Towards the Synthesis of Modular Software Systems

Chris Mellish
School of Informatics
University of Edinburgh

Contents

Building/maintaining a space of software systems

- Each system using a small number from a potentially large set of modules.
- Modules varying in their functionality, programming languages (and possibly host machine).
- Inter-module communication relatively infrequent, but involving relatively large and complex data.
- Data communicated between modules is of interest (human inspection, statistical modelling).

Natural Language Generation

NLG involves generating natural language text to express initially non-linguistic information.

- No general agreement on the architecture of an NLG system
- Many theoretical frameworks and programming paradigms
- Agreement in the abstract about useful NLG tasks
- Need for reusable and interchangeable modules, e.g. for evaluation/comparison

Limitations of Current Technology

Most current inter-process communication mechanisms (e.g. CORBA, DCOM, RMI):

- Don't facilitate reasoning about module compatibility
- Emphasise efficient binary exchange formats that are not inspectable
- Concentrate on modelling numerical data and ignore high-level distinctions (e.g. sets vs sequences)
- Impose a particular programming orientation (e.g. object-oriented)

RAGS - Specifying Modules

Module developers need to have:

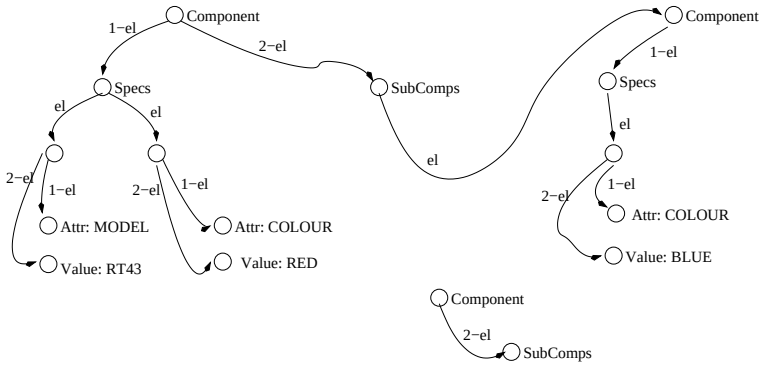
- A shared specification of possible data, expressed using abstract type definitions (essentially an upper ontology)
- A shared understanding of the set of possible information states exchangeable between modules, i.e. a position on:
  - Partiality
  - Structure of complex (e.g. mixed) datasets
  - Equality (reentrancy)

  These are embodied in the formal definition of a "reference implementation", the "objects and arrows model"

Example: Hardware components

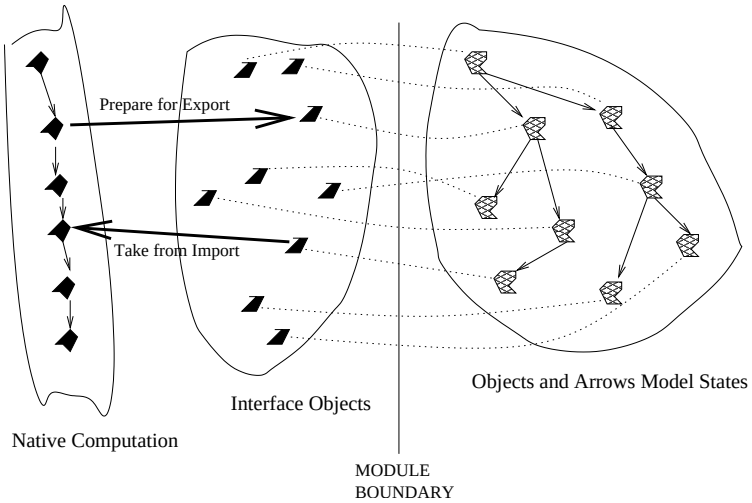
$$
\begin{aligned}
Component &= Specs \times SubComps \\
Specs &= Attr \rightarrow Value \\
SubComps &= 2^{Component} \\
Attr &\in Primitives \\
Value &\in Primitives
\end{aligned}
$$

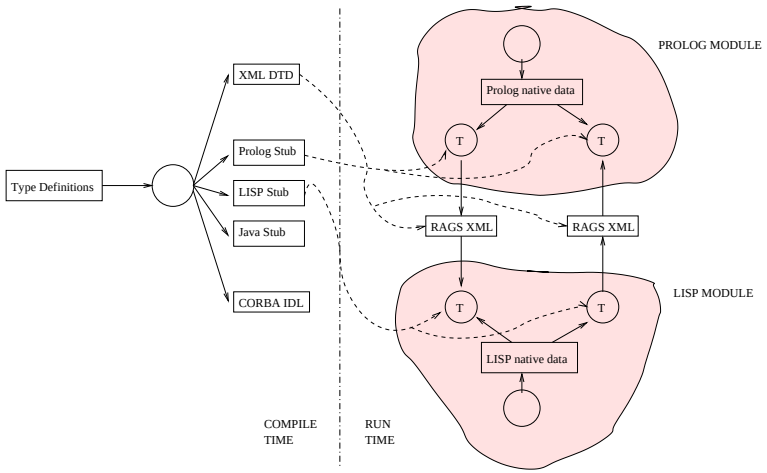
Objects and Arrows Structure



RAGS - Plugging them Together

- Code is provided for modules in LISP, Java and Prolog to exchange data (via sockets) in a neutral, faithful, XML format.
- Code is provided to support (i.e. produce XML input/output to/from) various "native" formats in the programming languages.
- A central configuration file specifies how module input and output channels are connected.
- A running module advertises its "role" to a server and carries out i/o through its logical channels without knowing where they are connected to.





What remains - the Synthesis

- Formal definition of module inputs and outputs. Idea: Use description logic based on the OA structure

$$Component \sqcap \exists 1el.\exists el.\exists 1el.MODEL$$

$$\neg Component \sqcup \exists 2el.\exists el.Component$$

  Problem: The expressions will be complex. The underlying theory $\mathcal{T}$ (type definitions, OA constraints) will be bulky.

- Use this to test whether an output satisfies an input:

$$satisfies(o,i) \equiv (\forall \mathcal{M}.\mathcal{M} \models_{\mathcal{T}} o \supset \mathcal{M} \models_{\mathcal{T}} i)$$

What remains...

- Use this to answer:
  - Could this module fit in at this point in a system?
  - How could this module be adapted so that it fitted in?
- Automatically plan possible configurations of modules to implement given objectives.
- Need to handle translation between low level parts of the ontology (the "Primitive" types)