5. apply H5.
Qed.
```

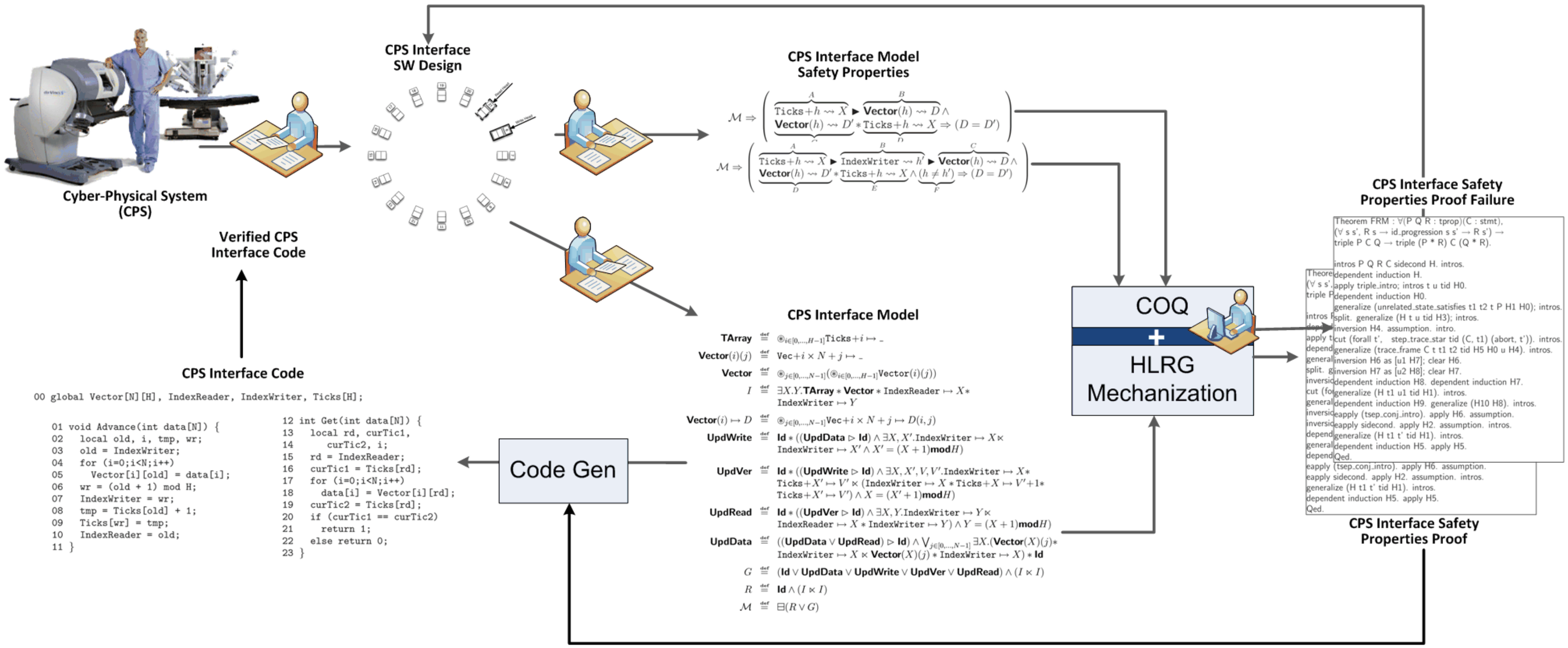
CPS Interface Safety Properties Proof

```
eapply (tsep_conj_intro). apply H6. assumption.
eapply sidecond. apply H2. assumption. intros.
generalize (H t1 t' tid H1). intros.
dependent induction H5. apply H5.
Qed.
```

Goal

Model Driven Design: code generation from verified models

- (At least for a critical subset of code)



CPS Interface Model Safety Properties

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{B} \wedge \\ \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{C} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{D} \Rightarrow (D = D') \end{array} \right)$$

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\texttt{IndexWriter} \rightsquigarrow h'}^{B} \blacktriangleright \overbrace{\textbf{Vector}(h) \rightsquigarrow D}^{C} \wedge \\ \underbrace{\textbf{Vector}(h) \rightsquigarrow D'}_{D} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{E} \wedge \underbrace{(h \neq h')}_{F} \Rightarrow (D = D') \end{array} \right)$$

CPS Interface Model

$$\begin{array}{rcl}
\textbf{TArray} & \overset{\text{def}}{=} & \circledast_{i\in[0,\dots,H-1]}\texttt{Ticks}+i \mapsto \_ \\
\textbf{Vector}(i)(j) & \overset{\text{def}}{=} & \texttt{Vec}+i \times N + j \mapsto \_ \\
\textbf{Vector} & \overset{\text{def}}{=} & \circledast_{j\in[0,\dots,N-1]}(\circledast_{i\in[0,\dots,H-1]}\texttt{Vector}(i)(j)) \\
I & \overset{\text{def}}{=} & \exists X.Y.\textbf{TArray} * \textbf{Vector} * \texttt{IndexReader} \mapsto X * \\
 & & \texttt{IndexWriter} \mapsto Y \\
\textbf{Vector}(i) \mapsto D & \overset{\text{def}}{=} & \circledast_{j\in[0,\dots,N-1]}\texttt{Vec}+i \times N + j \mapsto D(i,j) \\
\textbf{UpdWrite} & \overset{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdData} \rhd \textbf{Id}) \wedge \exists X, X'.\texttt{IndexWriter} \mapsto X \ltimes \\
 & & \texttt{IndexWriter} \mapsto X' \wedge X' = (X+1)\textbf{mod}H) \\
\textbf{UpdVer} & \overset{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdWrite} \rhd \textbf{Id}) \wedge \exists X, X', V, V'.\texttt{IndexWriter} \mapsto X * \\
 & & \texttt{Ticks}+X' \mapsto V' \ltimes (\texttt{IndexWriter} \mapsto X * \texttt{Ticks}+X \mapsto V'+1 * \\
 & & \texttt{Ticks}+X' \mapsto V') \wedge X = (X'+1)\textbf{mod}H) \\
\textbf{UpdRead} & \overset{\text{def}}{=} & \textbf{Id} * ((\textbf{UpdVer} \rhd \textbf{Id}) \wedge \exists X, Y.\texttt{IndexWriter} \mapsto Y \ltimes \\
 & & \texttt{IndexReader} \mapsto X * \texttt{IndexWriter} \mapsto Y) \wedge Y = (X+1)\textbf{mod}H) \\
\textbf{UpdData} & \overset{\text{def}}{=} & ((\textbf{UpdData} \vee \textbf{UpdRead}) \rhd \textbf{Id}) \wedge \bigvee_{j\in[0,\dots,N-1]} \exists X.(\textbf{Vector}(X)(j) * \\
 & & \texttt{IndexWriter} \mapsto X \ltimes \textbf{Vector}(X)(j) * \texttt{IndexWriter} \mapsto X) * \textbf{Id} \\
G & \overset{\text{def}}{=} & (\textbf{Id} \vee \textbf{UpdData} \vee \textbf{UpdWrite} \vee \textbf{UpdVer} \vee \textbf{UpdRead}) \wedge (I \ltimes I) \\
R & \overset{\text{def}}{=} & \textbf{Id} \wedge (I \ltimes I) \\
\mathcal{M} & \overset{\text{def}}{=} & \boxminus(R \vee G)
\end{array}$$

CPS Interface Safety Properties Proof Failure

```
Theorem FRM : ∀(P Q R : tprop)(C : stmt),
(∀ s s', R s → id_progression s s' → R s') →
triple P C Q → triple (P * R) C (Q * R).

intros P Q R C sidecond H. intros.
dependent induction H.
apply triple_intro; intros t u tid H0.
dependent induction H0.
generalize (unrelated_state_satisfies t1 t2 t P H1 H0); intros.
split. generalize (H t u tid H3); intros.
inversion H4. assumption. intro.
cut (forall t',   step_trace_star tid (C, t1) (abort, t')). intros.
generalize (trace_frame C t t1 t2 tid H5 H0 u H4). intros.
inversion H6 as [u1 H7]; clear H6.
inversion H7 as [u2 H8]; clear H7.
dependent induction H8. dependent induction H7.
generalize (H t1 u1 tid H1). intros.
dependent induction H9. generalize (H10 H8). intros.
eapply (tsep_conj_intro). apply H6. assumption.
eapply sidecond. apply H2. assumption. intros.
generalize (H t1 t' tid H1). intros.
dependent induction H5. apply H5.
Qed.
```

CPS Interface Safety Properties Proof

```
Theore
(∀ s s',
triple P

intros P
depe
apply t
depend
general
split. g
inversio
cut (fo
general
inversio
inversio
depend
general
depend
eapply (tsep_conj_intro). apply H6. assumption.
eapply sidecond. apply H2. assumption. intros.
generalize (H t1 t' tid H1). intros.
dependent induction H5. apply H5.
Qed.
```

CPS Interface Code

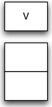
```
global Vector[N][H], IndexReader, IndexWriter, Ticks[H];

void Advance(int data[N]) {
  local old, i, tmp, wr;
  old = IndexWriter;
  for (i=0;i<N;i++)
    Vector[i][old] = data[i];
  wr = (old + 1) mod H;
  IndexWriter = wr;
  tmp = Ticks[old] + 1;
  Ticks[wr] = tmp;
  IndexReader = old;
}

int Get(int data[N]) {
  local rd, curTic1,
     curTic2, i;
  rd = IndexReader;
  curTic1 = Ticks[rd];
  for (i=0;i<N;i++)
    data[i] = Vector[i][rd];
  curTic2 = Ticks[rd];
  if (curTic1 == curTic2)
    return 1;
  else return 0;
}
```

Design: State Vector Storage/Access

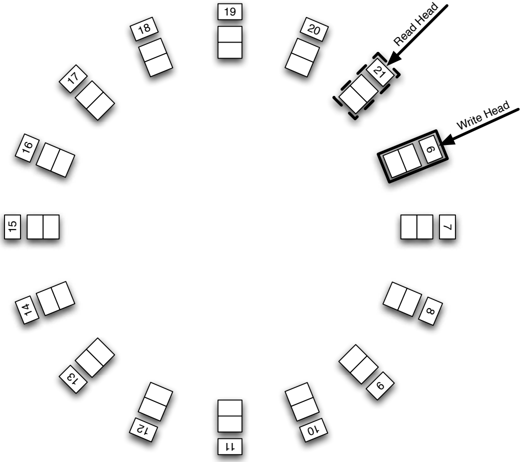
- Consider a system that has
    - A single “writer” thread
    - A single storage location for the state vector, with a version ’v’ to distinguish between updates
    - Many “reader” threads



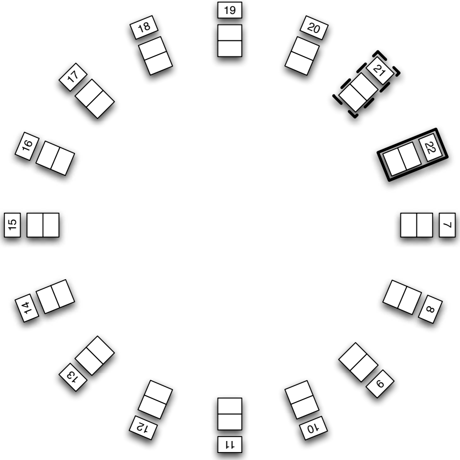
- Slow readers’ data will be corrupted by a fast writer
- Readers cannot tell whether the vector has been updated/corrupted during the read

Design: Circular Buffer

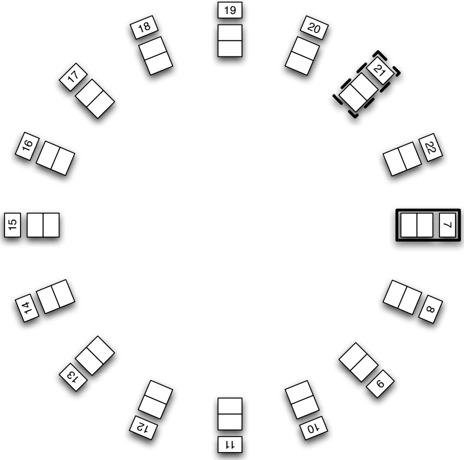
- SAW maintains a circular buffer of slots
- Each slot contains storage for the state vector and a version number
- Read and write indices indicate most recent slot completely updated, and slot currently being updated



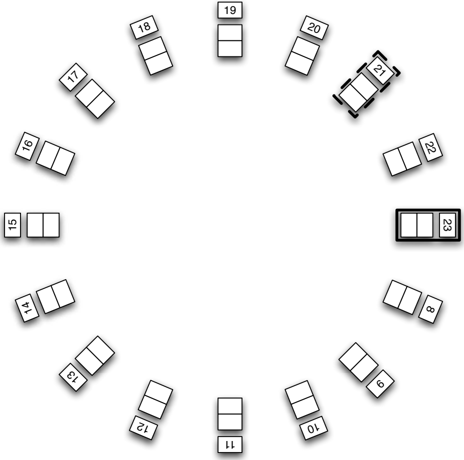
Design: Starting State



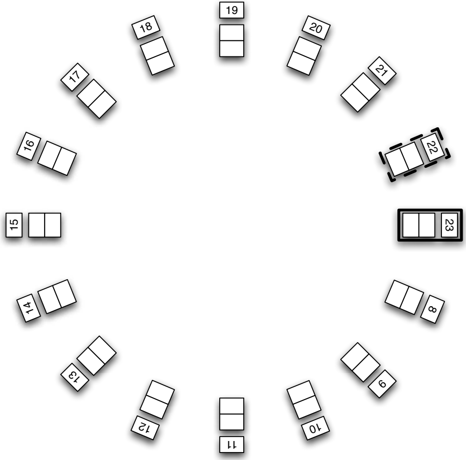
Design: Advance Write Index



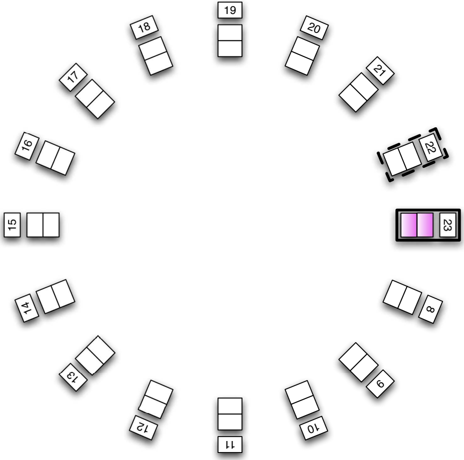
Design: Update Version



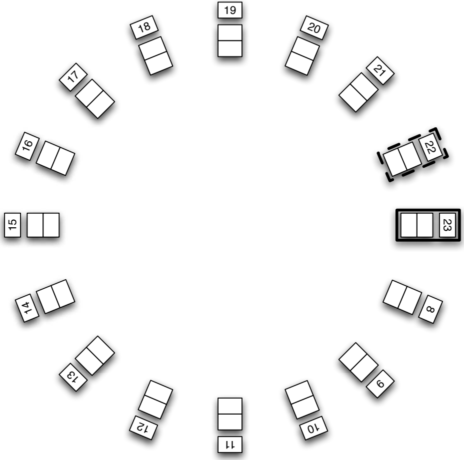
Design: Advance Read Index



Design: Write State Vector

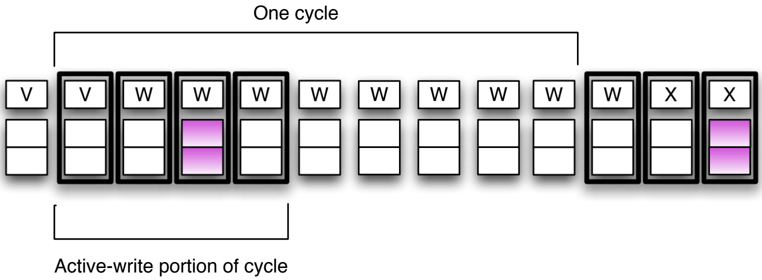


Design: Complete Write



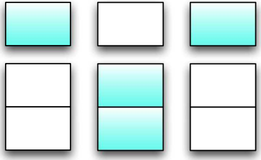
Implementation: Write Cycle

- Display one buffer slot as it changes state
- Time progresses from left to right



Implementation: Read Strategy

- Check version before and after read to ensure no corruption of data
- Reasoning: Writer updates version before writing, so any reader will notice different versions if the writer changed the data during the read



Implementation: Detecting A Corrupted Read

- This interleaving illustrates the detection of a corrupted read by observing a change in version numbers

FM Approach: HLRG Program Logic

- HLRG predicates apply to traces, or sequences of system states representing the progression of the system over time
  - Temporal operators allow expression of statements connecting state in the present to states in the past
- Rely/guarantee allows reasoning about concurrent threads
- Separation logic allows local reasoning
- Yale colleagues developed sound proof rules that worked in the presence of these traces and these operators[1][2]

---

[1] X. Feng. Local rely-guarantee reasoning, In Proc. 36th ACM Symp. on Principles of Prog. Lang., Jan. 2009

[2] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. Yale Technical Report

FM Approach: Operators in HLRG

- Some operators we might encounter:
    - $P \wedge Q$ is additive conjunction, where all of the context is used to satisfy P and Q
    - $P * Q$ multiplicative conjunction, where part of the context is used to satisfy P and another disjoint part satisfies Q
    - $P \blacktriangleright Q$ means at some point in the past, P was true, and at some later time, Q became true
    - $P \triangleright Q$ means at some point in the past, P was true, and thereafer, Q held
    - $\diamondminus P$ means at some point in the past or in the present, P happened
    - $\boxminus P$ means that P holds always

FM Approach: Describing the Domain of Shared State

- $$\begin{aligned} \mathsf{TArray} &\overset{\text{def}}{=} \circledast_{i\in[0,\ldots,H-1]}\mathtt{Ticks}{+}i \mapsto \_ \\ \mathsf{Vector}(i)(j) &\overset{\text{def}}{=} \mathtt{Vec}{+}i \times N + j \mapsto \_ \\ \mathsf{Vector} &\overset{\text{def}}{=} \circledast_{j\in[0,\ldots,N-1]}(\circledast_{i\in[0,\ldots,H-1]}\mathtt{Vector}(i)(j)) \\ I &\overset{\text{def}}{=} \exists X.Y.\mathsf{TArray} * \mathsf{Vector} * \mathtt{readindex} \mapsto X * \\ &\quad\ \mathtt{writeindex} \mapsto Y \end{aligned}$$
- $$\mathsf{Vector}(i) \mapsto D \overset{\text{def}}{=} \circledast_{j\in[0,\ldots,N-1]}\mathtt{Vec}{+}i \times N + j \mapsto D(i,j)$$

FM Approach: Atomic steps taken

- $$\mathrm{UpdWrite} \overset{\text{def}}{=} \mathrm{Id} * ((\mathrm{UpdData} \rhd \mathrm{Id}) \wedge \exists X, X'.\texttt{writeindex} \mapsto X \ltimes \texttt{writeindex} \mapsto X' \wedge X' = (X+1)\mathrm{mod} H)$$
- $$\mathrm{UpdVer} \overset{\text{def}}{=} \mathrm{Id} * ((\mathrm{UpdWrite} \rhd \mathrm{Id}) \wedge \exists X, X', V, V'.\texttt{writeindex} \mapsto X * \texttt{Ticks}+X' \mapsto V' \ltimes (\texttt{writeindex} \mapsto X * \texttt{Ticks}+X \mapsto V'+1 * \texttt{Ticks}+X' \mapsto V') \wedge X = (X'+1)\mathrm{mod} H)$$
- $$\mathrm{UpdRead} \overset{\text{def}}{=} \mathrm{Id} * ((\mathrm{UpdVer} \rhd \mathrm{Id}) \wedge \exists X, Y.\texttt{writeindex} \mapsto Y \ltimes \texttt{readindex} \mapsto X * \texttt{writeindex} \mapsto Y) \wedge Y = (X+1)\mathrm{mod} H)$$
- $$\mathrm{UpdData} \overset{\text{def}}{=} ((\mathrm{UpdData} \vee \mathrm{UpdRead}) \rhd \mathrm{Id}) \wedge \bigvee_{j \in [0,\ldots,N-1]} \exists X.(\mathrm{Vector}(X)(j) * \texttt{writeindex} \mapsto X \ltimes \mathrm{Vector}(X)(j) * \texttt{writeindex} \mapsto X) * \mathrm{Id}$$

FM Approach: Program Description

- We have transformed the program into a description of the possible atomic steps it may take that affect computer state
- $$\begin{aligned} G &\stackrel{\text{def}}{=} (\mathsf{Id} \vee \mathsf{UpdData} \vee \mathsf{UpdWrite} \vee \mathsf{UpdVer} \vee \mathsf{UpdRead}) \wedge (I \ltimes I) \\ R &\stackrel{\text{def}}{=} \mathsf{Id} \wedge (I \ltimes I) \\ \mathcal{M} &\stackrel{\text{def}}{=} \boxminus(R \vee G) \end{aligned}$$

Verification: First Attempt At Read Data Integrity

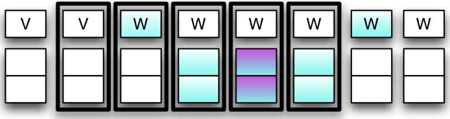
- Now state the theorem we seek to prove, and show that our machine implies that theorem
- Prove key lemma:

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\mathsf{Vector}(h) \rightsquigarrow D}^{B} \wedge \\ \underbrace{\mathsf{Vector}(h) \rightsquigarrow D'}_{C} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{D} \Rightarrow (D = D') \end{array} \right)$$

- This is unprovable, therefore there is a flaw in our design

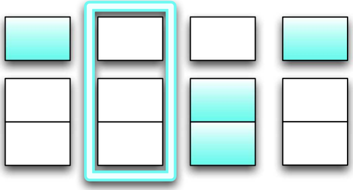
Verification: Example of Read Strategy Problem

- There is a brief span of time where the data becomes inconsistent without a change in the version number
- An example interleaving that illustrates the problem with our read strategy
- The reader cannot detect corruption in the read



Verification: Improved Read Strategy

- Check the position of the write index in between the first version check and the actual reading of data
- If the write index is pointing to the current slot (i.e. we are in the active write portion of the cycle, then assume that our data is corrupted
- If the version number has changed during the read, also assume the data is corrupted



Verification: Data Read Integrity Theorem

- Able to successfully complete proof of data read integrity
- Improved key lemma:
  -

$$\mathcal{M} \Rightarrow \left(\begin{array}{l} \overbrace{\texttt{Ticks}+h \rightsquigarrow X}^{A} \blacktriangleright \overbrace{\texttt{writeindex} \rightsquigarrow h'}^{B} \blacktriangleright \overbrace{\mathsf{Vector}(h) \rightsquigarrow D}^{C} \wedge \\ \underbrace{\mathsf{Vector}(h) \rightsquigarrow D'}_{D} * \underbrace{\texttt{Ticks}+h \rightsquigarrow X}_{E} \wedge \underbrace{(h \neq h')}_{F} \Rightarrow (D = D') \end{array} \right)$$

  - When a read completes, the value that is returned accurately reflects what was stored in memory for that state vector element during the read; and that value was stable during the read, i.e. no writer was altering it or may have altered it during that time.

Conclusions

- Towards practical application of formal methods in the design of medical systems
    - Moving in the direction of incrementally introducing FM into development process, using the SAW as a test case
    - Certify properties for critical pieces of reusable framework in which testing is inadequate
- Immediate benefit to the surgical assistant workstation
    - Found and fixed a design flaw in the SAW software
    - Guaranteed that there are no more bugs in the state-table component that could unexpectedly impact the data integrity of the state vector
    - Enumerated specific axioms on which this guarantee rests