Objectives and outline

◆ Provide the foundation to represent different semantic domains for the Metropolis intermediate format
◆ Study the problem of heterogeneous interaction
◆ Formalize concepts such as abstraction and refinement
**An example of interaction**

- Combine a synchronous model with a dataflow model

  - **Synchronous model**
    - Total order of event
  
  - **Data flow model**
    - Partial order of events

  - **Discrete Time model**
    - Metric order of events


---

**An example of heterogeneous interaction**

- The interaction is derived from a common refinement of the heterogeneous models

- The resulting interaction depends on the particular refinements employed

- Our objective is to derive the consequences of the interaction at the higher levels of abstraction
**Data flow model**

- Assume signals take values from a set $V$
- Each signal is a sequence from $V$ (an element of $V^*$)
- Let $A$ be the set of signals
- One behavior is a function
  - $f : A \rightarrow V^*$
- An data flow agent is a set of those behaviors

**Synchronous model**

- Signals are again sequences from $V$ (elements of $V^*$)
- But are synchronized
- One element of the sequence is $g : A \rightarrow V$
- One behavior is a sequence of those functions
  - $\langle g \rangle \in (A \rightarrow V)^*$
- A synchronous agent is a set of those sequences
**Discrete Time model**

- Assume time is represented by the positive integers $\mathbb{N}$
- Then define a behavior
  \[ h: \mathbb{N} \rightarrow (A \rightarrow V) \]
- A discrete time agent is a set of those functions

**Discrete to Synchronous abstraction**
**Discrete to Data flow abstraction**

**Interaction Propagation**

1. Refinement
2. Composition
3. Projection
4. Abstraction
**Key points**

- The outlined technique defines the effects of the interaction
- The result depends on
  - The notion of composition at the refined level
  - The particular abstraction and refinement
- We can't define the interaction uniquely!
  - Note: both Synchronous and Data flow are untimed sequences
  - Could just have equated them
- How can we generalize? Need a formal approach to this problem

**Maintaining consistency across refinement**

\[ \Phi_C, \Phi_A, \text{and } \Phi'_C \text{ must be consistent, together with the specifications} \]

\[ \Phi'_C : \{ x = y \} \]
Trace algebras and Trace Structures algebra

Let $T_{\text{spec}}$ and $T_{\text{impl}}$ be trace structures in $A$. Then

$$\text{if } \psi'_u(T_{\text{impl}}) \subseteq \psi'_l(T_{\text{spec}}) \text{ then } T_{\text{impl}} \subseteq T_{\text{spec}}$$

Trace Algebra

- Let $W$ be a set of signals and $A$ a subset of $W$
- A trace algebra is a set of traces, each taking symbols from $A$, with operations of projection and renaming
- Formally, a trace algebra $C_C$ is a triple $(B_C, \text{proj}, \text{rename})$ where
  - for each $A$, $B_C(A)$ is a non-empty set, called the set of traces over $A$. Let $B_C$ also be the set of all traces, i.e. the union over all subsets $A$ of $W$ of the traces $B_C(A)$.
  - for $B \subseteq A$, $\text{proj}(B)$ is a function from $B_C$ to $B_C$
  - for renaming function $r: W \to W$, $\text{rename}(r)$ is a function from $B_C$ to $B_C$

Trace Algebra

- A trace need not be a sequence. Any set for which “appropriate” projection and renaming functions are defined can be used as a trace.

- The meaning of the operations of projection and renaming is defined by a set of axioms.
  - Intuitively, the function $\text{proj}(B)$, for $B$ a subset of $A$, takes a trace $x$ and produces a trace $y$ where the symbols not in $B$ are dropped. This can be used to hide internal signals in the process of a composition.
  - The function $\text{rename}(r)$, where $r: W \to W$ is a bijection, renames the elements of a trace $x$. This function corresponds to the process of instantiation.

Axioms

- T1. $\text{proj}(B)(x)$ is defined iff there exists an alphabet $A$ such that $x \in A$ and $B \subseteq A$. When defined, $\text{proj}(B)(x)$ is an element of $B_C(B)$.
- T2. $\text{proj}(B)(\text{proj}(B')(x)) = \text{proj}(B)(x)$
- T4. Let $x \in B_C(A)$ and $x' \in B_C(A')$ be such that $\text{proj}(A \cap A')(x) = \text{proj}(A \cap A')(x')$. For all $A''$ where $A \cup A' \subseteq A''$ there exists $x'' \in B_C(A'')$ such that $x = \text{proj}(A)(x'')$ and $x' = \text{proj}(A')(x'')$.
- T5. $\text{rename}(r)(x)$ is defined iff $x \in B_C(\text{dom}(r))$. When defined $\text{rename}(r)(x)$ is an element of $B_C(\text{codom}(r))$. 

June 18, 2001
Example

- For every alphabet $A$ over $W$, $B_{c}(A)$ is the set $A^{\infty}$.
- $\text{proj}(B)(x)$ is the sequence formed from $x$ by removing every symbol $a$ not in $B$.
- $\text{rename}(r)(x)$ is the sequence formed from $x$ by renaming every symbol $a$ in $x$ according to $r$.
- Must prove the axioms of trace algebra
- Traces are not necessarily sequences
  - Let $B_{c}(A) = 2^{A}$

Trace Structures Algebra

- Let $C_{c} = (B_{c}, \text{proj}, \text{rename})$ be a trace algebra over $W$. A trace structure $T$ is a pair $(A, P)$ where $P \subseteq B_{c}(A)$.
- Let $TS$ be a subset of the trace structures. Then $A_{c} = (C_{c}, TS)$ is a trace structure algebra if $TS$ is closed under parallel composition, projection and renaming.
- Parallel Composition:
  - $T_{1} \parallel T_{2} = \{ x : \text{proj}(A_{1})(x) \in T_{1} \land \text{proj}(A_{2})(x) \in T_{2} \}$
Conservative approximations

- **Objective**
  - To translate a problem in one domain into a similar (but more tractable problem) in another domain
  - Ensure that false positives do not occur

- Let $A_C = (C_C, TS)$ and $A'_C = (C'_C, TS')$ be trace structure algebras and consider two functions $\Psi_u$ and $\Psi_f$ from $TS$ to $TS'$. We say that $\Psi = (\Psi_u, \Psi_f)$ is a conservative approximation if $\Psi_u(T_1) \subseteq \Psi_f(T_2)$ implies that $T_1 \subseteq T_2$.
Conservative Approximations

Homomorphisms on Trace Algebras

- Let $C$ and $C'$ be trace algebras. Let $h$ be a function from $B_C$ to $B'_C$ such that if $x \in B_C(A)$ then $h(x) \in B'_C(A)$. The function $h$ is a homomorphism iff
  - $h(proj(B)(x)) = proj(B)(h(x))$
  - $h(rename(r)(x)) = rename(r)(h(x))$

- Example: from $A^n$ to $2^A$
  - $h(x) = \{ a : \exists n \in \mathbb{N} \mid a = x(n) \}$
Approximations induced by homomorphisms

- If \( x' = h( x ) \) then intuitively \( x' \) is an abstraction of any trace \( y \) such that \( h( y ) = x' \). Then \( x' \) represents all such \( y \).
- Then define \( \Psi = ( \Psi_u, \Psi_i ) \) as follows
  - \( \Psi_u( T ) = h( T ) \)
  - \( \Psi_i( T ) = h( T ) - h( B_c( A ) - T ) \)
- Note that intuitively \( \Psi_u^{-1}( T' ) \supseteq T \supseteq \Psi_i^{-1}( T' ) \).
- Theorem: \( \Psi \) is a conservative approximation
- If \( \Psi_u^{-1}( T' ) = T = \Psi_i^{-1}( T' ) \) then we can define an inverse \( \Psi^{-1} \)

Trace algebras and Trace Structures algebras

Let \( T_{\text{spec}} \) and \( T_{\text{impr}} \) be trace structures in \( A \). Then

if \( \Psi_u( T_{\text{impr}} ) \subseteq \Psi_i( T_{\text{spec}} ) \) then \( T_{\text{impr}} \subseteq T_{\text{spec}} \)
**Inverses of Conservative Approximations**

![Diagram showing inverses of conservative approximations]

**Parallel composition**

- Example based on the theory of trace structures
- Parallel composition defined in each domain in terms of a projection operation
  - Data flow: $f : A \rightarrow V^*$, $\text{proj}(B)(f) = f_{|B}$
  - Synchronous: $<g> \in (A \rightarrow V)^*$, $\text{proj}(B)(<g>) = <g_{|B}>$
  - Discrete: $N \rightarrow (A \rightarrow V)$, $\text{proj}(B)(N \rightarrow g) = N \rightarrow g_{|B}$
- For each domain: $T_1 \parallel T_2 = \{ x : \text{proj}(A_1)(x) \in T_1 \land \text{proj}(A_2)(x) \in T_2 \}$
Conclusions

- We are defining semantics domains for the representation of various models of computation
- Using the formal techniques to study heterogeneous interaction
- Work in progress in generalizing the foundation layer to cover representation of different aspects

Basic elements: a model

- A model is a representation of an entity (an object or an idea)
- In the previous example we use mathematical structures as representations
- In our approach we use the theory of a structure as a logical representation
  ▲ Theory of a structure: the set of true statements about the structure in a logic
- A model is a set of properties that must be satisfied by the represented object
  ▲ Neutral with respect to representation
**Classes of models and local refinement**

- **A class of models** is represented by the set of properties $\Phi$ common to all models.
- **A model $\Psi$** belongs to a class $\Phi$ if and only if $\Psi \models \Phi$.
  - This notion corresponds to local refinement.
  - $\Psi$ must have all the properties of $\Phi$ plus (possibly) some more.
- **For example if $\Phi$** is the class of models with a partial order, then a total order $\Psi$ belongs to the class $\Phi$.

**Inter-class Refinement**

- **Let $P$ and $Q$** be two classes of models.
- **Define when elements $p \in P$ and $q \in Q$** represent the same underlying object.
- **Bipartite equivalence (or correspondence)**
  - Let $\Phi_\Lambda$ be a set of assertions that defines a notion of correspondence. We say that $p$ and $q$ are bipartite equivalent if and only if their disjoint union satisfies $\Phi_\Lambda$.
- **The theory $\Phi_\Lambda$** outlines what must be true in order for two heterogeneous models to represent the same entity.
**Example: abstraction of time**

- Abstract time away. For all properties $\varphi$
  
  $\Phi_A$: $\forall x (\varphi(x) \iff \forall t \varphi(f(x,t)))$
  
  $\Phi_A'$: $\forall x (\varphi(x) \iff \exists t \varphi(f(x,t)))$

---

**Abstraction or refinement?**

- $q$ is a refinement of $p$ if $q$ knows everything about $p$
  
  ▲ As in local refinement, we want $q \models p$

- But in order to do that we need the information on the bipartite equivalence
  
  ▲ $(q \cup \Phi_A) \models p$

- Can be extended to classes of models
  
  ▲ $(\Phi_Q \cup \Phi_A) \models \Phi_P$

- Transitive and reflexive relation: a pre-order
**Constraints**

- A constraint is a property that must hold, but that is derived from a corresponding model at a different level of abstraction.

- Let $P$, $Q$ be classes of models identified by specifications $\Phi_P$ and $\Phi_Q$. Let $\Phi_A$ be a theory of equivalence.

- Theorem: If $p \in P : p \not\models (\Phi_Q \cup \Phi_A)$, then there is no model $q \in Q$ such that $p$ and $q$ are equivalent.

- Hence we define the constraints of $Q$ over $P$, mediated by equivalence $\Phi_A$, as the consequence closure:

  $$\Psi = (\Phi_Q \cup \Phi_A)$$

---

**Interaction as a constraint application**

- A special case of constraint application.

- A theory $\Phi_C$ that defines how a set of properties is translated into another set of properties.

- Essential for heterogeneous systems (no notion of parallel composition).

- But also useful for homogeneous systems:

  - synchronous vs. asynchronous automata composition.
Assume-Guarantee for executable models

- For a class of executable models, declare the local model of computation:
  - For each model, declare the properties of data communication
  - For each model, declare the properties of activations
- The less stringent the properties, the more reusable the component

Properties are divided into requirements (assumptions) and guarantees

An explicit scheduler (or director) can be used to embed the resolution of assumption and guarantees

A collection of components must mutually satisfy their requirements through the use of guarantees
**Conclusions**

◆ By providing a common formal framework for heterogeneous systems we address

▲ The problem of maintaining consistency through the refinement process and the design flow

▲ A way to consciously reason about the interaction between heterogeneous models

◆ This provides a more precise verification that lowers time to market and increases productivity

---

**Conjoint structures**

◆ A way to talk about two structures at the same time

◆ Given structures A and B, in languages S and S’, consider the disjoint union U in language S”

◆ In general it is not the case that if $A \models \varphi$ then $U \models \varphi$

◆ However, let $\varphi'$ be $\varphi$ where all variables are constrained to range over the domain of A

◆ Theorem: $A \models \varphi$ if and only if $U \models \varphi'$

▲ By induction and by the definition of truth
**A special case: Executable models**

- Executable models are those that can be run as a simulation
- Characterized by a semantics of internal execution and of interaction
- The interaction is composed of
  - the way the data is exchanged
  - the relationships between the data transfers and the activations

---

**Example of requirements**

- Data flow: define a partial order
  - For each activation, a sufficient amount of data must been seen at the inputs in the past
- Synchronous: define a total order
  - For each activation, corresponding data must be seen at the same time at the inputs
- Synchronous guarantees satisfy data flow requirements
- Sub-type (refinement) when \( R \Rightarrow R' \) and \( G' \Rightarrow G \)
- Contravariant as in type systems
**Representation of properties**

- **serial**
  - 0-
  - 1a
  - 0b

- **handshake**
  - 0-
  - 1a
  - 0b

**Derivation of requirements**

- 0- 1a 0b
- 0- 1a 0b
- 0- 1a 0b
- 0- 1a 0b
Derivation of requirements

The satisfaction case
Choosing the model

Choosing the appropriate model is essential because

▲ If the model is too detailed:
  ▼ over-specification (miss opportunity for optimization)
  ▼ complexity (must deal with too many details)
  ▼ difficult to analyze (may not be possible to extract relevant properties)

▲ If the model is too abstract:
  ▼ under-specification
  ▼ unwanted non-determinism

◆ ⇒ Can’t really just do with one!
  ▲ But a common infrastructure is necessary
Domain transformation

Convert a representation from one model to another while preserving the properties of interest

◆ Classes of transformations
  ▲ Property preserving transformation (homomorphism)
  ▲ Identity preserving transformation (injective)
  ▲ Abstracting transformation (strictly non-injective)
  ▲ Non-deterministic transformation (relation: one-to-many)
  ▲ Sub-typing transformation (embedding)