Ten Virtues of Structured Graphs
(or why structured graphs can be better than flat ones)

Roberto Bruni
also based on joint work with F. Gadducci, A. Lluch-Lafuente, U. Montanari, E. Tuosto

Department of Computer Science, University of Pisa
bruni@di.unipi.it

Research supported by

March 28-29, 2009, GT-VMT’09, York, UK
Outline

Introduction
Styles for Visual Support
Dynamics
ADR
Concluding Remarks
GT-VMT 2009

- **Graph-based techniques**
  - formal semantics, concurrency, logics, verification, tools

- **Visual modelling**
  - project planning, network management, traffic control, business processes, software architectures, www site design, and many more...

- **Modern software and Sensoria project (service-oriented computing)**
  - key issues such as scalability, representation distance, open-endedness, dynamicity, distribution
  - within specification, design, validation and verification phases
Sensoria Poster Collage (http://www.sensoria-ist.eu)

Software Engineering for **Service-Oriented** Overlay Computers  www.sensoria-ist.eu

develops
semantically well-founded languages, novel theories, methods and tools for constructing and analysing the new generation of high-quality service-oriented systems

integrates
foundational theories, techniques, and methods with pragmatic software engineering

researches
- linguistic primitives for modelling and programming service-oriented systems
- qualitative and quantitative analysis methods for global services
- development and deployment techniques for systems services

offers
- model-driven approach for service-oriented software engineering
- modelling of service-oriented systems
- analysis of behaviour, security and quality of service properties
- suite of tools and techniques for
  - deploying service-oriented systems
  - reengineering legacy software into services

case studies
in automotive, finance, telecommunications and e-learning domains

List of partners
Co-ordinator: Prof. Dr. Martin Wirsing, Ludwig Maximilians-Universität München, Germany
Universita di Trento | University of Leicester | Warsaw University | TU Denmark at Lyngby | Università di Pisa
Università di Firenze | Università di Bologna | ISTI Pisa | Universidade de Lisboa | University of Edinburgh | ATX !
Telecom Italia Lab | Imperial College London | FAST GmbH | Budapest University of Technology and Economics
S&N AG | University College London | Politecnico di Milano

Ten Virtues of Structured Graphs (GT-VMT’09) 4/45
### All That Graphs

<table>
<thead>
<tr>
<th>( n_0 )</th>
<th>( n_1 )</th>
<th>( n_2 )</th>
<th>( n_3 )</th>
<th>( n_4 )</th>
<th>( n_5 )</th>
<th>( n_6 )</th>
<th>( n_7 )</th>
<th>( n_8 )</th>
<th>( n_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_0 )</td>
<td>( a_1 )</td>
<td>( a_2 )</td>
<td></td>
<td></td>
<td>( a_3 )</td>
<td>( a_4 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_1 )</td>
<td></td>
<td>( a_5 )</td>
<td>( a_6 )</td>
<td>( a_7 )</td>
<td></td>
<td></td>
<td>( a_8 )</td>
<td>( a_9 )</td>
<td>( a_{10} )</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( a_{11} )</td>
<td></td>
<td>( a_{12} )</td>
<td></td>
<td>( a_{13} )</td>
<td></td>
<td>( a_{14} )</td>
<td>( a_{15} )</td>
<td></td>
</tr>
<tr>
<td>( n_3 )</td>
<td>( a_{16} )</td>
<td>( a_{17} )</td>
<td>( a_{18} )</td>
<td>( a_{19} )</td>
<td>( a_{20} )</td>
<td>( a_{21} )</td>
<td>( a_{22} )</td>
<td>( a_{23} )</td>
<td>( a_{24} )</td>
</tr>
<tr>
<td>( n_4 )</td>
<td></td>
<td></td>
<td></td>
<td>( a_{25} )</td>
<td></td>
<td>( a_{26} )</td>
<td>( a_{27} )</td>
<td>( a_{28} )</td>
<td></td>
</tr>
<tr>
<td>( n_5 )</td>
<td></td>
<td>( a_{29} )</td>
<td>( a_{30} )</td>
<td></td>
<td></td>
<td>( a_{31} )</td>
<td>( a_{32} )</td>
<td></td>
<td>( a_{33} )</td>
</tr>
<tr>
<td>( n_6 )</td>
<td>( a_{34} )</td>
<td></td>
<td>( a_{35} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_7 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( a_{36} )</td>
<td>( a_{37} )</td>
<td></td>
<td></td>
<td>( a_{38} )</td>
</tr>
<tr>
<td>( n_8 )</td>
<td></td>
<td></td>
<td>( a_{39} )</td>
<td>( a_{40} )</td>
<td></td>
<td>( a_{41} )</td>
<td>( a_{42} )</td>
<td></td>
<td>( a_{43} )</td>
</tr>
<tr>
<td>( n_9 )</td>
<td>( a_{44} )</td>
<td>( a_{45} )</td>
<td>( a_{46} )</td>
<td>( a_{47} )</td>
<td>( a_{48} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All That Graphs
All That Graphs

Our choice

Ten Virtues of Structured Graphs (GT-VMT’09) 5/45
A Scenario: Software Architectures as Graphs

D. Garlan & D. Perry, 1995

“... the structure of the components of a program / system, their interrelationship, and principles and guidelines governing their design and evolution over time.”

- components (and connectors) as hyper-edges
  - (here represented as boxes of various shapes)
- ports (and roles) as tentacles
  - (here represented as arrows)
- attachments as nodes
  - (here represented as smaller circles)
- connectors and attachments are sometimes omitted
Why “Spaghetti” Graphs are Considered Harmful

- When GT applied to large case studies, graphs better be structured in order to be comprehensible
- Analogies with structured programming and type theory
  - it is helpful to use graphs that are conveniently formatted and annotated
  - discard / ignore non-conformant graphs
- Analogies with process calculi
  - containment and links (as in bigraphs)
  - dynamics and reconfiguration via inductive, conditional rewrite rules
Our proposal

- From graphs to **hierarchical hypergraphs**
  - certain hyperedges can contain hypergraphs that can be hierarchical themselves
  - arbitrary depth of nesting
- **ADR (Architectural Design Rewriting)**
  - graphs + their **blueprint** (like binaries + source templates)
  - exploit blueprint for applying formal methods
  - please visit [http://www.albertolluch.com/research/adr](http://www.albertolluch.com/research/adr) to know more
Outline

Introduction

Styles for Visual Support

Dynamics

ADR

Concluding Remarks
Visualization can Support Formal Methods
IEEE standard 1471

“... a set of patterns or rules for creating one or more architectures in a consistent fashion.”

Style = Vocabulary + Rules

- Used to construct and document
- Used to describe / explain
- Used to understand
- Used to validate
- Used for conformance check
- Used to reason about
- To be reused
Can you spot some “regularity”? 
Graph Re-drawing

And now?
Well...

Another try?
Another Graph Re-drawing

Can you describe its “shape” (or style)?
Styles from Productions

- Legenda: titled boxes as non-terminals, ordinary boxes as terminals

- Several readings are possible:
  - Refinement
  - Types (Pipeline) and ops (station and cat(·), based on hyperedge replacement)
    - station $\rightarrow$ Pipeline
    - cat : Pipeline $\times$ Pipeline $\rightarrow$ Pipeline
  - Abstraction
Types for Pipelines, Rings and Stars
Types and Ops for Pipelines, Rings and Stars

- $\text{cat : Pipeline Pipeline -> Pipeline}$
- $\text{net : Star -> Pipeline}$
- $\text{station : -> Pipeline}$
- $\text{par : Star Star -> Star}$
- $\text{cast : Ring -> Star}$
Simplified Memberships (for Pipelines and Stars)
An Example of Derivation (with "Blueprint")
An Example of Derivation (with “Blueprint”)

[Diagram of a pipeline with nodes labeled 'Pipeline' and 'Star']

Ten Virtues of Structured Graphs (GT-VMT'09) 20/45
An Example of Derivation (with “Blueprint”)

![Diagram of Pipeline and Star relationships]
An Example of Derivation (with “Blueprint”)
An Example of Derivation (with “Blueprint”)
An Example of Derivation (with “Blueprint”)

[Diagram showing a derivation process with nodes labeled Pipeline, Star, Ring, and connections between them.]

Ten Virtues of Structured Graphs (GT-VMT'09) 20/45
An Example of Derivation (with “Blueprint”)
An Example of Derivation (with “Blueprint”)
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")
Simplified Typing and Drawing ("Flattening")

The corresponding proof term is

\[
\text{net}(\text{par}(\text{cast}(\text{node}(\text{cat}(\text{station},
\text{cat}(\text{station}, \text{station})))),
\text{cast}(\text{node}(\text{station}))))
\]

Or just

\[
\text{net}(\text{par}(\text{node}(\text{cat}(\text{station}, \text{station}, \text{station})),
\text{node}(\text{station})))
\]

Note that nodes need not be mentioned.
Another Example: Workflows

Activities composable in series and in parallel (fork & join): disconnected activity and cyclic parts are not allowed
Another Example: Workflows

Is this a well-formed workflow?
Another Example: Workflows

Activities composable in series and in parallel (fork & join): disconnected activity and cyclic parts are not allowed
Another Example: Workflows

Is this a well-formed workflow?
Six Virtues of Structured Graphs

- **Requirements**
  - Type graphs are ok (and synergic to our approach) but limited
  - Additional logic languages often needed
  - We can account for many patterns in a natural way

- **Parsing and browsing**
  - Large graphs are hard to “understand” and navigate
  - Their blueprint (if any available) helps quite a lot

- **Model Construction and Model conformance**
  - Conformance is guaranteed by construction
  - Otherwise hard to recover from scratch (proof-carrying graphs)

- **Compositionality and Abstraction & Refinement**
  - Interfaces are needed to constrain composition, but hard to recover in flat graphs
  - The hierarchical approach makes them available at any level
  - Different levels of granularity can be considered
Six Virtues of Structured Graphs

- **Requirements**
  - Type graphs are ok (and synergic to our approach) but limited
  - Additional logic languages often needed
  - We can account for many patterns in a natural way

- **Parsing and browsing**
  - Large graphs are hard to “understand” and navigate
  - Their blueprint (if any available) helps quite a lot

- **Model Construction and Model conformance**
  - Conformance is guaranteed by construction
  - Otherwise hard to recover from scratch (proof-carrying graphs)

- **Compositionality and Abstraction & Refinement**
  - Interfaces are needed to constrain composition, but hard to recover in flat graphs
  - The hierarchical approach makes them available at any level
  - Different levels of granularity can be considered
Six Virtues of Structured Graphs

- **Requirements**
  - Type graphs are ok (and synergic to our approach) but limited
  - Additional logic languages often needed
  - We can account for many patterns in a natural way

- **Parsing and browsing**
  - Large graphs are hard to “understand” and navigate
  - Their blueprint (if any available) helps quite a lot

- **Model Construction and Model conformance**
  - Conformance is guaranteed by construction
  - Otherwise hard to recover from scratch (proof-carrying graphs)

- **Compositionality and Abstraction & Refinement**
  - Interfaces are needed to constrain composition, but hard to recover in flat graphs
  - The hierarchical approach makes them available at any level
  - Different levels of granularity can be considered
Six Virtues of Structured Graphs

- **Requirements**
  - Type graphs are ok (and synergic to our approach) but limited
  - Additional logic languages often needed
  - We can account for many patterns in a natural way

- **Parsing and browsing**
  - Large graphs are hard to “understand” and navigate
  - Their blueprint (if any available) helps quite a lot

- **Model Construction and Model conformance**
  - Conformance is guaranteed by construction
  - Otherwise hard to recover from scratch (proof-carrying graphs)

- **Compositionality and Abstraction & Refinement**
  - Interfaces are needed to constrain composition, but hard to recover in flat graphs
  - The hierarchical approach makes them available at any level
  - Different levels of granularity can be considered
Outline

Introduction

Styles for Visual Support

Dynamics

ADR

Concluding Remarks
Style-Preserving Reconfiguration

- A reconfiguration is a change in an architecture
  - static? e.g. for deployment on different platforms, improvements, updates, upgrades, model-driven transformation
  - partially specified? e.g. some components are not known at design time, except for their types
  - run-time? e.g. triggered by security policies, load balancing, mobility, QoS assurance, components joining and leaving the system, dynamic binding, wrapping, self-* architectures

- Style-preservation is relevant
  - from well-formed graphs to well-formed graphs (but possibly with different shapes)

- Examples
  - reverse all actions in a pipeline, serialize a workflow, star to ring transformation, migrate all clients of a server, close all sub-sessions upon termination of their parents
How to Write Reconfiguration Rules

- **Using graph transformation**
  - direct manipulation of flat graphs
  - applicable in non well-formed graphs
  - well-formedness of results must be proved
  - in the flat case: rules manipulate components (many steps required)
  - in the hierarchical case: rules manipulate groups of components (one step can suffice)

- **Exploiting structured graphs**
  - rules manipulate well-formedness proofs
  - inductive localization of the least part of the proof where the change is needed
  - style-preserving by construction
An Example: 3hub Network

Network hubs have three degrees of connectivity and connections are driven by the style (only allowed: some sort of reversed pyramids)
An Example: 3hub Network

A valid 3hubs network
An Example: 3hub Network

A valid 3hubs network? or maybe not?
An Example: 3hub Network

A valid 3hubs network? or maybe not?
An Example: 2hub Network

Network hubs have just two degrees of connectivity and connections are driven by the style (only allowed: rings)
An Example: 2hub Network

A valid 2hubs network
An Example: From 3hub Networks to 2hub Networks

- Under certain circumstances, it is required to reconfigure any valid 3hub network to a valid 2hub network
  - the whole network must be reconfigured (not just part of it)
  - total number of hubs is unchanged
  - 2hubs must form a ring

- Idea:
  - exploit blueprint, not the flat graph
  - reconfiguration is defined inductively on the structure of the network
  - exploit conditional rewrite rules
An Example: From 3hub Networks to 2hub Networks

Reconfigure a single 3hub (note that type is changed: some sort of transduction, context must be adapted)
An Example: From 3hub Networks to 2hub Networks

Reconfigure the link structure (a transduction, again)
An Example: From 3hub Networks to 2hub Networks

Reconfigure the whole network (note that type is preserved, rewrite is silent, applicable in any larger context)
An Example: Rewrite Rules for Network Transformation

\[
\text{3hub} \xrightarrow{\text{3to2}} \text{2hub}
\]

\[
\begin{align*}
&x_1 \xrightarrow{\text{3to2}} y_1 & &x_2 \xrightarrow{\text{3to2}} y_2 & &x_3 \xrightarrow{\text{3to2}} y_3 \\
&\text{3link}(x_1, x_2, x_3) \xrightarrow{\text{3to2}} \text{2link}(y_1, \text{2link}(y_3, y_2)) \\
&x \xrightarrow{\text{3to2}} y \\
&\text{3net}(x) \xrightarrow{} \text{2net}(y)
\end{align*}
\]
Three More Virtues of Structured Graphs

- **Reconfiguration and Evolution**
  - (flat) graph transformation requires ad-hoc studies and techniques (e.g., negative application conditions, interfaces, atomicity issues), augmenting the representation distance (high expertise, technology transfer more difficult)
  - structured graph rewrites can be more handy and efficient (e.g. graph matching not necessarily required)
  - style preservation: to be proved vs guaranteed by proofs
  - concurrency? special cases (edge to edge rules)?

- **Graphical encoding**
  - seamless grouping of item through the hierarchy (e.g. for representing nested sessions, transactions, scopes)
  - in the case of process calculi, facilitated by suitable graph algebras (see next part of the talk)
  - Encoding properties (soundness, completeness) shown by structural induction
Three More Virtues of Structured Graphs

- **Reconfiguration and Evolution**
  - (flat) graph transformation requires ad-hoc studies and techniques (e.g., negative application conditions, interfaces, atomicity issues), augmenting the representation distance (high expertize, technology transfer more difficult)
  - structured graph rewrites can be more handy and efficient (e.g. graph matching not necessarily required)
  - style preservation: to be proved vs guaranteed by proofs
  - concurrency? special cases (edge to edge rules)?

- **Graphical encoding**
  - seamless grouping of item through the hierarchy (e.g. for representing nested sessions, transactions, scopes)
  - in the case of process calculi, facilitated by suitable graph algebras (see next part of the talk)
  - Encoding properties (soundness, completeness) shown by structural induction
Outline

Introduction

Styles for Visual Support

Dynamics

ADR

Concluding Remarks
ADR in a Nutshell

ADR formulas:
- \( ADR = \text{Designs} + \text{Term Rewriting} \)
- Designs = Typed Hierarchical Graphs (with Interfaces)

ADR ingredients:
- **Sorts**: Vocabulary, Types (edge and node labels)
- **Values**: Designs (hierarchical graphs with interfaces)
- **Operations**: Graph-grammar-like rules
- **Terms**: proofs of construction
- **Terms (with variables)**: partial Designs, partial proofs
- **Axioms**: properties of operations
- **Membership predicates**: additional style rules
- **Rewrite rules**: behaviour, reconfigurations
- **Rewrite strategies**: style conformance, style analysis, etc.
A Flexible Unifying Framework for Design, Execution, Reconfiguration

- Not necessarily in the spirit of universal models:
  - node as names + hyper-edge as ops + parallel composition + name fusion + name hiding = any graph can be obtained
  - node as names + hyper-edge as ops + type annotation + tailored constructors = only well-formed designs are described
- Some other ADR features:
  - Membership equational theory (e.g. ACI1, subsorting, overloading)
  - Flattening axioms (e.g. not all operators are hierarchic)
- Some ADR caveats:
  - different proof terms for the same graph are possible
  - constraints not fully integrated yet
  - concurrency aspects not addressed yet
Maude Prototype for ADR

- Basic Modules
  - Maps, Sets, Lists, etc.

- Algebra of Hierarchical Designs
  - Graphs, Graph Morphisms, etc.
  - Hierarchical Designs

- Other ADR Algebras

- Other Scenarios
  - Spam Filter Scenario
    - Symbolic Modules
    - Design-Interpreted Modules

- Exporting Modules
  - designs2dot
  - designs2gml
Maude Prototype for ADR

Why Maude?

- built-in membership equational theories (e.g. to support style conformance check)
- conditional rewrite rules supported
- standard encoding of LTS
- built-in search strategies (e.g. to support model finding)
- built-in LTL model-checker
- defineable logic languages (within the same framework): e.g. graph logics (Courcelle’s MSO), modal logics, spatial logics
ADR Case Studies

Leg-o-motive Case Study

Network Topologies

Dynamics

Architectural Styles

Process Algebras

Service Modelling Languages
An Example: From Process Calculi to Graphs

The syntax of process calculi (with name handling)

\[ P, Q ::= \sum_{i \in I} \pi_i P_i \quad \text{Guarded Sum} \]

\[ | s.P \quad \text{Service Definition} \]

\[ | \overline{s}.P \quad \text{Service Invocation} \]

\[ | r > P \quad \text{Session} \]

\[ | P > Q \quad \text{Pipeline} \]

\[ | P | Q \quad \text{Parallel Composition} \]

\[ | (\forall n)P \quad \text{Restriction} \]

\[ | !P \quad \text{Replication} \]

Algebraic form:
- grammar
- structural congruence
An Example: From Process Calculi to Graphs

Terms as graphs

syntax trees, term graphs, bigraphs, etc.
An Example: From Process Calculi to Graphs

The syntax of graphs

Definition 22 (bigraph) A bigraph over the signature $\mathcal{K}$ takes the form $G = (V, ctrl, G^T, G^M) : I \rightarrow J$ where $I = (m, X)$ and $J = (n, Y)$ are its inner and outer interfaces, each combining a width (a finite ordinal), or each and

Definition 7 (hypergraph morphisms). A hypergraph $G$ is a triple $\langle E_G, N_G, t_G \rangle$ such that $E_G$ is the set of edges, $N_G$ is the set of nodes, and $t_G : E_G \rightarrow N_G^*$ is the function of

Definition 2.1 (Graph term). A graph term has form $\Gamma \vdash G$ where:

1. $\Gamma \subseteq \mathcal{N}$ is a finite set of names (the free nodes of the graph);
2. $G$ is a graph term generated by the grammar

$$G ::= L(x) \mid G|G \mid vy\ G \mid nil$$

where $x$ is a tuple of names, $L \in L$, rank($L$) = $|x|$ and $y$ is a name.
An Example: From Process Calculi to Graphs

Encoding can become cumbersome

\[
\llbracket (\nu a)P \rrbracket_G = \begin{cases} 
\llbracket P \rrbracket_G & \text{if } a \notin \text{fn}(P) \\
(\text{id}_P \otimes \nu_c \otimes \text{id}_G) \circ \llbracket P \rrbracket_G \otimes \llbracket \{c/a\} \rrbracket_G & \text{otherwise}
\end{cases}
\]

\[
\llbracket P \mid Q \rrbracket_G = \llbracket P \rrbracket_G \otimes \llbracket Q \rrbracket_G
\]

\[
\llbracket 0 \rrbracket_G = 0 \otimes 0
\]

\[
\llbracket (a(b))P \rrbracket_G = (\text{in}_{a,c} \otimes \text{id}_G) \circ \llbracket P \rrbracket_G \otimes \llbracket \{c/b\} \rrbracket_G
\]

\[
\llbracket ab.P \rrbracket_G = (\text{out}_b \otimes \text{id}_G) \circ \llbracket P \rrbracket_G
\]

\[
[0]_X = 1 \land X
\]

\[
[P \mid Q]_X = [P]_X \land [Q]_X
\]

\[
[\exists x.P]_X = \text{get}^{x,z} \circ [P]_X
\]

\[
[\forall x.P]_X = \text{send}^{x,z} \circ [P]_X
\]

\[
[[(\nu a)P]_n] = \text{hide}_n([(P \mid \cdots \mid a)]_{n+1})
\]

\[
[\forall P \mid Q]_n = \text{par}_n([P]_n \otimes [Q]_n)
\]

\[
[0]_n = \text{nil}_n
\]

\[
[\forall y].P]_n = \text{in}_n([P]_{n+1})
\]

\[
[M \mid N]_n = \text{choice}_n([M]_n \otimes [N]_n)
\]

Ten Virtues of Structured Graphs (GT-VMT'09) 37/45
A Re-usable Graph Algebra for Process Calculi

Components as edges \( l(\vec{x}) \), types as design labels \( L \).

\[
\begin{align*}
\text{(designs)} & \quad \mathcal{D} ::= L_{\vec{x}}[G] \\
\text{(graphs)} & \quad G ::= 0 \mid x \mid l(\vec{x}) \mid G \mid G \mid (\nu x)G \mid \mathcal{D}\langle \vec{y} \rangle
\end{align*}
\]

- In \( L_{\vec{x}}[G] \), the nodes \( \vec{x} \) in \( G \) are bound by the interface (as arguments), the other free names of \( G \) are global.
- We write \( L_{\langle \vec{y} \rangle}[G\{\vec{y}/\vec{x}\}] \) as a shorthand for \( L_{\vec{x}}[G]\langle \vec{y} \rangle \)
- A flattening axiom for some inessential design label \( L \) takes the form \( L_{\vec{x}}G\langle \vec{y} \rangle \equiv G\{\vec{y}/\vec{x}\} \) (but \( G\{\vec{y}/\vec{x}\} \) has still type \( L \))
- Structural equivalence as graph isomorphism
A Re-usable Graph Algebra for Process Calculi

Components as edges $I(\vec{x})$, types as design labels $L$.

\[
\begin{align*}
\text{(designs)} & \quad \mathbb{D} ::= L_{\vec{x}}[G] \\
\text{(graphs)} & \quad G ::= 0 \mid x \mid I(\vec{x}) \mid G\mid G \mid (\nu x)G \mid \mathbb{D}\langle \vec{y} \rangle \\
\end{align*}
\]

- In $L_{\vec{x}}[G]$, the nodes $\vec{x}$ in $G$ are bound by the interface (as arguments), the other free names of $G$ are global.
- We write $L_{\langle \vec{y} \rangle}[G\{\vec{y}/\vec{x}\}]$ as a shorthand for $L_{\vec{x}}[G]\langle \vec{y} \rangle$.
- A flattening axiom for some inessential design label $L$ takes the form $L_{\vec{x}}G\langle \vec{y} \rangle \equiv G\{\vec{y}/\vec{x}\}$ (but $G\{\vec{y}/\vec{x}\}$ has still type $L$).
- Structural equivalence as graph isomorphism.
A Re-usable Graph Algebra for Process Calculi

Components as edges $I(\vec{x})$, types as design labels $L$.

\[
\begin{align*}
\text{(designs)} & \quad \mathcal{D} ::= L_{\vec{x}}[G] \\
\text{(graphs)} & \quad G ::= 0 \mid x \mid I(\vec{x}) \mid G \mid G \mid (\nu x)G \mid \mathcal{D}\langle \vec{y} \rangle
\end{align*}
\]

- In $L_{\vec{x}}[G]$, the nodes $\vec{x}$ in $G$ are bound by the interface (as arguments), the other free names of $G$ are global.
- We write $L_{\langle \vec{y} \rangle}[G\{\vec{y}/\vec{x}\}]$ as a shorthand for $L_{\vec{x}}[G]\langle \vec{y} \rangle$.
- A flattening axiom for some inessential design label $L$ takes the form $L_{\vec{x}}G\langle \vec{y} \rangle \equiv G\{\vec{y}/\vec{x}\}$ (but $G\{\vec{y}/\vec{x}\}$ has still type $L$).
- Structural equivalence as graph isomorphism.
A Re-usable Graph Algebra for Process Calculi

Components as edges $I(\vec{x})$, types as design labels $L$.

\[
\begin{align*}
\text{(designs)} & \quad \mathcal{D} ::=} L_{\vec{x}}[G] \\
\text{(graphs)} & \quad G ::=} 0 \mid x \mid I(\vec{x}) \mid G \mid G \mid (\nu x)G \mid \mathcal{D}\langle \vec{y}\rangle
\end{align*}
\]

- In $L_{\vec{x}}[G]$, the nodes $\vec{x}$ in $G$ are bound by the interface (as arguments), the other free names of $G$ are global.
- We write $L_{\langle \vec{y}\rangle}[G\{\vec{y}/\vec{x}\}]$ as a shorthand for $L_{\vec{x}}[G]\langle \vec{y}\rangle$
- A flattening axiom for some inessential design label $L$ takes the form $L_{\vec{x}}G\langle \vec{y}\rangle \equiv G\{\vec{y}/\vec{x}\}$ (but $G\{\vec{y}/\vec{x}\}$ has still type $L$)
- Structural equivalence as graph isomorphism
A Re-usable Graph Algebra for Process Calculi

Components as edges $I(\vec{x})$, types as design labels $L$.

(designs) $\mathbb{D} ::= L_{\vec{x}}[G]$

(graphs) $G ::= 0 \mid x \mid I(\vec{x}) \mid G|G \mid (\nu x)G \mid D\langle \vec{y} \rangle$

- In $L_{\vec{x}}[G]$, the nodes $\vec{x}$ in $G$ are bound by the interface (as arguments), the other free names of $G$ are global.
- We write $L_{\langle \vec{y} \rangle}[G\{\vec{y}/\vec{x}\}]$ as a shorthand for $L_{\vec{x}}[G]\langle \vec{y} \rangle$
- A flattening axiom for some inessential design label $L$ takes the form $L_{\vec{x}}G\langle \vec{y} \rangle \equiv G\{\vec{y}/\vec{x}\}$ (but $G\{\vec{y}/\vec{x}\}$ has still type $L$)

Structural equivalence as graph isomorphism
A Re-usable Graph Algebra for Process Calculi

Components as edges $I(\vec{x})$, types as design labels $L$.

\[(\text{designs}) \quad D ::= L_{\vec{x}}[G]\]
\[(\text{graphs}) \quad G ::= 0 \mid x \mid I(\vec{x}) \mid G|G \mid (\nu x)G \mid D\langle \vec{y} \rangle\]

- In $L_{\vec{x}}[G]$, the nodes $\vec{x}$ in $G$ are bound by the interface (as arguments), the other free names of $G$ are global.
- We write $L_{\langle \vec{y} \rangle}[G\{\vec{y}/\vec{x}\}]$ as a shorthand for $L_{\vec{x}}[G]\langle \vec{y} \rangle$.
- A flattening axiom for some inessential design label $L$ takes the form $L_{\vec{x}}G\langle \vec{y} \rangle \equiv G\{\vec{y}/\vec{x}\}$ (but $G\{\vec{y}/\vec{x}\}$ has still type $L$).
- Structural equivalence as graph isomorphism.
Some Sketches of Encoding

$\pi$-calculus in ADR ($P$ process type, $G$ guarded sums type)

\[
\begin{align*}
\sem{(\nu x)Q} &= P_p[ (\nu x)\sem{Q} \langle p \rangle ] \\
\sem{N + M} &= G_p[ \sem{N} \langle p \rangle | \sem{M} \langle p \rangle ] \\
\sem{Q \parallel R} &= P_p[ \sem{Q} \langle p \rangle | \sem{R} \langle p \rangle ]
\end{align*}
\]

CaSPiS in ADR ($P$ process type, $S$ session type)

\[
\begin{align*}
\sem{Q \parallel R} &= P_{p,i,o,r}[ p | i | o | r | \sem{Q} \langle p, i, o, r \rangle | \sem{R} \langle p, i, o, r \rangle ] \\
\sem{s^+ \triangleright Q} &= P_{p,i,o,r}[ i | o | S_{\langle p,r \rangle}[ \sem{Q} \langle p, s^+, s^-, r \rangle ] ] \\
\sem{s^- \triangleright Q} &= P_{p,i,o,r}[ i | o | S_{\langle p,r \rangle}[ \sem{Q} \langle p, s^-, s^+, r \rangle ] ]
\end{align*}
\]
Some Sketches of Encoding

$\pi$-calculus in ADR ($\mathbf{P}$ process type, $\mathbf{G}$ guarded sums type)

$$
\llbracket (\nu x) Q \rrbracket = \mathbf{P}_p [ (\nu x) \llbracket Q \rrbracket \langle p \rangle ]
$$
$$
\llbracket N + M \rrbracket = \mathbf{G}_p [ \llbracket N \rrbracket \langle p \rangle | \llbracket M \rrbracket \langle p \rangle ]
$$
$$
\llbracket Q | R \rrbracket = \mathbf{P}_p [ \llbracket Q \rrbracket \langle p \rangle | \llbracket R \rrbracket \langle p \rangle ]
$$

CaSPiS in ADR ($\mathbf{P}$ process type, $\mathbf{S}$ session type)

$$
\llbracket Q | R \rrbracket = \mathbf{P}_{p,i,o,r}[ p | i | o | r | \llbracket Q \rrbracket \langle p, i, o, r \rangle | \llbracket R \rrbracket \langle p, i, o, r \rangle ]
$$
$$
\llbracket s^+ \triangleright Q \rrbracket = \mathbf{P}_{p,i,o,r}[ i | o | \mathbf{S}_{(p,r)}[ \llbracket Q \rrbracket \langle p, s^+, s^-, r \rangle ] ]
$$
$$
\llbracket s^- \triangleright Q \rrbracket = \mathbf{P}_{p,i,o,r}[ i | o | \mathbf{S}_{(p,r)}[ \llbracket Q \rrbracket \langle p, s^-, s^+, r \rangle ] ]
$$
Visualization: adr2graphs (Early Prototype)

Please have a try at [http://www.albertolluch.com/adr2graphs](http://www.albertolluch.com/adr2graphs)

*a simple visualiser of term-like specifications*

choose the input language: [pi-calculus] choose the output format: [formal hierarchical graph]

enter a term: (nu "secret" . "gossipers" ! "secret") | "gossipers"? "message"

Ten Virtues of Structured Graphs (GT-VMT'09) 40/45
One Last Virtue of Structured Graphs

▶ Logical specification and verification
  ▶ ad-hoc spatial logics: from “general” to “derived” modalities
  ▶ formulas closer to visualization (easier to use)
  ▶ types as properties: a property $\mathcal{P}$ demonstrated by structural induction on type $T$ show that all graphs of type $T$ satisfy $\mathcal{P}$.
  ▶ re-use existing (efficient) tools whenever possible
Outline

Introduction

Styles for Visual Support

Dynamics

ADR

Concluding Remarks
Where ADR can help

- Design of software architectures
  - drop & bind components + check + correct: tedious, error prone
  - bounded FO/SAT (Alloy): performant, but trial & error, no hint, no guidance

- Guaranteed reconfiguration
  - prove theorems on GT: ad-hoc, manual, limited re-use
  - model checking on GT: validate a particular instance, scalability issues, undecidable in general
  - monitor & repair: no guarantees

- Usability
  - other integrated environment require acquaintance with many different languages and theories
Related work

- **Ordinary GT:**
  - nice theory of concurrency, but structure must be encoded somehow in flat graphs,
  - problems with grouping and atomicity

- **Hierarchical graphs:**
  - main difference relies on interfaces

- **Alloy:**
  - highly specialized SAT solver, but Maude is more flexible
End of Talk (Graphs Powered by yEd)
End of Talk (Graphs Powered by yEd)