One Direction of Inscape Research

- Make practical use of specification and verification technology
  - Use the module interface specifications in the construction of software systems
  - Implement a semantic interconnection model that records unit, syntactic and semantic dependencies as the basis for evolution
  - Make practical trade-offs between logic and analysis
    - how weak a logic?
    - how strong a form of consistency checking?
- Instress — the module interface specification language
  - Specification logic (SL), C syntax for declarations
  - SL annotations for data object properties and operation interface behavior
- Inscape — using the specifications
  - Data object/operation specification instantiation
    - as the basis for the semantic interconnection and propagation
    - as a result of variable names and operation arguments
  - Propagation logic (PL)
    - construct interfaces for sequence, selection, iteration
    - construct interfaces for implementations of operations

Goals of Analysis

- Incremental Analysis
  - Use specifications as a bootstrapping mechanism — ie, assume that they are correct and use them
  - An operation is the gross grain of incremental analysis (the collection of operations individually analyzed forms the analysis of a module)
  - A statement is the fine grain of incremental analysis (the goal is to be able to consider each statement independent of its surrounding context)
- Basic Rule:
  - Every precondition and obligation must either be satisfied within the implementation or propagated to its interface.

Some Logical Intuition

- Separate consistency from propagation
  - Consistency depends on trade-offs about how to manipulate the specification logic
  - Propagation depends on trade-offs about how to statically represent a possible-execution tree as a single thread of knowledge
- Temptation to use or
  - In reducing a possible-execution tree to a static (sequential) directed graph, how do you handle the results of joining two paths?
  - Say P is true in one path, Q in the other — one is tempted to describe the result as P or Q
  - However, if not P later becomes true, the inference of Q is not a valid one, because it might well have been the path that produced P that was actually executed.
Some More Intuition

- Why we can treat each statement/operation as an indivisible unit
  - Intuitive picture of preconditions and postconditions
  - Postconditions "sink down" through an implementation
  - Preconditions "float up" through an implementation "looking" for satisfaction
- At the point where a precondition \( P \) occurs, either \( P \) is known to be true or false, or it is not known whether \( P \) is true or false
  - If \( P \) is true, then the precondition is satisfied (and it does not matter where in the preceding sequence it became true)
  - If \( P \) is false, \( P \) cannot be propagated to the interface and hence there is a problem with the implementation (it also does not matter where \( P \) became false, except to provide a range in which to fix the problem)
  - If \( P \) is unknown, then it is unknown "in" all of the preceding statements as well
- Unfortunately, obligations are not quite so tractable

Inscape’s Propagation Logic (PL)

- PL is a proposition calculus (SL is a predicate calculus with quantification) in which
  - A proposition \( P \) is either \( \text{true} (P) \) or \( \text{false} (\neg P) \)
  - The state of a proposition is either
    - \( \text{unknown} P \) is not known to be either true or false in any execution path
    - \( \text{known} P \) is known to be true in all execution paths
    - \( \text{possible} P \) is known to be true in at least one execution path
  - There is one sentence connective, and
  - There are the inference rules
    - \( +\)seq a sequential addition or join based on the state of the propositions
    - \( +\)par a parallel addition or join based on the state of the propositions
    - \( +\)con a sequential addition or join based on the consistency of the propositions

The Definition of \( +\)seq

- \( P_1 +\)seq \( P_2 \) is defined as follows

<table>
<thead>
<tr>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg P )</td>
<td>known ( P )</td>
<td>known ( P )</td>
</tr>
<tr>
<td>( P )</td>
<td>known ( \neg P )</td>
<td>known ( \neg P )</td>
</tr>
<tr>
<td>( \neg P )</td>
<td>unknown ( P ), ( \neg P )</td>
<td>( \neg P )</td>
</tr>
<tr>
<td>known ( P )</td>
<td>possible ( P )</td>
<td>known ( P )</td>
</tr>
<tr>
<td>known ( P )</td>
<td>possible ( \neg P )</td>
<td>possible ( \neg P )</td>
</tr>
<tr>
<td>known ( P )</td>
<td>possible ( P ), possible ( \neg P )</td>
<td>possible ( P ), possible ( \neg P )</td>
</tr>
<tr>
<td>possible ( P )</td>
<td>possible ( \neg P )</td>
<td>possible ( P ), possible ( \neg P )</td>
</tr>
<tr>
<td>possible ( P )</td>
<td>possible ( P )</td>
<td>possible ( P )</td>
</tr>
<tr>
<td>unknown ( P )</td>
<td>possible ( \neg P )</td>
<td>possible ( \neg P )</td>
</tr>
<tr>
<td>unknown ( P )</td>
<td>possible ( P )</td>
<td>possible ( P )</td>
</tr>
</tbody>
</table>

- Note: \( P +\)seq \( Q \) is not symmetric
- The state of \( P_2 \) supercedes the state of \( P_1 \)
  - whatever is known in \( P_2 \) supplants that of \( P_1 \)
  - whatever in \( P_1 \) is unknown in \( P_2 \) retains its state from \( P_1 \)
  - what is possible in \( P_2 \) remains so in the result, but may also reduce what is known in \( P_1 \) to a possible in the result
  - except where it was known in \( P_1 \)
The Definition of \( \text{par} \)

- \( P_1 \text{ par} P_2 \) is symmetric and defined as follows

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>known ( P \neg P )</td>
<td>known ( P \neg P )</td>
<td>known ( P \neg P )</td>
</tr>
<tr>
<td>known ( P \neg P )</td>
<td>unknown ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>known ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>known ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>known ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>known ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>unknown ( P \neg P )</td>
<td>unknown ( P \neg P )</td>
<td>unknown ( P \neg P )</td>
</tr>
<tr>
<td>unknown ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
<tr>
<td>unknown ( P \neg P )</td>
<td>possible ( P \neg P )</td>
<td>possible ( P \neg P )</td>
</tr>
</tbody>
</table>

- Only what is known (unknown) to be true in both parts is known (unknown) to be true in a parallel join of the two parts. What is known in only one part becomes possible in the result.

The Definition of \( \text{con} \) and \( \text{con} \)

- \( P_1 \text{ con} P_2 \) is defined as
  \[
  \{ p_1 \ldots p_k \in P_1 \mid p_1 \ldots p_k \text{ are consistent } P_2 \} \cup P_2
  \]
- \( P_1 \text{ con} P_2 \) is defined as
  \[
  P_1 - ( P_1 \text{ con} P_2 )
  \]

Reasoning about Sequences

- The following sets are used for reasoning about each sequent \( S_i \) in the sequence \( S = S_1 \ldots S_n \)
  - \( \text{Pre}_i \) the set of preconditions for sequent \( S_i \)
  - \( \text{Post}_i \) the set of postconditions for sequent \( S_i \)
  - \( \text{Obli}_i \) the set of obligations for sequent \( S_i \)
  - \( \text{PreCeili} \) the preconditions ceilinged by sequent \( S_i \)
  - \( \text{ObliFloor}_i \) the obligations floored by sequent \( S_i \)
  - \( \text{State}_i \) the current state after sequent \( S_i \)
  - \( \text{Promised}_i \) the set of obligations outstanding after sequent \( S_i \)
  - \( \text{Needed}_i \) the accumulated set of unsatisfied preconditions up to and including sequent \( S_i \)
  - \( \text{SatPre}_i \) the satisfied preconditions for sequent \( S_i \)
  - \( \text{UnsatPre}_i \) the unsatisfied preconditions for sequent \( S_i \)
  - \( \text{SatObli}_i \) the satisfied obligations for sequent \( S_i \)
Sequents

- For the Sequence S there is an initial State_0 and Promised_0.
- For each Sequent S_i in Sequence S = S_1 .. S_n
  \[
  \begin{align*}
  \text{State}_i &= \text{State}_{i-1} + \text{seq Post}_i \\
  \text{SatPre}_i &= \{ P \in \text{Pre}_i | P \text{ is satisfied by State}_{i-1} \} \\
  \text{UnsatPre}_i &= \text{Pre}_i - \text{SatPre}_i \\
  \text{Needed}_i &= (\text{Needed}_{i-1} - \text{PreCeil}_i) \cup \text{UnsatPre}_i \\
  \text{SatObl}_i &= \{ O \in \text{Obl}_i | O \text{ is satisfied by State}_{i} \} \\
  \text{Promised}_i &= (\text{Promised}_{i-1} - \text{OblFloor}_i) \cup \text{UnsatObl}_i \\
  \text{OblFloor}_i &= \text{Promised}_{i-1} - \text{con Obl}_i
  \end{align*}
\]

Sequence Example

Let the Sequence S = S_1 .. S_n where State_0 and Promised_0 have been initialized according to the context of the sequence.

The interface for S is propagated as follows:
\[
\begin{align*}
  S.\text{Pre} &= \text{Needed}_1 \\
  S.\text{Post} &= \text{State}_n \\
  S.\text{Obl} &= \text{Promised}_n
  \end{align*}
\]

The content of S.\text{PreCeil} and S.\text{OblFloor} may be amended according to the context of the use of the sequence S

Selection (IF)

Let the Selection Statement S consist of BE = the boolean expression, T = the then sequence, and E = the else sequence where
\[
\begin{align*}
  T.\text{State}_0 &= \text{BE True}\text{.Post} \\
  E.\text{State}_0 &= \text{BE False}\text{.Post} \\
  T.\text{Promised}_0 &= \text{BE True}\text{.Obl} \\
  E.\text{Promised}_0 &= \text{BE False}\text{.Obl}
  \end{align*}
\]

The interface for S is propagated as follows:
\[
\begin{align*}
  S.\text{Pre} &= \text{BE Pre} \cup (T.\text{Pre} - T.\text{PreCeil}) \cup (E.\text{Pre} - E.\text{PreCeil}) \\
  S.\text{Post} &= T.\text{Post} + \text{par E}\text{.Post} \\
  S.\text{Obl} &= T.\text{Obl} \cap E.\text{Obl}
  \end{align*}
\]

The state of the selection statement S is amended as follows:
\[
\begin{align*}
  T.\text{OblFloor} &= T.\text{Obl} - S.\text{Obl} \\
  E.\text{OblFloor} &= E.\text{Obl} - S.\text{Obl} \\
  T.\text{PreCeil} &= (T.\text{Pre} - \text{con BE True}\text{.Post}) \cup (T.\text{Pre} - \text{con E}\text{.Pre}) \cup (T.\text{Pre} - \text{con B}\text{.Pre}) \\
  E.\text{PreCeil} &= (E.\text{Pre} - \text{con BE False}\text{.Post}) \cup (E.\text{Pre} - \text{con T}\text{.Pre}) \cup (E.\text{Pre} - \text{con B}\text{.Pre})
  \end{align*}
\]
Iteration [While]

Let the Iteration Statement \( W \) consist of \( BE = \) the boolean expression, and \( B = \) the loop body (a sequence) where

\[
\begin{align*}
B_{\text{State}}_0 & = BE_{\text{True}}.Post \\
B_{\text{Promised}}_0 & = BE_{\text{True}}.Obl
\end{align*}
\]

The interface for \( I \) is propagated as follows:

\[
\begin{align*}
W_{\text{Pre}} & = BE_{\text{Pre}} \cup (B_{\text{Pre}} - B_{\text{PreCeil}}) \\
W_{\text{Post}} & = (B_{\text{Post}} + \text{par } \emptyset) + \text{seq } BE_{\text{False}}.Post \\
W_{\text{Obl}} & = BE_{\text{False}}.Obl
\end{align*}
\]

The state of the loop body \( B \) is amended as follows:

\[
\begin{align*}
B_{\text{OblFloor}} & = B_{\text{Obl}} - I_{\text{Obl}} \\
B_{\text{PreCeil}} & = (B_{\text{Pre}} - \text{con } BE_{\text{True}}.Post) \\
& \quad \cup (B_{\text{Pre}} - \text{con } BE_{\text{Pre}}) \\
& \quad \cup (B_{\text{Pre}} - \text{con } B_{\text{Post}})
\end{align*}
\]

There is an Error when \( W_{\text{Pre}} - \text{con } B_{\text{Post}} \neq \emptyset \)

Iteration [Repeat]

Let the Iteration Statement \( R \) consist of \( BE = \) the boolean expression, and \( B = \) the loop body (a sequence) where

\[
\begin{align*}
B_{\text{State}}_0 & = BE_{\text{True}}.Post \\
B_{\text{Promised}}_0 & = BE_{\text{True}}.Obl
\end{align*}
\]

The interface for \( I \) is propagated as follows:

\[
\begin{align*}
R_{\text{Pre}} & = BE_{\text{Pre}} \cup (B_{\text{Pre}} - B_{\text{PreCeil}}) \\
R_{\text{Post}} & = B_{\text{Post}} + \text{seq } BE_{\text{False}}.Post \\
R_{\text{Obl}} & = B_{\text{Obl}} + \text{seq } BE_{\text{False}}.Obl
\end{align*}
\]

The state of the loop body \( B \) is amended as follows:

\[
\begin{align*}
B_{\text{OblFloor}} & = B_{\text{Obl}} - I_{\text{Obl}} \\
B_{\text{PreCeil}} & = (B_{\text{Pre}} - \text{con } BE_{\text{True}}.Post) \\
& \quad \cup (B_{\text{Pre}} - \text{con } BE_{\text{Pre}}) \\
& \quad \cup (B_{\text{Pre}} - \text{con } B_{\text{Post}})
\end{align*}
\]

There is an Error when \( R_{\text{Pre}} - \text{con } B_{\text{Post}} \neq \emptyset \)
Operation

- Let the Operation O have an implementation sequence S = S₁ .. Sₙ where S.State₀ = and S.Promised₀ = ∅.

- The interface for O is propagated as follows:
  - O.Pre = S.Pre - {P | P refers to local variables}
  - O.Post = S.Post - {P | P refers to local variables}
  - O.Obl = S.Obl - {P | P refers to local variables}

- The state of the sequence S is amended as follows:
  - S.PreCeil = S.Pre - O.Pre
  - S.OblFloor = S.Obl - O.Obl

Completeness & Correctness

- An implementation I = sequence S = S₁ .. Sₙ for an operation O is complete if and only if:
  - Every precondition P has either been satisfied or is in the interface of O. That is, all precondition ceilings (recursively) in S are empty.
  - Every obligation O has either been satisfied or is in the interface of O. That is, all obligation floors (recursively) in S are empty.
  - There are no iteration errors (that is, I.Pre -con B.Post = ∅)

- A propagated interface PI for operation O is correct with respect to the specified interface SI for operation O if and only if:
  - the implementation I for operation O is complete
  - the interfaces PI and SI are identical
    - PI.Pre = SI.Pre
    - PI.Post = SI.Post
    - PI.Obl = SI.Obl
  - Note: Redefinition of the propagated interface may be needed to cast it in terms of the specified interface.

Conclusions

- Have the building blocks for the synthesizing the interfaces for complex language statements
- Mechanisms are in place for extension to handling exceptions
- Working on the semantics of assignment: partly automatic, partly interactive
- Punt on expressions: interact with the programmer encapsulation in functions is a way out
- Working on the specification logic (SL) and how consistency determination can be strengthened and made efficient.

Error Handling
Error Handling

- 50 - 70% of code in a large system is error handling code
- Error handling code is a weak link
  - no theory, little methodology
  - the errors are not well understood
  - the error handling code is not well-tested
- 20% of interface errors
  - Based on MRs (modification reports) from the third release of UNIX RTR
  - 68.6% of all the MRs represented interface errors

Extensions to Hoare Specifications

- Hoare specifications provide the following paradigm for describing programs:
  - Preconditions (Program) Postconditions
- The Inscape paradigm of specification captures the notions of exceptions and obligations with the following single entrance, multi-exit specification:
  - Preconditions (Program)
  - Postconditions, Obligations
  - Postconditions, Obligations
- Instress distinguishes three kinds of preconditions: assumed, validated, and dependent preconditions.

Kinds of Assumptions

- Hoare states in “Programs are Predicates” (in Mathematical Logic and Programming Languages, Prentice-Hall, 1985):
  - If the assumptions are falsified, the product may break, and its subsequent (but not its previous) behaviour may be wholly arbitrary. Even if it seems to work for a while, it is completely worthless, unreliable, and even dangerous.
- Instress distinguishes three kinds of preconditions to indicate the different kinds of effects that these preconditions may have.
  - Assumed preconditions are those which are assumed to be true. Their falsification may indeed produce arbitrary behavior.
  - Validated preconditions are those which are tested for. Their falsification results in predictable results.
  - Dependent preconditions are those whose truth is dependent upon system circumstances. While the truth or falseness of the these conditions is unpredictable, the behavior of the program itself is predictable.
Formalization of Exception Handling

- **precluded** - The associated precondition has been satisfied — there is no need to handle the exception.
- **pruned** - A refusal to handle the exception. The associated preconditions become **assumed** preconditions that must be satisfied.
- **reported** - The exception is propagated, possibly with some repair, to the interface.
- **recovered** - The exception is handled by retrying the operation that caused the exception. There may be some repair to increase the likelihood of success.
- **repaired** - The results of the exception are fixed, or compensated for, in some fashion — for example, fixed to match the successful results. Control flow then proceeds in the same fashion as the successful case.
- **ignored** - The results of the exception are satisfactory. The exception is treated as a successful result.
Examples of Exception Handling

- If \( \text{RecordExists}(R) \) is known to be true, then the handling of \( \text{RecordNonexistent}(R) \) is precluded.
- If it is not known to be true, the most obvious course is to report (ie, propagate) it to the caller.
- One possible way to handle the \( \text{RecordInconsistent}(R) \) exception is to fix the record (use the repair handling form) by means of the recovery routine and then continue.
- If the damage to the record is unimportant, then ignore the exception and treat the result as successful.

### Summary

- **Forward Error Recovery**
- **Instress**
  - explicit specification of each exception
  - explicit specification of minimal handling
  - pragmatic information about severity and recovery
  - method for determining exceptions
- **Inform/Infuse**
  - formalization of exception handling
  - enforced handling of exceptions
  - knowledgeable about relations between preconditions and exceptions
  - automatic construction of coherent interface (with exceptions) from the implementation