Design Derivation of an Open, Java-based Reengineering Platform

Paul Bailes
Ian Peake
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 AUSTRALIA
{paul, ian.peake}@itee.uq.edu.au

Abstract

Essential reengineering platform functionality can be made more accessible, and even extended, using Java as a basis for platform development, as follows. The process of software reengineering tool development should be able to enjoy the benefits conferred by "open" enabling technologies. Reengineering tool development and use involves distinctive processes and thus has distinctive requirements of enabling technologies. The earlier dominant "Software Refinery" proprietary reengineering tool development platform is both lacking with respect to required reengineering characteristics and moreover has limited accessibility. However, using Java as basis, the functionality of the proprietary platform can be substantially recreated to achieve open technology goals of adaptability, portability, accessibility and eventually heterogeneity also.

1. Introduction

The general benefits of "open" approaches to software development – their methodologies and enabling technologies – as opposed to the alternative idiosyncratic/stand-alone/proprietary systems are well-known [1]. They include:

- *heterogeneity* – ability to integrate components developed using different technologies;
- *portability* – ability to use on a wide variety of hardware/software environments;
- *accessibility* – beyond mere portability, not prohibitively expensive as well (the nature of software economics seems to be that portability is incompatible with prohibitive expense, thus accessibility implies portability);
- *adaptability* – relative ease of modification/extension of existing behaviour, on account of visibility to some extent of the internal components of a software system.

However, one significant software domain – that of the development of (semi-) automatic reengineering tools – has hitherto tended to be dominated by proprietary rather than open technology. The purpose of this paper accordingly is to introduce an open platform for software reengineering tool development, by sequentially:

a) deriving, after consideration of the special characteristics of software reengineering, the technical requirements for software reengineering tool platforms;
b) demonstrating, with respect to these requirements, the limitations of a dominant proprietary platform;
c) exhibiting the design of "Bahasa" - an alternative Java-based reengineering platform;
d) presenting the conceptual hierarchical progression of the extensions to Java that lead to a canonical implementation of Bahasa.

Thus, while the platform resulting from this process ("Bahasa") is potentially significant, the deeper significance of this paper is its exposition of how the platform is derived from the requirements of the reengineering domain, and its transparency to other platform developers.

2. Reengineering Platform Requirements

It is important carefully to distinguish between the three kinds of software:

- reengineering *platforms*, in which are developed ...
- reengineering *tools*, which in order to facilitate reengineering are applied to ...
- *application systems*.

It is the first of these that is our concern here, though it would be surprising if the use of an open reengineering tool platform (and thus an open approach to reengineering tool development) did not engender a degree of openness in the tools themselves and maybe likewise even in the resulting applications themselves.

The essence of the derivation of reengineering platform requirements (according to which our open platform is to be developed) is a reengineering tool development and application process model – our Generic Reengineering Architecture (GRA). The GRA in turn is

534-5351/03 $17.00 © 2003 IEEE

derived from an analysis of what distinguishes reengineering from other types of software development.

2.1. Distinguishing characteristics of reengineering

Reengineering tool development may be characterised by the important distinctions between it and other kinds of programming, and in particular from other kinds of metaprogramming such as compiler-writing.

2.1.1. Imperfectibility. Source-level transition requires a much higher level of program comprehension compared to compilation, because of the requirement that the result be expressed in high-level terms comprehensible to a human reader (such as a maintenance programmer). Moreover, the meaning of source code can be deeply disguised, ultimately in the form of idiomatic expressions which we conjecture are as opaque as the idioms of natural language that contribute so much to the "AI-hardness" of such as natural language understanding, and related problems such as theorem-proving. Thus a transition tool will generally require some degree of human operator assistance. See also [2].

2.1.2. Infrequency. Whereas compilers are applied many times to the same source code during development and maintenance, the singular nature of transition implies that these tools are applied much less frequently to each Origin-Target pair under consideration. The implication is that transition processes and tools can be designed with the usual compromise between machine efficiency vs tool capability/expressiveness weighted more heavily towards the latter than normal. While tool use might be affected by such a compromise, application use is not affected, since application source code itself would simply have been translated from one form to another and could still be compiled as usual.

2.2. The GRA: reengineering process model

We have earlier reported [3] on an initial version of a process model for reengineering tool development and application: the Generic Reengineering Architecture (GRA). The latest version of the GRA is a 3-level hierarchy as summarised in fig 1 above. Note the distinctions between:

1. application of a reengineering tool (*ITALIC TEXT*)
2. implementation of a reengineering tool (**BOLD TEXT**)
3. specification of a reengineering tool and a reengineering process (**BOLD UNDERLINE**)

2.2.1. Reengineering - application-level architecture. The architecture's tool-application components are as follows.

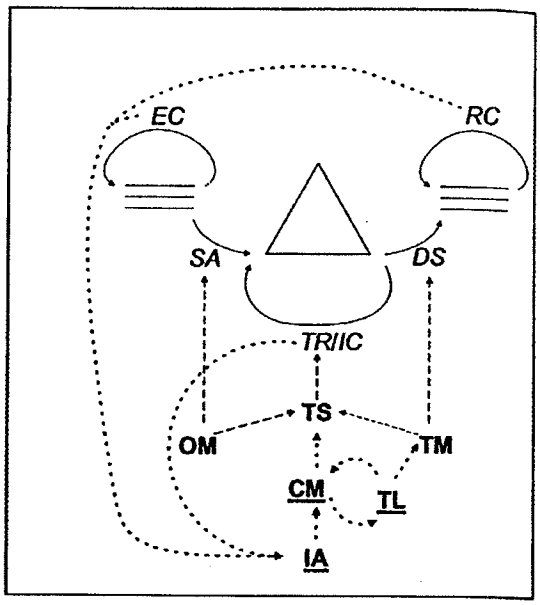

Figure 1. Generic Reengineering Architecture

- External Censoring (EC): because not all reengineering functions can be automated ("imperfectability"), manual preprocessing of original source code may be considered in order to elide constructs not amenable to automatic processing - quasi-retention of such constructs may be achieved by "commenting out".
- Syntactic Analysis (SA): parse remaining original source code into internal (e.g. abstract syntax tree) form; note that valuable extra-linguistic content e.g. comments may be retained by attachment to nearest AST node. (By "ASTs" we refer metaphorically to the internal representation of any analysed artifact of the maintenance process, and representations of the output of subsequent derived high level analyses.)

The explicit iteration of the next two stages implies that human operators have the opportunity to inspect the results and to make intelligent decisions about which TR or IC operation should be performed next.

- Transformation (TR): apply (typically) preprogrammed transformations to AST in order to effect desired transition.
- Internal Censoring (IC): delete or reduce to comment-status further AST nodes that emerge as not amenable to automatic processing (corresponds to EC).
- Differential Synthesis (DS): generate target source code from AST; because not all original source code

may have been transformed by TR ("imperfectability"), the tree may be a hybrid, hence synthesis should differentiate between source code generated in origin vs. target languages; also, original comments may be regenerated at some corresponding location in target code.

- Reconciliation (RC): manually postprocess hybrid code, involving replacement of residual origin by target code and inspection comments for elided (by EC or IC) origin code and replacement by target code.

2.2.2. Reengineering - tool implementation. While the *infrequency* of transition has implications for the technology used to implement transition tools, the implementation process itself reflects the familiar "compilers" approach.

- Origin Modelling (OM): specify concrete and abstract syntax for origin language (from which parser and AST generator for SA above can be generated).
- Target Modelling (TM): specify concrete and abstract syntax for target language (from which unparse/printer/code generator for DS can be generated).
- Transformation Specification (TS): programming of origin-target transformations to be available for selection in TR above.

2.2.3. Reenginering - tool specification. The ultimate *imperfectability* of transition tools is manifested in the feedback from tool application into tool specification.

- Concept Mapping (CM): high-level specification of origin-target conversions for implementation in TS.
- Target Lifting (TL): simplify CM (and so TS) by semantic extension of target language (e.g. by development of libraries) to match semantics of origin.

The above two operations are mutually-iterative: TL is inspired by requirements of CM; CM is simplified by suitable TL.

- Idiom Analysis (IA): take account of possibly application-, or organisational-dependent means of expressing concepts that are not otherwise obviously expressible in the origin language; occurs prior to CM in order that CM may safely ignore potentially imperfectable special cases otherwise manageable by IA.

The feedback from application architecture manual components (EC, TR, IC RC) into IA, preceding CM, is indicative of how human interaction with transition tools is likely to yield insights into further idioms and other mappings between origin and target constructs.

2.3. Required characteristics of reengineering platforms

The basic characteristics of reengineering and their refinement in terms of our GRA imply that reverse engineering platform should support key attributes of persistence, expressiveness and extensibility.

2.3.1. A framework for linguistic adequacy. We propose that the adequacy of a programming language (and by inheritance, reengineering platforms also) can be ordered:

1. primary criterion – adequacy to an intended application domain;
2. secondary criterion – adequacy to a prevailing programming methodology;
3. tertiary criterion – adequacy to universal principles of language design [4].

These "universal principles" merit further elaboration. The principle of *adequacy*, as elaborated by this framework, is one such. Another such principle is *simplicity*, which is nothing but the converse of obviously-undesirable complexity. A third principle, *extensibility*, arises out the tension between adequacy and simplicity as follows. Naively, the most adequate language would be one with every conceivable value, type, operation, control construct in some way predefined. Equally naively, the simplest language would have a correspondingly simple formal definition, namely { }. Extensibility therefore is the means by which the tension between adequacy and simplicity can be managed: a small extensible base language will be simple, yet by virtue of its various extensions can achieve adequacy according to specific instances of the primary and secondary criteria above.

Example characterizations of programming languages according to this framework are as follows.

- FORTRAN has primary adequacy to the application domain of scientific computation, little adequacy to any recognized programming methodology and some extensibility at the operator/statement level (from subroutines and functions).
- C has primary adequacy to the systems programming domain, support for "structured programming" (control constructs) and, as well as FORTRAN-style extensibility, a useful macro preprocessor.
- Functional languages [14] by contrast glory in providing little inbuilt support for specific domains. Rather, they emphasise tertiary adequacy (general extensibility) from which support for specific applications (e.g. in the Haskell case from arithmetic to animation) can be synthesized. Explicit support of a specific methodology is increasingly popular (e.g. Haskell [5] emphasizes a functional flavour of object-orientation), and research continues in how to synthesise such secondary adequacy by extending a methodologically-neutral basis [6].

We acknowledge that primary application-domain-specifics may impinge upon the secondary and tertiary notions of adequacy.

2.3.2. Reengineering-specific adaptations. In the case of reengineering, domain influences upon the above-identified three levels of adequacy are as follows.

1. Primarily, GRA components SA, DS, OM and TM all require concrete/abstract syntax specification capability of reengineering platforms. Furthermore, with respect to the further specifics of the reengineering domain, *imperfectability*, as manifested in particular by the user control of the Transformation (TR) and Internal Censoring (IC) components of the GRA, requires that the platform support persistent storage of internal representations of source code (AST and related structures). This use of "persistent" is in the specific sense of the potential for arbitrary user-control of operations on and interaction with data structures that results from their continued existence after a program that processes them has terminated. An example of such persistence would be the file system component of an operating system – individual programs process files, but by default the files "persist" after each program terminates.

2. Methodologically, TR and TS require capability for expressing source-source transformations on ASTs. *Infrequency* means that as the time/resource efficiency of reengineering tools may not need to be as great as for applications development tools, very-high-level language technologies that emphasise expressiveness over efficiency may be considered as bases for reengineering platforms. Congruently, *imperfectability* as manifested by the cycle from application-level components back to Idiom Analysis (IA) suggests an overall view of reengineering tool development as a kind of continuous evolutionary prototyping, the very-high-level enabling technologies for which being compatible with the foregoing expressiveness requirements.

3. GRA's Target Lifting (TL) is a good example of the universal need for extensibility, in this case of the target language if possible. Further, the feedback loop into reengineering tool development (starting at IA) from reengineering tool application (IC, TR) is indicative of increments that may need to be made in TS, OM and TM so that e.g. as insights into the particular conversion are discovered during reengineering tool use, they can be incorporated into the automated capability. For example, it may be necessary to develop a family of reengineering tools for different dialects of one language (e.g. proprietor variants), and in so doing it would be desirable to express the variants relative to a common basis. Ultimately however, the fact that the GRA is itself a work in progress means that there is also a meta-level feedback loop from the GRA process of reengineering tool development into the design of reengineering platforms. The consequent necessary adaptability of the platform itself could well result from how it has been developed explicitly by an open extension process from a base language.

2.3.3. Summary. These requirements can be grouped under headings as follows.

- *Expressiveness* – very-high-level programming language, especially metalinguistic and transformational capabilities.
- *Persistence* – storage permitting user interaction/selection of operations on ASTs.
- *Adaptability* – incremental language transformation and modeling capability, and an open implementation architecture for the platform itself.
- *Open-ness* – in addition, recall that in addition to adaptability we have embraced the further aspects of heterogeneity, portability and accessibility.

3. Proprietary Reengineering Platform—Software Refinery

Software Refinery [7] was arguably an industry standard software reengineering platform, with numerous practical successes to its credit [8]. However, there are significant reasons for extensive review:

a) while its level of fulfillment of the above criteria is high, they are incompletely so, particular in respect of open-ness;

b) as a proprietary technology, Software Refinery (and its user base) is at the mercy of its developers (Reasoning Systems Inc.); in fact, they seem to have lost interest in the technology (see http://www.reasoning.com) for presumably commercial reasons.

In essence, Software Refinery consists of

- "Refine" programming language – Common LISP front-end incorporating wide-spectrum VHLL elements from logic, functional and OO paradigms;
- "object base" – providing object-oriented storage with persistence relative to LISP interactive sessions;
- "Dialect" – concrete/abstract syntax modeling;
- "Intervista" – GUI tool;
- all supported by interactive Common LISP development/execution environment.

4. Bahasa in Summary

"Bahasa" names our attempt to provide for Java a set of features meeting the reengineering platform requirements above. In detail, Bahasa was to recreate the essential functionality of Software Refinery in an open-technology, specifically Java-based, context, up to and including the extended metaprogramming capabilities of our Dialect/LXWB. A comparison to Software Refinery is given later. Bahasa is conceptually developed as a

sequence of preprocessors from Java incorporating successively constructs from Pizza, Refine and Dialect/LXWB.

4.1. Java as basis for development

Java offers an accessible, modern language design that not only already satisfies many of our reengineering platform design criteria but is moreover likely to inherit from other developments. Java is already:

- *Expressive* - generally, in that it directly supports object-orientation, and specifically for reengineering in that it supports metaprogramming (e.g. through the Java-CUP [12] compiler-compiler);
- *Persistent* – in that e.g. CORBA-compliant adapters exist;
- *Adaptable* – if necessary, Java's reflective capabilities could be used in developing extensions;
- *Open* – Java is at least portable/accessible.

The combination of these desirable properties also increases the likelihood that separate Java development efforts in parallel to ours will generate useful improvements that can be applied in the reengineering domain.

4.2. Pizza provides parameterised programming

The validity of the foregoing hypothesis was almost immediately proven when contemplating extensions to Java to support higher-order functions and data types as found in Software Refinery, when it was discovered that Pizza [13] went a long way to satisfying our requirements. Pizza is conceptually a Java preprocessor, extending Java to provide higher-order functions and data types (aka generic data types) and algebraic type specifications. Of these: higher-order/generic data types are most essential since they are required for implementing the Bahasa standard library of counterparts of Refine primitives. Higher-order functions are arguably indispensable because of their contribution to adaptability/extensibility in general, in particular because of their role in programming as prototyping [14] since they directly contribute to extensibility, and especially because of their applicability to advanced parsing methods [9, 10].

4.3. Refine-style transformation specification

Pizza/Java now serves as a basis for packaging and presenting additional constructs characteristic of Refine, a useful selection of which is as follows.

- *Generic data types:* following Refine, Bahasa supports definition and processing of generic sets, and a specialization thereof – sets of pairs or "maps" that can be applied like functions. For example:
  - "set <T>" is the type of a set of elements of type T
  - "{{X1, ..., Xn}}" defines a set by enumeration
  - "{{1 .. 10}}" is a set of the integers from 1 to 10
  - "S1 union S2" unites sets
  - "filter (P, S)" filters from S elements that don't satisfy P
  - "map <T1, T2>" is the type of a set of pairs with elements respectively of types T1 and T2
  - "M (X)" gives the second element of a pair in M for which X is the first element
  - "compose (M1, M2)" composes the maps Mi
  - "closure_under (M, S)" closes set S under map M

  Implementation of all the above in Pizza is straightforward.
- *Generic tree operations:* also following Refine, Bahasa predefines (higher-order) functions to package familiar operations and traversals of abstract syntax trees. For example, given that certain fields of specific classes can be restricted to maintaining a tree structure over the related instances (via application of a new "tree" modifier to their definitions):
  - "replace (X1,X2)" replaces the existing tree node X1 by X2 in situ
  - "parent (X)" returns the parent node of X in its tree (if it exists, otherwise null)
  - "descendants (X)" returns the set of all descendants of X (i.e. closure of the immediate subtrees)
  - "postorder_transform (X, F, B)" applies F to every descendant of X following a post-order traversal; reapplication of F to subtrees after a node is transformed is flagged by argument B ("preorder_transform" likewise).

  Implementation of all the above in Pizza is likewise straightforward.

4.4. Dialect-style language modeling

Bahasa realises the essentials of Dialect's language specification style by following Refine's exercise of the class mechanism for AST definition and overloading thereof for concrete syntax as well (see Introductory Bahasa Example above, but also see later for how ncremental syntax specifications are planned to be achieved). In this summary, we first focus on the challenge of how to achieve familiar concrete syntax functions with the overloading.

- *Nonterminal symbols:* each distinct syntactic category corresponds to a class.
- *Alternate productions:* alternatives for a syntactic category are expressed in terms of alternate subclasses for the corresponding class.
- *Concatenation within productions:* because syntactic categories correspond to attribute types (via the class-

type correspondence), and because the types of attributes are well-defined, the concrete syntax for an AST node can be specified indirectly in terms of its attributes rather than their types. The advantage of this approach (as in Dialect) is that no additional notation is required to derive concrete-abstract mappings.

- *Lexical analysis:* from the parser's point of view, terminal symbols are values from Bahasa primitive types. A generic lexer-parser interface that mimics Software Refinery's is being developed.

Naturally, there are pathological cases where the above general correspondences break down. Dialect's "local-nonterminal" and "semantics" constructs etc. which provide for these exceptions are effected in Bahasa by a combination of dummy classes and higher-order functions.

In addition to enhanced classes as above, a Bahasa "language model" also supports routine disambiguation of operator precedence and associativity, identification of start symbol, etc.

5. Introductory Bahasa Example

A standard demonstrator of the basic yet key components of Software Refinery that support metaprogramming, is the interactive calculator. Its Bahasa rendition is as follows, where *metacomments* about Bahasa interrupt the `Bahasa source code` as needed.

(Note – this presentation will be more immediately accessible to readers with a background in Software Refinery; see also the succeeding section for an alternative presentation summarising developments from Java.)

Bahasa modules are packages as in Java

```
package csm.bahasa.ready_reckoner;
```

import predefined entities

```
import net.sf.pizzacompiler.lang.*;
import net.sf.pizzacompiler.util.*;
import csm.bahasa.*;
```

a language-model declaration extends a class declaration with concrete syntax specifications

```
language_model Calc = {
```

fix operator precedence and associativity

```
precedence for expression
brackets "(" .. ")"
highest left associative "**", "/";
        left associative "+", "-";
```

miscellaneous lexical details

```
keywords case_sensitive
"float" may_replace "real";
```

start symbol for grammar (Refine-style)

```
file_classes calc_program
```

```
classes
```

the class of AST nodes for calculator expressions are a subclass of user_object, with any specific attributes to be provided by further subclasses

```
calc_ast = user_object { }

// type identifiers
```

types are represented in the AST, and each different type has a different concrete syntax

```
type = calc_ast { }
real_type = type{} ::= "real";
integer_type = type{} ::= "integer";

// identifier references
```

identifiers are represented in the AST with a symbol-typed attribute for their spelling; concrete syntax in each case is determined by the spelling; NB following Dialect, in order that the one specification define both concrete and abstract syntax, concrete syntax RHSs refer to attributes rather than attribute types (= nonterminal symbols)

```
identifier = calc_ast {
  symbol spelling;
} ::= spelling;
```

AST nodes for expressions have a single method/attribute – to evaluate the expression; the evaluator also takes a symbol table s (of type calc_ast->int mapping) and the default evaluator returns 0; concrete syntax for expression derives from that of each of its subclasses (as for types above)

```
expression = calc_ast {
  public int evalExp (map<calc_ast,int> s) {
    return 0;
  }
}
```

identifier occurrences within expressions combine an evaluation method with identifiers as above, but have the same basic concrete syntax.

```
identifier_expr = expression {
  identifier ident;
  int evalExp (map<calc_ast,int> s) {
    return s.get (ident);
  }
} ::= ident;
```

integer constants have value and evaluator attributes, and the concrete syntax of int(egers)

```
integer_constant = expression {
  int intv;
  int evalExp (map<calc_ast,int> s) {
    return intv;
  }
} ::= intv;
```

(the subclass of) binary_expressions have a pair of subtrees as attributes

```
binary_expression = expression {
  expression arg1, arg2;
}
```

add_expressions etc. are binary_expressions with an appropriate evaluator attribute and the usual concrete syntax

```
add_expression = binary_expression {
  int evalExp (map<calc_ast,int> s) {
    return arg1.evalExp(s) + arg2.evalExp(s);
```

```
  }
} ::= arg1 "+" arg2;

sub_expression = binary_expression {
  int evalExp (map<calc_ast,int> s) {
    return arg1.evalExp(s) - arg2.evalExp(s);
  }
} ::= arg1 "-" arg2;

mul_expression = binary_expression {
    int evalExp (map<calc_ast,int> s) {
    return arg1.evalExp(s) * arg2.evalExp(s);
    }
} ::= arg1 "*" arg2;

div_expression = binary_expression {
  int evalExp (map<calc_ast,int> s) {
    return arg1.evalExp(s) / arg2.evalExp(s);
  }
} ::= arg1 "/" arg2;

// statements
```

statements update a symbol table (= calc_ast->int map); hence the evaluator for a statement yields a new symbol table; default is empty map; specific attributes and concrete syntax derive from subclasses

```
statement = calc_ast {
  map<calc_ast,int> eval (map<calc_ast,int> s)
  {
    return new map();
  }
}
```

an assignment statement updates a symbol table by mapping its lhs attribute to the value of its exp, and the usual concrete syntax

```
assignment_statement = statement {
  identifier lhs;
  expression exp;

  map<calc_ast,int> eval (map<calc_ast,int> s){
    map<calc_ast,int> t =
      s.with(new tuple2(lhs,exp.evalExp(s)));
    return t;
  }
} ::= lhs ":=" exp;
```

the new map is s but "with" an additional identifier (lhs), value (exp) pair

a print_statement has a sequence of expressions as attribute, as well as an evaluator; the evaluator evaluates and prints the value of each expression and yields the symbol table unchanged

```
print_statement = statement {
  seq <expression> expressions;

  map<calc_ast,int> eval (map<calc_ast,int> s){
    for (
      seq<expression> e = expressions;
      !e.isEmpty();
      e = e.tail()
    ) {
      System.out.println(e.head().evalExp(s));
    }
    return s;
  }
}
```

the essence of print_statement concrete syntax is the comma-separated sequence of the concrete syntax of 1 or more constituent expressions, surrounded by "(...)" and preceded by keyword "print"

```
::= "print" "(" expressions +/ "," ")";

// declarations
```

declarations are a subclass of calc_ast, with any specific attributes to be provided by further subclasses

```
declaration = calc_ast {}
```

identifier_declarations have anonymous attributes of the given types "type" and "identifier"; references to these attributes (e.g. in concrete syntax) can simply cite the type names

```
identifier_declaration = declaration {
  type;
  identifier;
} ::= "var" identifier ":" type;

// program
```

a calc_program has attributes for its name, sequence of declarations, sequence of statements

```
calc_program = calc_ast {
  identifier program_name;
  seq <declaration> declarations;
  seq < statement > statements;
```

the evaluator for a calc_program applies a local method on its sequence of statements (below) to an initially empty symbol table

```
  void eval () {
    map<calc_ast,int> state = new map();
    evalStmts(statements, state);
  }
```

the local method exists to iterate evaluation over a calc_program's statement sequence

```
  static void evalStmts (
    seq<statement> stmts,
    map<calc_ast,int>
  ) {
    if (!stmts.isEmpty()) {
      map<calc_ast,int> t=
        stmts.head().eval(s);
      evalStmts (stmts.tail(), t);
    }
  }
}
```

the essence of program concrete syntax is 0 or more ';'-separated declarations followed by 0 or more ';'-separated statements, with the usual keywords; if 0 declarations/statements occur, the corresponding attribute values are the empty sequence.

```
::= "program" [program_name]
        declarations */ ";"
      "begin"
        statements */ ";"
      "end";

} //end Calc
```

6. Implementing Bahasa

While most of Bahasa is directly inherited from its Java/Pizza ancestors, innovations inspired by Re-

fine/Dialect/LXWB have to be accommodated. Some of these accommodations interact with Java/Pizza.

6.1. Overall architecture

The translation from Bahasa ultimately down to Java can be thought of as an instance of the GRA, additional complexity deriving from the multiple levels and the multiple technologies that coexist at some levels.

1. *Metaprogramming* originates from Bahasa, but with two targets: Java-oriented compiler-compiler "Java-CUP" to handle concrete syntax specifications; and an intermediate "Bahasa-minus" for the remainder of Bahasa.
2. *Persistence* can be implemented by next intercepting references to persistent classes and objects in the outputs of the above and translating them into IDL; however the current prototype uses "native" Pizza/Java objects and classes (i.e. standardised e.g. CORBA-compliant persistence is yet to be implemented).
3. *Refine-inspired extensions* beyond Pizza are translated (from Bahasa-minus) into Pizza.
4. *Parameterised programming* features provided with Pizza are handled by the Pizza-Java preprocessor.
5. *Java* forms the target that reconciles outputs from several of the above: Java-CUP from 1., IDL/Java combination from 2., Pizza-Java preprocessor from 4. Needless to say, great care has to be taken to ensure that the various to-Java translations are consistent, but no objective obstacle to this has as yet arisen.

6.2. Bootstrapping from Refine

In writing the preprocessors written by us and our colleagues we naturally are using Software Refinery as the best transition tool available. (This in the absence of Bahasa, but NB deferral of 2. means that its preferred implementation platform will be the non-persistent prototype implementation of Bahasa.)

Following usual bootstrapping practice, we write a Refine-Bahasa translator (in Refine – see more details below) and apply that translator to itself to produce Refine-Bahasa (in Bahasa). At present, Refine-Bahasa still requires a Reconciliation process, but once that is avoided further development of Bahasa (e.g. implementing persistence) could then be implemented in Bahasa.

The current implementation will be found at http://www.itee.uq.edu.au/~csmweb/

6.3. Refine-Bahasa

This is a straightforward exercise in translator-writing, with a minor optimization opportunity exploited in that whereas Dialect constructs translate into Bahasa, simple Refine constructs can and do translate directly into Bahasa-minus. A practical test of Refine-Bahasa was made in the generation of a COBOL design recovery tool. A language model for COBOL in Java was successfully generated from a Software Refinery language model for COBOL. The Java-based version was made operational with a minimum of Reconciliation.

6.4. Persistence

Early versions of the prototype involved a generator for CORBA IDL definitions, anticipating connection to appropriate persistent object stores. However, as mentioned above, this aspect of the platform has not been formalized or completed. This is discussed further in our comparison with Software Refinery below.

7. Comparison with other work

Other platforms, notably including Software Refinery and ASF+SDF, satisfy identified reengineering platform criteria, however we know of no effort explicitly to address all the criteria, nor moreover any effort focused specifically on a Java-based multiparadigmatic platform.

7.1. Software Refinery

7.1.1. Expressiveness. Software Refinery programmers are limited to a single wide-spectrum compromise between different paradigms. This is not to say that the existing compromise cannot be improved upon, nor that wide-spectrum languages have no place, but rather that a more robust solution to the expressiveness challenge would be to support heterogeneous interworking of different language technologies, each of which may give optimal support to different paradigms that may different roles in reengineering tool development. For example, a logic language may be a better basis for TS than the language used for OM and TM. Bahasa's complex of very-high-level and metaprogramming capabilities derived from Java, Pizza, Java-CUP and the projected standardized object store interface is evidently competitive with the reference point of Software Refinery, and even without the Refine patterns and transforms that are yet to be incorporated into Bahasa (and which have not seen the greatest of use in our experience). By way of compensation, Bahasa's high degree of faithfulness to the rest of Software Refinery means that incorporation of our Dialect/LXWB extensions into Bahasa is obviously feasible, and in our opinion would be of greater practical benefit.

7.1.2. Persistence. The persistence of Software Refinery's "object base" in which ASTs etc are stored is lim-

ited to individual LISP interactive sessions, unless the entire contents are explicitly saved. Saving object base contents is quite coarse-grained – individual objects may not be selected for saving in this manner. Moreover, the prospect of multiple users at least inspecting (perhaps not transforming) an AST is inconceivable from the Software Refinery point of view. A genuine, e.g. CORBA-compliant persistent store [11] would obviate these problems.

As acknowledged, persistence is yet to be implemented in Bahasa. This involves prescribing representations for Bahasa entities on disk or in memory to enable standardisation. However, several paths to standardisation are perfectly compatible with our Java basis and extensions (besides CORBA/IDL, there are XML options perhaps more worthy of consideration [20]). Moving persistence out of the closed environment of Software Refinery is not without potential drawbacks, e.g. the loss of uniformity of interactions in reengineering tool development and application, but against this must be offset the standard advantages of open systems: heterogeneity, portability etc.

7.1.3. Adaptability. The self-implementation of Software Refinery does achieve overall effects similar to an open implementation architecture. Organic changes to the platform (e.g. adding function-valued functions to Refine) can be achieved with the same ease with which reengineering tools can be developed.

Software Refinery offers however limited support for incremental language modeling in that structures for abstract syntax types may be synthesized using inheritance, but complementary facilities for incremental concrete syntax definition are inflexible. While syntax defined in one grammar may be reused in another, syntax can only be redefined at the level of granularity of a nonterminal. Concrete and abstract syntax must be located separately. We have accordingly developed a variant to Software Refinery's Dialect component – the "Language eXtension WorkBench" [9, 10]. Dialect/LXWB remedies the above infelicities respectively as follows: syntax may be redefined at the level of granularity of a single production, and concrete and abstract syntax may be defined together in a single "language model" construct. Moreover, LXWB provides additional alternatives for parser generation, including modular, general, recursive descent parsers.

Software Refinery's self-implementation within a closed interactive environment advantageously extends the tool development-application continuum established for persistence to include platform development as well. The Bahasa open implementation architecture seems to achieve at least comparable ultimate control over the platform's design and according implementation, if not as "comfortably" then certainly more "openly" than Software Refinery, again with all that implies for heterogeneity, portability etc.

7.1.4. Open-ness. Software Refinery drawbacks in adaptability and heterogeneity have been highlighted above. Further, not only does the platform run on a limited range of systems, but also applications require a proprietary run-time environment. Combined with commercial pricing, the result is severely to limit portability and accessibility. Adaptability aside, the remaining open systems benefits are all actually or potentially available in Bahasa to very significantly greater extents compared to Software Refinery:

- *Heterogeneity* will only be tested once CORBA-compliant persistence is implemented; however
- *Portability and Accessibility* are assured by using Java as development basis (and eventually a standard such as CORBA for persistence), and by self-implementation.

7.2. ASF+SDF

ASF+SDF [18] is a mature metalanguage featuring many-sorted logic and concrete syntax based transformations, which satisfies the reengineering platform criteria except for inadequacies pertaining to expressiveness.

7.2.1. Expressiveness. ASF+SDF has a fully featured metalanguage for defining algebraic terms and associated concrete syntax, with integrated lexical syntax definition and mechanisms for ambiguous languages. ASF+SDF provides for "equations" to be used in a declarative style to define evaluations or transformations on terms. Equations are defined via the concrete syntax of terms and admit pattern matching in the style of Scheme's metavariables [15]. ASF+SDF's current implementations support interpretation as well as compilation to C. However, as observed by Moreau et al [19], ASF+SDF is itself still a single-paradigm (rule-based) language which itself lacks features that would normally be provided by conventional imperative languages such as input/output or user-interface implementations, let alone the object-oriented domain modeling found in Bahasa as well as Software Refinery. This is compensated somewhat by ASF+SDF's integration with the "ToolBus" (see below) and the ability to compile to native C code. ASF+SDF's developers clearly express the need for increased expressiveness even within the transformational paradigm in the form of e.g. support for traversal functions and graphs within the language, as well as support for debugging and error messages [18]. On the other hand, both Bahasa and Software Refinery are already wide-spectrum languages by design.

7.2.2. Persistence. While persistence is not completely and explicitly provided for by ASF+SDF, it is certainly enabled via the ToolBus interface and ATerm support (see "Openness" below). Note though that the ToolBus itself is not a persistent store. The ATerm interface could be employed to implement persistence.

7.2.3. Adaptability. ASF+SDF has incremental language modeling abilities, and has been substantially bootstrapped, so provides all the features necessary for adaptability.

7.2.4. Open-ness. Heterogeneity is enabled via the "ToolBus" architecture [16] which provides a mechanism for interconnecting tools with widely different control and data mechanisms. Further, the ATerm protocol [17] used by ASF+SDF-generated tools provides a coarse-grained, standard mechanism for storing terms optimized both for human-readability (text) or space and speed efficiency (binary). Adaptors for ATerms exist for C and Java.

8. Conclusions

While several important details need to be resolved, the prospect of recreating proprietary reengineering platform functionality in an open, Java-based context seems substantially achievable. The strength of our approach derives as much from its grounding in a detailed domain analysis (i.e. of the reengineering process) as it does in its embrace of the "open" approach, including especially open-ness to parallel developments. Thus, even allowing for the possibility of infelicities in the detailed engineering of Bahasa, its merits may still be accessed.

9. Acknowledgements

We gratefully acknowledge the contributions to the development of Bahasa made by current and former colleagues, especially Paul Burnim, Ven-Yu Chong, Dan Johnston and Eric Salzman. Our use of Software Refinery was sponsored by the DSTO Australia.

10. References

[1] www.sei.cmu.edu/str/descriptions/cots.html/

[2] Weide, B.W., Heym, W.D. and Hollingsworth, J.E., *Reverse Engineering of Legacy Code is Intractable*, OSU-CISRC-10/94-TR55, Ohio State University, Columbus, Ohio 1994.

[3] Bailes, P.A., Atkinson, S., Chapman, M., Johnson, D. and Peake, I., "Towards an Open Software Conversion Architecture", *International Journal of Software Engineering and Knowledge Engineering*, 1995, pp. 423-444.

[4] McKeeman, W.M., "Programming Language Design", in Bauer, F.L. and Eickel, J. (ed.), *Compiler Construction - An Advanced Course*, Springer, Berlin, 1976, pp. 514-524.

[5] www.haskell.org

[6] Bailes, P.A., "The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design)", *Proc. 1986 Australian Software Engineering Conference*, Canberra, 1986, pp.14-18,.

[7] Reasoning Systems, *Refine User's Guide*, Palo Alto, 1992.

[8] Newcomb, P. and Markosian, L., "Automating the Modularisation of Large COBOL Programs: Application of an Enabling Technology for Reengineering", *Proc. Working Conference on Reverse Engineering*, IEEE, 1993, pp. 222-230.

[9] Peake, I. and Salzman, E.J., "Support for modular parsing in software reengineering", *Proc. Conf. Software Technology and Engineering Practice '97*, London, 1997, pp. 58-66.

[10] Peake, I., "Enabling meta-level support for language design and implementation through modular parsers", PhD Thesis, The University of Queensland, 2002.

[11] Soley, R.M. (ed.), *Object Management Architecture Guide*, Object Management Group, 1992.

[12] www.cs.princeton.edu/~appel/modern/java/CUP/

[13] Odersky, M. and Wadler, P., "Pizza into Java: Translating Theory into Practice", *Proc. POPL 97*, ACM, 1997, pp. 132-145.

[14] Hughes, J., "Why Functional Programming Matters", *Comput. J.*, vol. 32, no. 2, 1989, pp. 98-107.

[15] Kelsey, R. et al. (ed), "Revised report on the algorithmic language Scheme (1998)", *ACM SIGPLAN Notices*, 1998.

[16] Bergstra, J.A. and P. Klint, P. "The discrete time ToolBus – a software coordination architecture." *Sci. Comput. Prog.*, vol. 31, no. 2-3, 1998, pp. 205-229.

[17] van den Brand, M.G.J., de Jong, H.A., Klint, P. and Olivier, P.A., "Efficient Annotated Terms", *Software Practice & Experience*, vol. 30, no. 3, 2000, pp. 259—291.

[18] Van Den Brand, M.G.J., Heering, J., Klint, P. and Olivier, P.A., "Compiling language definitions: the ASF+SDF compiler", *ACM TOPLAS* vol. 24, no. 4, pp. 334-368, 2002

[19] P. Moreau and C. Ringeissen and M. Vittek, "A pattern-matching compiler", Workshop on Language Descriptions, Tools and Applications (LDTA), Electronic Notes in Theoretical Computer Science, vol. 422 (2001)

[20] http://www.gupro.de/GXL/