Process Extraction in a Logic of Events

Robert L. Constable
Cornell University
Based on joint work with Mark Bickford
Outline

1. Introduction

2. Event Structures

3. Process Extraction from Proofs
Introduction

State of Semantics

research – thinking 20 years out

technology – very promising (Intel, Microsoft, AMD, …)

Next Steps

needs in distributed (network) computing

implementing a logic of events
Outline

1. Introduction

2. Event Structures

3. Process Extraction from Proofs
Design of the Logic of Events was strongly informed by the systems group and partners in several research teams.

We captured the conceptual level at which the protocol designers worked.

  event based analysis
  high level atomicity

We abstracted many high level distributed system concepts into an accessible logic with practical value:
many details can be automatically added by the extractors to MA to Java
At the level of abstraction we use, one can see many interpretations, “good science stories,” as well as good “technology stories.”

The general setting is **observables** in time/space.

Space: locations at which “things happen”

Time: events happen at locations, but no global clock, only causal order on events

Two kind of events

- local action
- receive a message

Every event send messages (on links $l$ with tag $t$ value $v$)
Logic of Events, circa 2005

What distinguishes our event structures approach?

- based directly on Leslie Lamport’s insights;
  type theory captures them naturally
- used by distributed computing researchers, matches their intuitions
- integrated into LPE, hence into procedural programming
- completely formalized
- supports proofs-as-processes of synthesis and programming
- widely applicable: verification, optimization, documentation, security, performance
- organizes a fundamental set of concepts
Here is the signature of events with order. We need the large type $\mathbb{D}$ of discrete small types (those with decidable equality). Let $\mathbb{D}$ be this large type.

**EOrder**

- $E : \mathbb{D}$
- $\text{Loc} : \mathbb{D}$
- $\text{Pred?} : E \rightarrow (E + \text{Loc})$
- $\text{Sender?} : E \rightarrow (E + \text{Unit})$
Axiom 1: For any event e that emits a signal, we can find an event e’ by which e is received.

Axiom 2: The predecessor function, pred?, is injective.

Axiom 3: The predecessor relation, x L y, is strongly well founded, where x L y iff for y not the first event

\[ x = \text{pred?}(y) \text{ or } x = \text{sender?}(y). \] Namely, there is a function \( f \) from E to Nat such that \( x L y \) implies \( f(x) < f(y) \).
We can squeeze a lot of information from these functions.

loc: \( E \rightarrow \text{Loc} \)

first: \( E \rightarrow \text{Bool} \)

rcv: \( E \rightarrow \text{Bool} \)

\(< \) causal order \((E \rightarrow E \rightarrow \text{Prop})\)

\(<_{\text{loc}} \) local linear order \((E \rightarrow E \rightarrow \text{Prop})\)
Events in Space/Time
Progression of Event Structures

We progressively define the following richer structures

EOrder – events with causal order

EVValue – events have values, val(e)

EState – locations have state, temporal operators

    initially, when, after

ECom – communication topology is given by a graph

Etime – real time is added

ETrans – transition function
Event Structures (with state)

Discrete types: Events (E), Loc, Actions (Act), Tag, Link (L)

\[
\begin{align*}
\text{loc: } & \quad E \rightarrow \text{Loc} \\
\text{kind: } & \quad E \rightarrow \text{KND} \\
\text{first: } & \quad E \rightarrow B \\
\text{pred: } & \quad e: \{x:E | \neg \text{first}(x)\} \rightarrow \{x:E | \text{loc}(x) = \text{loc}(e)\} \\
\text{snder: } & \quad \{x:E | \text{kind}(x) = \text{rcv}\} \rightarrow E \\
\wedge: & \quad E \rightarrow E \rightarrow B \\
\text{Ty: } & \quad E \rightarrow \text{Type} \\
\text{val: } & \quad x:E \rightarrow \text{Ty}(x) \\
\text{when: } & \quad x: \text{Id} \rightarrow e: E \rightarrow \text{Ty}(\text{loc}(e), x) \\
\text{after: } & \quad x: \text{Id} \rightarrow e: E \rightarrow \text{Ty}(\text{loc}(e), x) \\
\text{initial: } & \quad x: \text{Id} \rightarrow i: \text{Loc} \rightarrow \text{Ty}(i, x)
\end{align*}
\]
Outline

1. Introduction

2. Event Structures
   - Signatures and axioms
   - Sample specification
   - Computation model

3. Process Extraction from Proofs
Example – Two-Phase Handshake

\[ E_p = \{ e \in E \mid \text{loc}(e) = p \} \]

\[ \text{Snd}_{p,l} = \{ e \in E_p \mid \text{sends}(e, l) \neq \text{null} \} \]

\[ \text{Rcv}_{p,l} = \{ e \in E_p \mid \text{kind}(e) \text{ is receive on } l \} \]
We illustrate this process by deriving a protocol for the two-phase handshake from a proof that its specification is realizable.

(1) \( \forall e_1, e_2 : \text{Snd}_{S,1_1} \cdot \exists r : \text{Rcv}_{S,1_2} \cdot (e_1 < e_2 \Rightarrow e_1 < r < e_2) \)

(2) \( \forall e_1, e_2 : \text{Snd}_{R,1_2} \cdot \exists r_1, r_2 : \text{Rcv}_{R,1_1} \cdot (e_1 < e_2 \Rightarrow r_1 < e_1 < r_2 < e_2) \)
Picture of a Computation

1. $\langle s_1, a_1, m_1 \rangle @ t$
2. $\langle s_2, a_2, m_2 \rangle @ t$
3. $\langle s_3, a_3, m_3 \rangle @ t$
4. $\langle s_4, a_4, m_4 \rangle @ t$
5. $\langle s_5, a_5, m_5 \rangle @ t$

1. $\langle s_1, a_1, m_1 \rangle @ t + 1$
2. $\langle s_2, a_2, m_2 \rangle @ t + 1$
3. $\langle s_3, a_3, m_3 \rangle @ t + 1$
4. $\langle s_4, a_4, m_4 \rangle @ t + 1$
5. $\langle s_5, a_5, m_5 \rangle @ t + 1$
Executions of distributed systems are event systems in a natural way. Executions are typically indexed by time, and that can be discrete, say $t$ is a natural number, $t \in \text{Nat}$.

At each moment of time, a process at $i$ is in a state, $s(i, t)$, and the links are lists of tagged messages, $m(l, t)$. At each locus $i$ and time $t$, there is an action, $a(i, t)$, taken. The action can be null, i.e., no state change, no receives, hence no sends.
Fair-Fifo Executions

We assume executions are \textit{fair}: channels are loss-less; and \textit{fifo}: messages are received in the order sent.

1. Only the process at $i$ can send messages on links originating at $i$.

2. A receive action at $i$ must be on a link whose destination is $i$ and whose message is at the head of the queue on that link.

3. There can be \textcolor{red}{null actions} that leave a state unchanged between $t$ and $t + 1$.

4. Every queue is examined \textcolor{red}{infinitely often}, and if it is nonempty, a message is delivered.

5. The \textcolor{red}{precondition} of every local action is examined infinitely often and if true the action is taken.
1. Introduction
2. Event Structures
3. Process Extraction from Proofs
\[ \vdash \forall x:A. \exists y:B. \text{R}(x, y) \quad \text{ext} \quad \text{C}(g_1, g_2) \]

\[ H_1 \vdash G_1, \text{ext} \ g_1 \quad H_2 \vdash G_2 \quad \text{ext} \quad g_2 \]
Refinements for Programs

\[
\vdash \forall x : A. \exists y : B. R(x, y) \text{ ext } \lambda x. \text{cut}(x, z.g(x, z); l(x))
\]

\[
x : A \vdash \exists y : B. R(x, y) \text{ ext } \text{cut}(x, z.g(x, z); l(x))
\]

by cut L

1. \[
x : A, z : L \vdash \exists y : B. R(x, y) \text{ ext } g(x, z)
\]

by D \quad \neg g(x, z)\]

\[
x : A, z : L \vdash R(x, g(x, z))
\]

2. \[
x : A \vdash L \text{ ext } l(x)
\]

by ___
Refinements for Systems

\[ \vdash \exists D : \text{System}. \ \forall es : ES(D). \]
\[ R(es) \text{ ext } \text{Comp}(pf_1(D_1, es_1), pf_2(D_2, es_2)) \]

by Comp

1. \[ D_1 : \text{System}(G, \text{Loc}, \text{Lnk}) \]
   \[ es_1 : ES(D_1) \vdash R_1(es_1) \text{ ext } pf_1(D_1, es_1) \]

2. \[ D_2 : \text{System}(G, \text{Loc}, \text{Lnk}) \]
   \[ es_2 : ES(D_2) \vdash R_2(es_2) \text{ ext } pf_2(D_2, es_2) \]
Two-Phase Handshake Theorem

Theorem:
\[ \forall e_1, e_2 : \text{Snd}_S . e_1 < e_2 \Rightarrow \exists r : \text{Rcv}_S . e_1 < r < e_2 \]

What are the consequences of \( e_1 < e_2 \)?

S sent two messages

Can we infer further consequences?

Relate send events to knowledge, create a boolean variable \( rdy \)

- Require \( rdy \) to be true when a send event occurs
- Require \( rdy \) to be false after a send event \( e \)
Two-Phase Handshake

Theorem:
\[ \forall e_1, e_2 : \text{Snd}_S. e_1 < e_2 \Rightarrow \exists r : \text{Rcv}_S. e_1 < r < e_2 \]
How to establish this by reasoning?

Suppose \( e_1 < e_2 \) are sends on link \( \ell \) from \( S \),
Then by \( \text{L1} \) (\( \text{rdy} \) after \( e_1 \)) = false.
Since \( e_2 \) is a send, \( \text{rdy} \) must be true at \( e_2 \) by \( \text{L2} \).
Therefore some event \( e' \) before \( e_2 \) and after \( e_1 \)
must set \( \text{rdy} \) to true.
By \( \text{L3} \) the event \( e' \) must be received from \( R \) on link \( \ell' \).
Let \( r \) be this \( e' \).
If we have the lemmas, then the theorem is true.
L1. If $S$ sends on link $\ell$ then it waits
   $\forall e: \text{Snd}_S, \ell (\text{rdy after } e) = \text{false}$

L2. $S$ sends on link $\ell$ only when $\text{rdy} = \text{true}$
   $\forall e: \text{Snd}_S, \ell (\text{rdy when } e) = \text{true}$

L3. After $S$ on link $\ell$ sends it is ready only after an
    acknowledgement on link $\ell'$
   $\forall e : \text{Snd}_S, \ell e \text{ changes } \text{rdy to true} \Rightarrow$
   $e$ is a receive from $R$ on link $\ell'$. 
L1. \[ \forall e : \text{Snd}_s, \ell . (\text{rdy after } e) = \text{false} \]

We can require that the sends on \( \ell \) are caused by the action of setting \text{rdy} to false.

\[ a : \text{rdy} := \text{false}; \text{send (} \ell, < \text{tag, m } > \) \]

We also require that only action \( a \) can send on \( \ell \).

This is a frame condition.

L2. \[ \forall e : \text{Snd}_s, \ell . (\text{rdy when } e) = \text{true} \]

We require the \text{precondition} on the action \( a \) that \( \text{rdy} = \text{true} \)

\[ a : \pre(\text{rdy} = \text{true}) \Rightarrow \text{rdy} := \text{false}; \text{send (} \ell, < \text{tag, m } > \) \]
L3: $\forall e : S\text{nd}_s, \ell . (e \Delta \text{rdy}) = \text{true} \Rightarrow$

kind($e$) = rcv on $\ell'$

We stipulate that a receive sets rdy to true

and that only a rcv or a can change rdy.

b: rcv($\ell'$, m) $\Rightarrow$ rdy:= false

only [a,b] affect rdy (frame condition)
action local (a) sends on $l_1 < \text{tag, v} >$
only [a] sends on $l_1$
state $\text{rdy} : \mathbb{B}$ ; initially $\text{rdy} = \text{true}$
precondition a is $\text{rdy} = \text{true}$
  effect local (a) $\text{rdy} := \text{false}$
action $\text{rcv}_{\ell_2} < \text{ack} > : \text{Atom}$
  effect $\text{rdy} := \text{true}$
only [local (a), $\text{rcv}_{\ell_2} < \text{ack}>$] affect $\text{rdy}$
Outline

1. Introduction

2. Event Structures

3. Process Extraction from Proof
   Examples – two phase handshake
   -- leader election in a ring

   Realizers

   Realizability Theorems
Specification for Leader Election in a Ring

Leader Election

In a Ring $R$ of Processes with Unique Identifiers (uid’s)

Specification

Let $R$ be a non-empty list of locations linked in a ring

Let $n(i) = \text{dst(out}(i)),\text{ the next location}$

Let $p(i) = n^{-1}(i),\text{ the predecessor location}$

Let $d(i,j) - \mu k \geq 1. n^k(i) = j,\text{ the distance from } i \text{ to } j$

Note $i \neq p(j) \Rightarrow d(i,p(j)) = d(i,j)-1.$
Leader (R,es) == \exists \text{l.dr}: R. (\exists e@\text{l.dr}. \text{kind}(e)=\text{leader}) \&
(\forall i:R. \forall e@i. \text{kind}(e)=\text{leader} \Rightarrow i=\text{l.dr})

Theorem \ 
\forall R: \text{List(Loc)}. \text{Ring}(R)
\exists D: \text{Dsys}(R). \text{Feasible}(D) \&
\forall es: \text{ES}. \text{Consistent}(D,es). \text{Leader}(R,es)
Let $LE(R,es) = \forall i:R.$

1. $\exists e.\ kind(e) = rcv(out(i), <vote,uid(i)>)$

2. $\forall e.\ kind(e) = rcv(in(i), <vote,u>) \Rightarrow$
   $(u > uid(i) \Rightarrow \exists e'.\ kind(e') = rcv(out(i), <vote,u>))$

3. $\forall e'.\ [ (kind(e') = rcv(out(i), <vote,uid(i)>)) \lor$
   $\exists e.\ (kind(e) = rcv(in(i), <vote,u>) \land (e < e' \land u > uid(i))) ]$

4. $\forall e@i.\ kind(e) = rcv(in(i),uid(i)).\ \exists e'@i.\ kind(e') = leader$

5. $\forall e@i.\ kind(e) = leader.\ \exists e@i.\ kind(e) = rcv(in(i), <vote,uid(i)>)$
Realizing Leader Election

Theorem

\[ \forall R : \text{List} (\text{Loc}) . \text{Ring} (R) \]
\[ \exists D : \text{Dsys} (R) . \text{Feasible} (D) . \]
\[ \forall \text{es} : \text{Consistent} (D, \text{es}) . (\text{LE}(R, \text{es}) \implies \text{Leader}(R, \text{es})) \]

Proof: Let \( m = \max \{ \text{uid}(i) \mid i \in R \} \), then \( \text{ldr} = \text{uid}^{-1}(m) \).

We prove that \( \text{ldr} = \text{uid}^{-1}(m) \) using three simple lemmas.
Intuitive argument that a leader is elected

1. Every $i$ will get a vote from predecessor for the predecessor.

2. When a process $i$ gets a vote $u$ from its predecessor with $u > \text{uid}(i)$ it sends it on.

3. Every rcv is either vote of predecessor $\text{rcvin}(i)$ for itself or a vote larger than process id before.

4. If a processor sets a vote for itself, it declares itself ldr.

5. If a processor declares ldr it got a vote for itself.
Lemmas

Lemma 1. \( \forall i : R. \exists e @ i. \text{kind}(e) = \text{rcv} (\text{in}(i), <\text{vote}, \text{ldr}>) \)

By induction on distance of \( i \) to \( \text{ldr} \).

Lemma 2. \( \forall i, j : R. \forall e @ i. \text{kind}(e) = \text{rcv} (\text{in}(i), <\text{vote}, j>). \)

\( (j = \text{ldr} \lor d(\text{ldr}, j) < d(\text{ldr}, i)) \)

By induction on causal order of \( \text{rcv} \) events.

Lemma 3. \( \forall i : R. \forall e' @ i. (\text{kind}(e') = \text{leader} \Rightarrow i = \text{ldr}) \)

If \( \text{kind}(e') = \text{leader} \), then by property 5, \( \exists v @ i. \text{rcv} (\text{in}(i), <\text{vote}, \text{uid}(i)>) \).

Hence, by Lemma 2 \( i = \text{ldr} \lor (d(\text{ldr}, i) < d(\text{ldr}, i)) \)

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

\( \exists e. \text{kind}(e) = \text{rcv} (\text{in}(\text{ldr}), <\text{vote}, \text{uid}(\text{ldr}>) \)

which follows from Lemma 1.

QED
Realizing the clauses of $\text{LE}(R, es)$

We need to show that each clause of $\text{LE}(R, es)$ can be implemented by a piece of a distributed system, and then show the pieces are compatible and feasible.

We can accomplish this very logically using these Lemmas:

- Constant Lemma
- Send Once Lemma
- Recognizer Lemma
- Trigger Lemma
Implementing Clause 1 of $\text{LE}(R,es)$

1. $\exists e. \text{kind}(e) = \text{rcv}(\text{out}(i), <\text{vote}, \text{uid}(i)>)$

We need to send $<\text{vote}, \text{uid}(i)>$ from each $i$.

The Constant Lemma allows us to create a state variable $\text{me}$ at each $i$ with $\text{me} = \text{uid}(i)$

$\forall e @ i. (\text{me when } e) = \text{uid}(i)$

The Send Once Lemma lets the process at $i$ send $\text{uid}(i)$

$\exists e. \text{kind}(e) = \text{rcv}(\text{out}(i), <\text{vote}, \text{uid}(i)>)$
Implementing Clauses 4, 5 of $\text{LE}(R, es)$

We can instantiate the Trigger Lemma to obtain

$$\forall i : \text{Loc.}$$

$$\forall e' @ i. \text{kind}(e') = \text{leader}.$$  

$$(\exists e @ i. e < e'. \text{kind}(e) = \text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle))$$

$$\forall e @ i. \text{kind}(e) = \text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle) \Rightarrow$$

$$\exists e' @ i. \text{kind}(e') = \text{leader}$$
Leader Election Message Automaton

state $me : \mathbb{N}$; initially $uid(i)$
state $done : B$; initially $false$
state $x : B$; initially $false$
action $vote$; precondition $\neg done$
  effect $done := true$
  sends $[msg(out(i), vote, me)]$
action $rcv_{in(i)}(vote)(v) : \mathbb{N}$;
  sends if $v > me$ then $[msg(out(i), vote, v)]$ else[]
  effect $x :=$ if $me = v$ then $true$ else $x$
action $leader$; precondition $x = true$
only $rcv_{in(i)}(vote)$ affects $x$
only $vote$ affects $done$
only $\{vote, rcv_{in(i)}(vote)\}$ sends $out(i), vote$
1. Introduction

2. Event Structures

3. Process Extraction from Proof

   Examples – two phase handshake

   -- leader election in a ring

   Realizers

   Realizability Theorems
Message Automata Clauses

- \( \text{\texttt{\textbf{@ i x : T initially = v}}} \)
- \( \text{\texttt{\textbf{@ i effect k(v : t) on x}}} \)
  \[
  x := f \text{ state f}
  \]
- \( \text{\texttt{\textbf{@ i precondition a(v:t) is P state v}}} \)
- \( \text{\texttt{\textbf{k(v : T) sends on link l}}} \)
  \[
  [t_{g_1}, f_1 \text{ state v}; \ldots; t_{g_n}, f_n \text{ state v}]
  \]
- \( \text{\texttt{\textbf{@ i only [k_1, \ldots, k_n] sends on link l with tag t}}} \)
- \( \text{\texttt{\textbf{A \oplus B, where A, B are message automata}}} \)
There are many basic formulas of type theory that have atomic (or axiomatic) realizers, e.g., induction.

\[
(P(0) \land \forall x: \mathbb{N}. P(x) \Rightarrow P(x + 1)) \Rightarrow \forall x: \mathbb{N}. P(x)
\]

realized by \( \lambda (h. \lambda (x. \text{ind}(x; 1 \circ f(h); u, v. 2 \circ f(h)(u)(v)))) \)

There are six kinds of event specifications that have atomic (or axiomatic) realizers.
Realizable specifications

- **initial** -- gives the value of a variable
- **effect** -- defines a change of state based on an action
- **frame** -- limits actions that can change a variable
- **pre** -- takes an action if a precondition is true
- **sends** -- sends tagged messages on a specified link
- **sframe** -- limits actions that can send
initial (using Message Automata)

@i state x:T; initially p(x)
realizes: p(initial (x,i))

frame

Define \( x_{\Delta e} \) as \( x \) after \( e \neq x \) when \( e \)

\[ @i \text{ only } L \text{ affects } x \]
realizes: \( \forall e@i. (x_{\Delta e} \Rightarrow \text{kind}(e) \in L) \)
effect

@i  state  x:T_1;  action  k:T_2;

k(v)  effect  x: = f(s  when  e,  val  (e))

realizes

\forall e@i.  kind(e)=k \implies

x  after  e = f(s  when  e,  val  (e))
pre

@i action k:T; k(v) precondition p(s,v)
realizes:

\( \forall e @ i. (\text{kind}(e) = k \Rightarrow p(s\text{where}, \text{val}(e))) \)

& \( \forall e @ i. \exists e'@$i.e \leq e' \) &

\( (\text{kind}(e')) = k \vee \forall v : T. \neg p(s\text{after }e', v) \)
& \( \exists v : T. p(\text{init}(es) (i), v) \Rightarrow \exists e : E. \text{loc}(e) = i \)

The skolem function in the \( \forall e \exists e' \) clause gives a "schedule" for the action k.
Compound Realizers

Realizers are built by combining the six basic clauses. In the concrete case of Message Automata, the clauses are just joined by union. In the abstract setting, there is a combining operator,

\[ R_1 \oplus R_2. \]
Realizer  
\[\equiv_{\text{def}} \hspace{0.5em} \text{rec}(X.\text{Unit}) \]
\[\begin{align*}
+X \times X \\
+Id \times T : Type \times Id \times T \\
+Id \times Type \times Id \times (\text{KndList}) \\
+IdLnk \times Id \times (\text{KndList}) \\
+Id \times ds : x : Id fp \rightarrow Type \times \text{Knd} \times T : Type \times x : Id \times (\text{State}(ds) \rightarrow T \rightarrow \text{DeclaredType}(ds;x)) \\
+ds : x : Id fp \rightarrow Type \\
\times \text{Knd} \\
\times T : Type \\
\times IdLnk \\
\times dt : x : Id fp \rightarrow Type \\
\times \((tg : Id \times (\text{State}(ds) \rightarrow T \rightarrow (\text{DeclaredType}(dt;tg) \text{ List}))) \text{ List}) \\
+Id \times ds : x : Id fp \rightarrow Type \times Id \times T : Type \times (\text{State}(ds) \rightarrow T \rightarrow \text{Prop})
\end{align*}\]
Compatibility

Arbitrary compositions, \( R_1 \oplus R_2 \) might not be compatible. For example, \( R_1 \) might be a frame condition that says

\[
R_1: \text{ only action } k \text{ can change } x \text{ and }
\]

\[
R_2: \text{ action } k' \text{ changes } x \text{ for some } k \neq k'
\]

Also compatible realizers must have compatible types

\[
R_1: \text{ declares } x \text{ to be of type } T_1
\]

\[
R_2: \text{ declares } x \text{ to be of type } T_2
\]

We must have \( T_1 \subseteq T_2 \) or \( T_2 \subseteq T_1 \)
Compatibility, continued

Compatibility is defined by 15 conditions from the 6 by 6 matrix of possibilities (half minus the diagonal). They are not decidable in theory but are in practice.
Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proof
   
   Examples – two phase handshake
   
   -- leader election in a ring

   Realizers

   Realizability Theorems
If $P$ is any event specification, then the type theory expression of the goal is this

$$\vdash \exists D : \text{DSyst. Feasible}(D) \land \forall es : \text{ES. Consistent}(D, es) \Rightarrow P(es)$$

We say that $\text{Feasible}(D)$ if $D$ has at least one execution.

We say $\text{Consistent}(D, es)$ provided $es$ is an event system that arises from a possible execution of $D$. 
A realizer R is feasible if it has an execution. For this to be possible, the clauses of R must be compatible and the types of variables, event values, and message content must be nonempty.
One of the main theorems of Bickford’s massive library is that if distributed system $D$ is feasible, then we can construct the possible executions, worlds, of it.

Moreover, from a world $W$ of $D$, we can construct event systems for $D$, $\text{es}(D)$, consistent with it.

This is a constructive proof, as are all in the library. So it defines the computational rules for the realizers given a schedule.
Theorem 1

For all DSys D, Feasible(D) ⇒ ∃w:World. Possible(D,w)

Theorem 2

For all DSys D and all Possible Worlds w of D, we can build an EventSystem es(w) Consistent (D, es(w)).