Domain-Oriented Component-Based Automatic Program Generation

Teodor Rus
rus@cs.uiowa.edu

The University of Iowa, Department of Computer Science
A challenging problem:

- Development of a programming methodology where programmers manipulate computing processes, rather than data and operations, according to the logic of their problems.

- Processes manipulated by programmers are high-level abstractions representing solutions to subproblems of a given problem.

- Domain specific problem solving environments populate a computer system.
A domain specific PSE is characterized by:

- Specification mechanisms, that allow problem solvers to specify their problems formally
- Tools, that map problem specifications into data and function structures on which domain-specific universal algorithms operate
- Components, that are domain-specific universal algorithms that solve predefined classes of problems
Problem solving methodology:

- Programmers express the architecture of the systems solving their problems in terms of the stand-alone components.
- Interpreters map these expressions into processes solving problems.
- Hence, programmers handle the process of problem solving as developed by humans (mathematicians), using tools provided by computing environment.
Examples PSEs:

- Time-sharing systems, where:
  - Specifications: system file names
  - Tools: compilers, loaders, linkers, etc.
  - Components: executables
  - Software architectures: specified by control languages; software functionality: defined by interpreters


- Language processing environment used in TICS
Current PS methodology:

- Develop symbolic notations for machine computations called programming languages
- Map automatically programming language notations into machine language programs

**Note:** one handles machine computations not problem solving process

**Problem:** software system cost + complexity;

**Solution:** automatic program generation
Observations

- All systems developed so far for APG manipulate text representing machine logic not the human logic of problem solving.
- Therefore this path of APG cannot result in automatic problem solving as expected.
- To achieve this stage of automatism, systems that manipulate abstractions representing human logic need to be developed.
- This may only come with the rethinking of the programs as language expressions of human logic rather then higher or lower level notations of machine computations.
The missing link:

- separation of software system architecture from software system functionality.

Note: today software systems are intertwines of structures and functions representing high-level language expressions of machine-language computations called programs.
System architecture

- Develop software system architecture independent of its functionality, in terms of components;
- Components are specified by expressions of an architecture development language;
- A component’s correctness and consistency is specified by invariants and properties

**Note:** during system architecture development components may or may not have expressions as programs running on an abstract or concrete machine.
System functionality:

Define software system functionality by an interpreter that:

- Analyzes the system architecture proving its consistency and correctness by verifying properties and invariants
- Maps code associated with components into processes performing the functions expressed by their architectures
- Compose processes performing components into the process performing the system functionality
Two resulting methodologies:

- A methodology to handle the architecture that allows computer users to develop software system blueprints, independent of their functionality.

- A methodology to handle the functionality that allows computer users to associate a software system architecture with the functionality they want it to perform.

**Note:** architecture methodology evolves with problem domain; functionality methodology evolves with computer systems.
Conclusions

- A natural language provided with a human logic is the appropriate tool to express a software architecture.
- A conventional programming language can be used to express its functionality.
- Interpreters mapping natural language expressions into computer processes bind architectures to their functionality.

Note: A true APG process becomes within the reach...
Architecture description languages:


- Domains of computer applications characterized by a set of software architectures called *styles*
- Design element types, formal expressions of expertise
- A decidable logical system to express properties and invariants

The representative ADL are *Acme* and *Armani*
Dissing element types:

1. *Components*: computation units of the system. Components are provided with interfaces called *ports*.

2. *Connectors*: interactions between components. Connectors are provided with interfaces called *roles*.

3. *Systems*: graphs whose nodes are components and edges are connectors.
Design element specification

- **Topology**: element structure defined by attaching roles to ports and ports to roles

- **Expression**: element description using a vocabulary of design elements

- **Properties**: logical expressions representing semantic information. Different ADLs focus on different such properties

- **Constraints**: properties that allow a system architecture to evolve over time while preserving given invariants.
TICS system:

1. **Components**: universal algorithms such as: comparing strings, lexical analyzers, parsers, APT constructors, macro-generators, etc.

2. **Connectors**: filters mapping data types to XML expressions and vice-versa.

3. Each component and filter is associated with calling and exiting patterns.

4. Interpreters that map expressions composing calling and exiting patterns into processes that execute the associated programs.
Component/Filter implementation

Shown in Figure 1.

Figure 1: Structure of a component
Note:

- The *action* is the fixed part and the *data* is the variable part of the algorithm implementation.

- The *input* and *output* are the calling and exiting patterns also referred to as *interface specification* and are the only parts of the algorithm a user may have to handle.

- The process performing the stand-alone program component described in Figure 1 is generated from the specification schemes.
The tuple \((I, <\text{action}, \text{data}>, O)\) is further used to denote a program component (automatically) generated from specification.

The action part, while fixed within a given program component, can vary over a given number of universal algorithms solving the same problem.

A generating tool selects the action from a finite set depending upon the properties of the specification.
System architecture

- An expression that shows the system structure in terms of components and filters using sequential and parallel compositions.

- Examples of such architectural expressions are makefiles.

- Language processors are processes generated by the interpreter of these architectural expressions.

- Example: *make* program in a Unix system.
Sequential composition

If \( P_1=(I_1,\langle action_1, data_1 \rangle, O_1) \), \( P_2=(I_2,\langle action_2, data_2 \rangle, O_2) \) are components then
\[
P_3=(I_1,\langle \{action_1, O_1 \xrightarrow{F} I_2, action_2\}, \{data_1, data_2\} \rangle, O_2)
\]
is their sequential composition and \( \forall w \in I_1 \), \( P_3 \) performs the sequence of actions:
1. \( x = action_1(w, data_1) \);
2. \( y = F(x) \);
3. \( z = action_2(y, data_2) \)

The architecture of \( P_3 \) is shown in Figure 2.
Architecture

Figure 2: Architecture of sequential composition
Parallel composition

If $P_1 = (I_1, < action_1, data_1 >, O_1), P_2 = (I_2, < action_2, data_2 >, O_2)$ are components then $P_3 =$

$(I_1 \times I_2, < \{action_1, O_1 \overset{F_1}{\rightarrow} I_2 \times O_2 \overset{F_2}{\rightarrow} I_1, action_2\}, \{data_1, data_2\} >, O_1 \times O_2)$ is their parallel composition and performs as follows:

$\forall (w, x) \in I_1 \times I_2$, $P_3$ determines $(y, z) \in O_1 \times O_2$ by the parallel actions

$y = action_1(w, data_1) || z = action_2(x, data_2)$.

Architecture of $P_3$ is shown in Figure 3.
Figure 3: Architecture of parallel composition
Sequential and parallel compositions can be combined, thus constructing more complex program architectures, as seen in Figure 4. This architecture complicates further if we allow iterations, i.e., filters $F_i : O_i \rightarrow I_i$, $F'_i : O'_i \rightarrow I'_i$, and $G : O'_i \rightarrow I_i$. 
Figure 4: Parallel and sequential compositions
Complexity

- Complexity of a system depends only on the number of communication channels and the nature of their compositions.

- Correctness can be mathematically proven from the correctness of the components and can be incrementally verified.

- The users of such a system operate with high level abstractions such as "compare", "parse", "compile", "execute", etc.
TICS specification rules:

- Equations of the form

\[ \text{lhs}(r) = \text{rhs}(r); \text{Sem}_1(r), \text{Sem}_2(r), \ldots, \text{Sem}_k(r) \]

- Here \( \text{Sem}_i(r), 1 \leq i \leq k \), are semantic macro-operations defining various semantic values of the constructs \( w \) specified by \( r \).
Example semantics:

Transition semantics of a construct $w$ specified by the rule $r$:

$$\langle \text{Type}(\text{rhs}(r)), \text{State}(\text{rhs}(r)), \uparrow (\text{rhs}(r)) \rangle \xrightarrow{\text{w}} \langle \text{Type}'(\text{rhs}(r)), \text{State}'(\text{rhs}(r)), (\text{rhs}(r)) \downarrow \rangle$$

1. $\uparrow (\text{rhs}(r))$ is the computation before execution of $w$, and $(\text{rhs}(r)) \downarrow$ is the computation reached after the execution of $w$

2. $\text{Type}(\text{rhs}(r))$ and $\text{Type}'(\text{rhs}(r))$ are the sets of types available before and after computation, respectively;

3. $\text{State}(\text{rhs}(r))$, $\text{State}'(\text{rhs}(r))$ are functions showing the values of the reachable variables before and after computation, respectively.

**Note:** $\text{Type}(\text{rhs}(r))$, $\text{State}(\text{rhs}(r))$, and $\uparrow (\text{rhs}(r))$ are maintained in three data structures called $\text{def}(r)$, $\text{dec}(r)$, and $\text{app}(r)$. 
Semantics expression language, SEL:

- SEL is a user-oriented notation to express computation meanings, independent of the machine and problem domain.
- SEL manipulates three kinds of abstractions: types, functions, and operators.
- Types represent the universe of values manipulated by SEL computations,
- Functions express the states of SEL computations,
- Operators express state transitions.
Currently SEL consists of:

- **Predefined types**: `integer: +, -, *, /, %`; `boolean: and, or, not`; `string: concatenation, comparison`; `file: read, write`.

- **Type constructors**: `record: element extraction (.)`; `function: definition, declaration, invocation`; `list: append, prepend, insert, delete`; `set: add, remove, union, intersection, subtraction`; `array: subscriptions`; `reference: dereference`.
Abstraction manipulation in SEL:

- **Definitions:** define \( \langle name \rangle \) as \( \langle type \rangle \);
  such as define words as list(string);
- **Declarations:** let \( \langle name \rangle \) be \( \langle type \rangle \);
  such as let dictionary be words;
- **Applications:**
  
  set \( \langle object \rangle \) to \( \langle expression \rangle \)
  
  if \( \langle boolean expression \rangle \) then \( \langle statement \rangle \)
  
  if \( \langle boolean expression \rangle \) then \( \langle statement \rangle \) else \( \langle statement \rangle \)
  
  while \( \langle boolean expression \rangle \) do \( \langle statement \rangle \)
  
  begin \( \langle statement \rangle ; \langle statement \rangle ; \ldots ; \langle statement \rangle \) end
Language specification file:

Two parts: (1) a semantics-header, where all semantic domains are defined, and (2) a sequence of language specification rules:

Semantic domains:

\[
\text{beginImage Name}_1 \text{ Image}_1 \text{ endImage} \\ \ \ldots \ \ \ \ldots \ \ \ \ldots \ \ \ \ldots \ \ \ \ldots \\
\text{beginImage Name}_n \text{ Image}_m \text{ endImage}
\]

Specification rules: Rule_1 \ldots Rule_n where Rule_i is:

\[
A_0 = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n;
\]

\[
\text{beginImage Name}_1 \text{ SEL-macro}_1 \text{ endImage}
\]

\[
\ldots
\]

\[
\text{beginImage Name}_n \text{ SEL-Macro}_k \text{ endImage}
\]

Precedence relations between images.
Environment construction tools:

1. The Splitter: extracts specification layers from specification rules and constructs TICS database on which other tools operate.
Lexical analysis tools:

1. The first level scanner generator, (FLSG): maps regular expressions into the table of the finite automaton specified by them.

2. The second level scanner generator, (SLSG): maps regular expressions of conditions into the conditional automaton thus specified.
Language analysis tools

1. The interactive parser constructor, (IPC): maps BNF rules into the parse table of the automaton thus specified.

2. Language analysis system, (LAS): maps BNF rules into a language space, collects context information, and constructs the Table of Universal Operations, TUSO.
Semantics analysis tools:

1. **IPC Sem**: an IPC augmented with the capability of handling semantic information associated with the rules using functions registered in the parse table entries.

2. **Semantics processors, (SP)**: expand semantic macro-operations into semantics constructs called *images*.
There are two kinds of errors to be considered:

- Errors made by the TICS users while developing and implementing (new) languages.
- Errors made by the programmers using TICS compilers.

The errors made by the TICS users are manipulated by the TICS tools they use and the implementation systems of these tools.

The errors made by programmers are manipulated by error managers provided as components of the TICS environment.
Component generation in TICS:

See the dashbox in Figure 5.

Figure 5: Components and their generation
Component integration tools

1. Interface manager, IM: a forest-data type that provides interface operations for component integration.

2. Trees manipulated by IM, called construct recognition trees, (CRT), see Figure 6.

Figure 6: Construct recognition tree
The UniText is a universal text representation of the CRT similar to XML.

The UniText representation of the tree in Figure 6 is:

\[
\text{lhs}(r): \\
\text{Syn}: \text{UniText} (\text{SyntaxRT}); \\
\text{Sem}: \text{UniText} (\text{SemanticsRT});
\]

where \(\text{lhs}(r)\) is the left-hand side of \(r\) and \(\text{UniText} (\text{SyntaxRT})\) and \(\text{UniText} (\text{SemanticsRT})\) are the UniText representations of the syntactic and semantic information held by the tree.
TICS components:

- Lexical analyzers
- Syntax analyzers
- Semantics analyzers
- Code generators
Lexical analyzers:

- **First Level Scanner, FLS**: recognizes a lexicon that is common to all programming languages, such as identifiers, numbers, white spaces, unprintable characters;

- **Second Level Scanner, SLS**: implements a finite automaton of conditions, where conditions are properties of the lexical elements of the language recognized by FLS, such as lexeme, length of the lexeme, token of a lexeme, range of a character-set, position of the lexeme in the input, etc.

**Note**: FLS is efficient and universal; SLS is powerful and convenient.
TICS parsers:

- **LR (LR(0), LR(1), LALR):** consume their input from left-to-right while performing shift, reduce, accept, and error actions, accompanied by calls to IM to construct CRT-s and to perform attribute computations.

- **Pattern Matching Parsers, PMP:** consume their input in any order; the input to a PMP is a tokenized form of the input-text called the *file of internal form*, (FIF), with records of the form \( \langle CRT, Token \rangle \)

**Note:** actions performed by PMP are: (1) match \( rhs(r) \) with FIF; (2) if a match is discovered, call IM to replace the FIF portion matched by \( rhs(r) \) with the record \( \langle CRT, lhs(r) \rangle \)
A transition semantics analyzer, TSA, constructs tuples $\langle \text{def}(r), \text{dec}(r), \text{app}(r) \rangle$ and asks the IM to associate them as $\text{SemanticsRT}$ to the corresponding CRTs.

**Note:** TSA operates in parallel with a parser, synchronizing its actions on the availability of the information it requires to perform its actions, such as types and scopes of the construct components it manipulates.
Images of constructs recognized by parsers using the rule $r$ are specified by macros $Sem_i(r), \ 1 \leq i \leq k$, associated with $r$

$Sem_i(r)$ may represent assembly language constructs, C constructs, Java constructs, various process dependence graphs, etc.

A code generator is a macro-processor that expand $Sem_i(r), \ 1 \leq i \leq k$, into appropriate target images
TICS code generators:

- Assembly-language macro-processors
- C and Java languages macro-processors
- Process dependence graphs constructors
- Program parallelization constructors
Component integration in TICS:

by two kinds of filters:

1. $InF_{i,j} : \text{UniText} \rightarrow C(\text{Input})$

2. $OutF_{i,j} : C(\text{Output}) \rightarrow \text{UniText}$.

where $C(\text{Input})$ is the input expected by component $C$ and $C(\text{Output})$ is the output generated by component $C$. 
Integration rules:

- Components $C_i$ and $C_j$ are sequentially integrated into the component $C_i;C_j$ by a filter $F_{ij} : \text{Output}(C_i) \rightarrow \text{Input}(C_j)$.

- Components $C_i$ and $C_j$ are parallel integrated into the component $C_i||C_j$ by $F_{i,j}^{out} : \text{Output}(C_i) \rightarrow \text{Buffer}(\text{UniText})$ and $F_{i,j}^{in} : \text{Buffer}(\text{UniText}) \rightarrow \text{Input}(C_j)$.

- $F_{i,j}^{out}$ and $F_{i,j}^{in}$ are implemented by message-passing primitives or by critical-sections, depending upon the sharing mechanism used to access the Buffer(\text{UniText}).

- Filters are independent of the components they integrate.
Comparisons:

TICS environment is similar to the environments of Eli and Zephyr

- syntax specification uses BNF rules, though TICS rules are not necessarily interpreted as defining grammars;
- semantics specification uses SEL macro-operations which are similar to attributes; however, SEL macro-operations specify language constructs not properties of language constructs;
- some of the tools we use, such as FLSG and IPC, are similar with earlier tools such as Lex and Yacc; however, FLSG and IPC are incremental and compositional, allowing user to work rule by rule and semantic-domain by semantic-domain;
- the interface manager we use bears similarities to the SUIF system, though it manipulates a well-defined data type using an open-ended collection of operations, added or removed as
TICS success story:

- TICS is rather a methodology than a software engineering tool.
- All the experiments of language processors designed and implemented by TICS have only a theoretical value.
- However, generations of students succeeded to learn compiler construction by hands on language design and compiler implementation using TICS.
- Educational process using TICS have shown that the new software technology requires new teaching and learning methods based on hands-on-tools rather than on hands-on-textbooks.
Current work on TICS

**Goal**: use TICS as a tool for the development of a new problem solving methodology where programmers manipulate computing abstractions rather than machine representations

- To allow system’s architecture specification and reasoning about systems using natural language we implement Armani in TICS.
- To allow system’s functionality specification we add to TICS a scripting language, SELScript, designed on top of SEL.
- An interpreter will map SELScript expressions into sequential or parallel processes, executing functions of the software system architectures these expressions represent.