Developing a Translator from C Programs to Data Flow Graphs Using RAISE

Haxthausen, Anne Elisabeth

Published in:
Proceedings of COMPASS'96

Link to article, DOI:
10.1109/CMPASS.1996.507878

Publication date:
1996

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Developing a Translator from C Programs to Data Flow Graphs Using RAISE

Anne Elisabeth Haxthausen
Department of Information Technology
Technical University of Denmark, bldg. 344
DK-2800 Lyngby, Denmark
ah@it.dtu.dk

Abstract

This paper describes how a translator from a subset of C to data flow graphs has been formally developed using the RAISE method and tools. In contrast to many development examples described in the literature, this development is not a case study, but a real one, and it covers all development phases, including the code-generation phase. The translator is now one of the components of the LYCOS system, which is a software/hardware co-synthesis system under development at the Technical University of Denmark. The translator together with the other components of LYCOS provides a means for moving parts of C programs to dedicated hardware and thereby obtaining a better performance. The translator was refined in steps starting with an abstract specification and ending with a concrete specification from which C++ code was then automatically generated by the RAISE tools. In addition to illustrating the general methodology of RAISE, the paper also contributes with a specific method for refining set comprehensions.

1. Introduction

The reliability of software is an increasingly important demand in software engineering. One of the techniques proposed to increase the reliability is the use of formal methods. During the last decade several formal methods have been developed. However, their industrial usage is still limited. RAISE is a formal method which is intended for real industrial developments, not just toy examples, and it is currently being used by a number of companies and taught at universities. Examples of industrial applications are an automated train protection system made by Matra and a tethered satellite system made by SSI. Some of the industrial experiences are described in [5].

This paper presents how a translator from a subset of C to data flow graphs has been developed using RAISE, and the goal is to illustrate some of the features that make RAISE useful for the development of high-assurance systems. What makes this development example interesting is that it was not done as a case study, but because the author actually had the task of producing a translator and decided to use RAISE to increase the reliability of the translator. The paper illustrates the following features of the RAISE method: how to structure specifications to allow for separate development, how to refine abstract property-oriented specifications into concrete model-oriented ones, and how to handle a combination of manual and automatic translation into code. Below, the purpose of the translator is explained.

1.1. The purpose and context of the translator

For many systems it is crucial that the performance is high. One way to increase the performance of an existing software system may be to move time-consuming parts of the software to dedicated hardware. A commonly used method for deciding how a system can be partitioned in an optimal way into software and hardware is to translate the existing program into a data flow graph representing the computation of the program and then analyzing this. As many existing applications are written in C, it would be useful to have a translator from C programs to data flow graphs, and we have therefore developed such a tool. As these applications may be safety critical the reliability of the translator is very important.

The translator is one of the components of a hardware/software co-synthesis system named LYCOS, which is currently being developed at the Department of Information Technology at the Technical University of Denmark. (LYCOS is an acronym for LYngby CO-Synthesis system.)
aim of the system is to provide a number of tools for translating system specifications\(^2\) into data flow graphs, for partitioning a data flow graph into two communicating data flow graphs (for software and for hardware, respectively), and for transforming a data flow graph into software and hardware, respectively. Several of the tools have already been implemented and further information on Lycos can be found in [16].

1.2. Organization of the paper

The paper is organized as follows. First, in section 2, a survey of RAISE is given. Then, in sections 3–4, the requirements are stated and an initial specification of the translator is given. Next, in section 5, the specification is developed into a form which can be automatically transformed by the RAISE C++ code generator. After that, we explain how to generate C++ code for the final specification, and finally, in section 7, a summary and discussion is given.

2. RAISE

RAISE is an acronym for Rigorous Approach to Industrial Software Engineering and is a product consisting of a formal method, an associated formal specification language (RSL), support tools and documentation.

RAISE is the result of two ESPRIT projects carried out during 1985 - 1995 by companies from six European countries. The starting point for RAISE was the Vienna Development Method, VDM [2, 13], which had had success in industry, but lacked a number of useful features. Hence, the aim was to enhance VDM with structuring facilities, algebraic specification, concurrency, formal semantics and computer-based tools. Many languages and methods have been sources of inspiration for these enhancements, e.g. Z [1], ML [15], Clear [4], ASL [19], ACT ONE [6], LARCH [8], OBJ [7], CSP [12] and CCS [17]. A comparison of RAISE with other formal methods can be found in [9] and [20].

This section gives a short survey of the principles of the RAISE method and some of the language features. For more details we refer to [21] and [20]. The RAISE tools are described in [18].

2.1. The RAISE method

RAISE is based on the stepwise development paradigm according to which the software is developed in a number of steps. Each step starts with a description of the software and produces a new description which is more detailed. The specifications are formulated in RSL. The first specification is typically very abstract. After a number of design steps in which design decisions are made, one may obtain a specification which is sufficiently concrete to be (perhaps automatically) translated into a program.

The exact relationship of the specifications in a development is typically the predefined implementation (refinement) relation which stands for theory inclusion.

As a very important feature, the RSL structuring mechanisms, together with the implementation relation allow for separate development. For instance, assume that two modules, A and B, where B depends on A, are to be developed by two different teams. The initial versions of A and B are A\(_0\) and B\(_0\), respectively. One team refines A\(_0\) to A\(_m\) in \(m\) implementation steps, and another team refines B\(_0\) to B\(_n\) in \(n\) implementation steps, while still assuming the properties of A\(_0\), which acts as a contract between the two developments.

When the developments of the two teams are complete they integrate their developments by using A\(_m\) instead of A\(_0\) in B\(_n\) to form B\(_{n+1}\). Then B\(_{n+1}\) implements B\(_0\). Refinement is compositional: We can refine components separately and then integrate them to get a refinement of the whole specification.

Verification, or justification as it is called in RAISE, is rigorous (as the R in RAISE indicates): the method allows the verification to be formal but does not require it.

2.2. The RAISE Specification Language

The RAISE specification language, RSL, is a wide-spectrum language which encompasses and integrates different specification styles in a common conceptual framework. RSL enables the formulation of modular, structured specifications which are model-oriented or algebraic; applicative or imperative; sequential or concurrent. Below we give a short summary of some basic language constructs used in this paper. A detailed description of RSL can be found in [20].

2.2.1. Specifications

An RSL specification consists of module definitions. A module may define types, values, variables, channels and (sub-)modules, and may also present axioms.

2.2.2. Types

Types may be defined as sorts as known from algebraic specification, i.e. as abstract data types for which only a name is given:

\[
\text{type Id}
\]
Variables, as indicated by type. For example, in RSL, functions may access, i.e., read or write, declared model-oriented specification:

\begin{verbatim}
type Table = Id \rightarrow Int
\end{verbatim}

Types may also be defined as subtypes of other types, e.g.

\begin{verbatim}
type NegInt = \{ i : Int \mid i < 0 \}
\end{verbatim}

RSL additionally provides union and short record type definitions similar to those in VDM, and variant type definitions. For example, the variant type definition

\begin{verbatim}
type Colour == black | white
\end{verbatim}

defines a type \( \text{Colour} \) containing exactly two values \( \text{black} \) and \( \text{white} \).

2.2.3. Values

Values (constants and functions) can be defined in a signature/axiom style as known from algebraic specification:

\begin{verbatim}
value x, y : T
axiom x \neq y
\end{verbatim}

in a pre/post style:

\begin{verbatim}
value
  square_root : Real \rightarrow Real
  square_root(x) as y
  post y \ast y = x \land y \geq 0.0
  pre x \geq 0.0
\end{verbatim}

or in an explicit (signature/body) style as known from model-oriented specification:

\begin{verbatim}
value
  update : Id \times Int \times Table \rightarrow Table
  update(id, i, t) \equiv t \uplus \{ id \rightarrow i \}
\end{verbatim}

2.2.4. Variables

In RSL, functions may access, i.e., read or write, declared variables, as indicated by \texttt{read} and \texttt{write} clauses in their type. For example,

\begin{verbatim}
variable t : Table
value mk_empty : Unit \rightarrow write t Unit
\end{verbatim}

declares a variable \( t \) and a function \( \text{mk\_empty} \), which may write in \( t \). Instead of writing names of variables after the keywords \texttt{read} and \texttt{write}, one can write \texttt{any} to indicate that the function is allowed to read or write any variable.

The \texttt{Unit} type corresponds to the void type in C and is used as argument type for functions without parameters and as result type for functions which do not return any result.

2.2.5. Modularity

RSL has two kinds of modules, schemes and objects, as explained below.

Both kinds of modules are built from \texttt{class} expressions, where a basic class expression just embraces a set of definitions and axioms by the keywords \texttt{class} and \texttt{end}. A class expression denotes the set of all models concordant with its definitions and axioms (i.e., it has loose semantics).

RSL provides a number of class-building operators for renaming and hiding entities, for extending one class expression with another, etc.

A \textit{scheme} is a (generic) class and an \textit{object} is an instance of a class (i.e., denotes a single model from a class). For example

\begin{verbatim}
scheme S = class variable i : Int ... end
object M1 : S
object M2 : S
\end{verbatim}

declares a scheme \( S \) and two objects \( M1 \) and \( M2 \) which are distinct instances of \( S \). In other modules, the two distinct variables provided by \( M1 \) and \( M2 \) can be referred to as \( M1.i \) and \( M2.i \), respectively.

This example illustrates that if two modules use the same scheme \( S \), this gives rise to two \textit{copies} of \( S \). If instead two modules \( M1 \) and \( M2 \) are going to \textit{share} entities specified in \( S \), this can be achieved by defining an object, \( OS \), which is an instance of \( S \) and then letting \( M1 \) and \( M2 \) use the entities (e.g., \( OS.i \)) provided by \( OS \):

\begin{verbatim}
object OS : S
object M1 : class ... OS.i ... end
object M2 : class ... OS.i ... end
\end{verbatim}

(The same holds for schemes \( M1 \) and \( M2 \).)

3. Requirements

The overall aim is to produce a translator from a subset, SubC, of C to the kind of data flow graphs used in the LY-COS system. To be more precise, the system should provide a function, \texttt{translate}, which takes as input a well-formed\footnote{The well-formedness can be checked by existing C compilers.} C program. If the input belongs to the SubC subset of C, it should return a data flow graph (in a textual representation), which represents the computation of the program, otherwise it should produce an error message. The input and output should be text files.

The following two subsections give an informal description of the SubC subset of C and an informal introduction to data flow graphs, respectively.
3.1. SubC programs

There are two reasons for only supporting a subset of C. The first is that not all C programs are translatable due to limits in the expressive power of data flow graphs, and the second reason is that we want to solve the problem in stages, starting with a small subset of C and then later extending this subset. A SubC program consists of a sequence of declarations of global integer variables followed by a definition of a parameterless main function, which may access the global variables and which does not return any value (the result type is void). I.e. the form of a program is

\[
\text{int vl; \ldots; int vn;}
\]
\[
\text{main()} \text{ body}
\]

The body of the function definition is a compound statement of the form

\[
\{ \text{int vl; \ldots; int vn;}
\]
\[
\text{statement1 \ldots statementm}
\}
\]

where \(n \geq 0, m \geq 0\). A statement is an assignment, an if statement, a while statement or a compound statement. An expression is an integer constant, a variable name, a prefix expression or an infix expression.

3.2. Data flow graphs

This section provides a short informal introduction to the data flow graph model used in LYCOS and gives an idea of how they can represent C programs. Later, in section 4, a formal specification of an abstract syntax for data flow graphs is given. A more detailed description of the graphs and their computational semantics is presented in [3] and [14], and an informal description of how all SubC constructs can be translated is given in [11].

The purpose of data flow graphs is to represent computations of programs.

An example of a graph representing the computation of the C program

\[
\text{int x, y, z;}
\]
\[
\text{main()} \{ \text{ z = x + x * y; } \}
\]

is given in figure 1.

A data flow graph is a directed graph consisting of nodes and edges. The semantics is based on a token passing mechanism, similar to colored petri nets. The edges are entities on which tokens (i.e. values) can flow between nodes. Nodes can remove tokens from their input edges and place tokens on their output edges according to certain firing rules. There are different kinds of nodes, and they each have their own firing rules. The only kind of nodes shown in figure 1 are infix nodes, which have two input edges, one output edge and an associated infix operator. When an infix node, \(op\), has tokens, say \(v1\) and \(v2\), on its input edges and no token on its output edge, it can fire by placing the token, \(v1 \ op \ v2\), on its output edge, and removing \(v1\) and \(v2\) from its input edges. Other kinds of nodes are prefix nodes, constant nodes, control nodes to express conditionals and loops, void nodes to absorb tokens from edges etc. For more details on these, see [3] and [14]. A graph is executed by placing tokens on its input edges and letting the nodes fire until no more firing rules are satisfied.

Note that an edge can have more than one sink node. This is for instance the case for the \(x\) edge in figure 1, because its value is needed by the Add node as well as the Mult node. If an edge, \(ed\), has several sink nodes, the edge can be considered as split into several arrows, one for each sink. When a token is placed on the edge, each arrow gets a copy of that token, and it will not be considered empty until each of the sinks has removed its copy.

The graph in figure 2 represents the following C program

\[
\text{int a, x, y, z;}
\]
\[
\text{main()} \{ \text{ a = x * y; z = x + a; } \}
\]
If we compare this graph with the graph in figure 1, we note how the \( a \) edge now also appears as an output edge, even though it has a sink. That is because \( a \) is a global variable which has been written to, and we want to know its contents after the execution of the graph program.

A SubC program should be translated into a data flow graph representing the computation of the program, i.e. the graph should have an input edge for each variable that it might read before writing (i.e. making an assignment) to it, and an output edge for each global variable that it writes to. For any given values on the input edges, “execution” of the graph leads to values on the output edges, which are the same as the values the corresponding variables would have had after a call of the function, if the variables had the input values before the call.

There is a strong relationship between variables in the program and edges in the graph. At any point in the program, any variable in scope has a corresponding edge in the graph. When the graph is executed, the value transferred on that edge is the same as the value of the variable at that point in the program.

In LYCOS, there is also a textual representation of graphs. It is not presented here, as it is sufficient to know the graphical representation in order to read this paper.

4. Initial formulation

In this section the initial RSL specification of the translation system is created. A modular decomposition of the system into modules is made in the first subsection, then these modules are defined in the following subsections, and finally, in subsection 4.8, the correctness of the initial specification wrt. the requirements is discussed.

4.1. Modular decomposition

We aim at making a modular decomposition of the initial specification which can provide a good base for separate development.

We do this by investigating which functions the system should provide and which kind of data the functions should manipulate. The idea is then to have a function module for each collection of functions which conceptually belong together and a data type module for each main data type. These modules will typically be defined as schemes, because they allow for more reuse than objects do. Schemes are subject to refinement and we give them names of the form \( S_i \), where \( i \) is a number indicating at which development level the scheme is defined. If entities specified in a module, \( S_i \), are going to be shared by several other modules, \( A, ..., Z \), this can be achieved by defining an object, \( OS \), which is an instance of \( S_i \) (object \( OS : S_i \)), and then letting \( A, ..., Z \) use the entities provided by \( OS \). For such objects the name is retained throughout the development.

Clearly, there should at least be one function module: the system module, \( SYSTEM \), which provides the translation function, \( translate \).

We now look for a functional decomposition of \( translate \).

4.1.1. Functional decomposition of the translation function

The task of translating a C program into a data flow graph can be divided into three subtasks:

- parsing the C program (given in external text representation) into a parse tree
- translating the parse tree into an internal representation of data flow graphs
- unparsing the internal graph into a graph in the text representation used in LYCOS

This gives two (internal) data types, \( Progr \) and \( Graph \), one for parse trees and one for graphs, and it gives three functions, \( parse, tr, unparse \), one for each of the three tasks. The functional decomposition is shown in figure 3. This analysis leads to the decision to define two data type modules, \( PROGRAM1 \) and \( GRAPH1 \), which provide the data types \( Progr \) and \( Graph \), respectively, and three function modules: \( PARSER1, TRANS1 \) and \( UNPARSER1 \), which provide the functions \( parse, tr \) and \( unparse \), respectively.

In order to ensure that \( PARSER1 \) and \( TRANS1 \) share the data type \( Progr \), and that \( TRANS1 \) and \( UNPARSER1 \) share the data type \( Graph \), we define objects, \( C \) and \( G \), which are instances of \( PROGRAM1 \) and \( GRAPH1 \), respectively.

A sketch of the above mentioned modules is given below:

```
object C : PROGRAM1, G : GRAPH1
scheme
PARSER1 =
  class
    value parse : Unit \( \rightarrow \) read any Bool \( \times \) C.Progr ...
end,
```

Figure 3. Functional decomposition of translate

---

5 In the following, when we say read, we mean read before writing to it.
4.1.2. Decomposition of data types

In order to decide which additional data type modules to define, we investigate in figure 4 what the various kinds of data of the system are and what their relationships are. We have already seen that there are programs (parse trees) (type Progr) and graphs (type Graph). Programs contain variables (type VarId). Graphs are built from nodes (type Node) and edges (type Edge). Nodes may have associated operators (type InfixOp or PrefixOp), and edges as well as operators have types (type Type). As mentioned in section 3.2 there is a strong relationship between the variables of a program and the edges of the graph into which the program should be translated, and it turns out that in order to define the tr function, we need environments (type Env), which keep track of the relationship between variables and edges.

This analysis leads to the decision to define four additional data type modules, VAR1, EDGE1, TYPE1 and ENV1, which provide the data types VarId, Edge, Type and Env, respectively. In order to ensure that ENV1 and PROGRAM1 share VarId, that GRAPH1 and EDGE1 share Type, and ENV1 and GRAPH1 share Edge, the following objects are defined:

object V : VAR1, T : TYPE1, E : EDGE1

The modular decomposition made so far for the initial specification is shown in figure 5.

Figure 4. Data and their relationships

Figure 5. Modular decomposition of the initial specification

4.2. The PARSER and UNPARSER modules

The parsing is made using the well-known recursive descent parsing technique and the unparsing is straightforward, so in this paper we will not show the specifications of the parse and unparse functions.

The only difficulty in writing the parser and unparsers is that RSL does not provide IO functions. Therefore we specify input and output functions which should model IO functions in the programming language we are going to translate the final specification into (in our case C++). The functions can only be given signatures and must be translated by hand. They are therefore defined in a separate global object, IO:

object IO:

    class
    
    value
    
    input : Unit \rightarrow read any Char,
    output : Text \rightarrow write any Unit

end

4.3. The PROGRAM and VAR modules

A specification of variables and parse trees (abstract syntax trees) for SubC programs is given below:
scheme PROGRAM1 =
  class type
  Progr :: globals : Decls id : Text body : S,
  Decls = Decl*,
  Decs :: var : V.VarId type.of : Type,
  S == mk.Asg(var : V.VarId, expr : Expr) |
  mk.If(test : Expr, then.sen : S, else.sen : S) |
  mk.While(test : Expr, body : S) |
  mk.Block(decls : Decs, sens : S*),
  Expr == mk.Const(val : Int) |
  mk.VarRef(id : V.VarId) |
  mk.PrefixExpr(prefix_op : PrefixOp, expr : Expr) |
  ...
  PrefixOp == not | ...., InfixOp == add | ....,
  Type == integer
end

scheme VAR1 = class type VarId = Text ...

4.4. The GRAPH, EDGE and TYPE modules

An abstract property-oriented specification of graphs and edges, and a concrete specification of types are given below:
scheme GRAPH1 =
  hide is.wff in class
  type Graph = { g : Graph' * is.wff(g) }, Graph'
  value
  /* Graph observers */
  nodes : Graph' → Node-set,
  edges : Graph' → E.Edge-set,
  out-edges : Graph' → E.Edge-set,
  in-edges : Graph' → E.Edge-set
  in.edges(g) ≡ edges.with.no.source(g),
  is.wff : Graph' → Bool
  is.wff(g) ≡
  no.illegal.cycles(g) ∧
  (∀ n : Node * n ∈ nodes(g) ⇒ is.wff(n)) ∧
  (∀ ed : E.Edge * ed ∈ edges(g) ⇒
    card.sources(ed, g) ≤ 1) ∧
  edges.of.nodes(g) ⊆ edges(g) ∧
  out-edges(g) ⊆ edges.with.no.source(g),
  ...
  type
  Node ==
  Prefixnode(PrefixOp, E.Edge, E.Edge) |
  Infixnode(InfixOp, E.Edge, E.Edge, E.Edge) |
  Nopnode(E.Edge, E.Edge) |
  Voidnode(E.Edge) |
  ...
  PrefixOp == not | ...., InfixOp == add | ...
  value
  /* Node observers */
  is.wff : Node → Bool
  is.wff(α) ≡ ...
end

scheme EDGE1 =
  class type Edge
  value /
  edge observer */ type.of : Edge → T.Type
end

scheme TYPE1 =
  class type Type == integer | boolean | ...
end

The type Graph of graphs is chosen to be a sort. In this way the decision on how graphs should be represented has been deferred to a later stage of the development. A number of observer functions have been defined. They implicitly state that graphs are entities, which have nodes, edges, input edges and output edges. We also define the conditions under which a graph is well-formed: (0) it does not contain illegal cycles, (1) all its nodes are well-formed, (2) all its edges have at most one source, (3) the set of edges which have a sink or a source node is a subset of the edges of the graph, and (4) the output edges is a subset of the edges of the graph and a superset of those edges which do not have a sink. Some of these observers are derived, because they can be expressed in terms of the other functions.

The only non-derived Graph observers are nodes, edges and out.edges. Note, that out.edges(g) cannot be derived as edges.with.no_sink(g), since there may be output edges which have a sink, cf. the a edge in figure 2.

The Node type is defined as a variant type with one variant for each kind of node. Each variant has a constructor which produces a node of that kind. For instance, Prefixnode(op, i, a) is a prefix node with associated prefix operator op, input edge i and output edge a. The is.wff function defines under which conditions a node is well-formed.

The InfixOp and PrefixOp types are defined as variant types with one constant variant for each kind of infix operator and prefix operator, respectively. Operators are characterized by having certain argument and result types. A number of observer functions define these.

The Type type is defined as a sort having an observer which gives the type of the edge.

4.5. The ENV module

When translating a SubC program, each assignment, v = e, should cause a new edge for v in the graph, and when
translating a SubC value expression which reads a variable, \( v \), we need to know what the current edge of \( v \) is. Therefore, in the translation process, we need environments to keep track of (1) what the current edge of each variable is and (2) which edges have been used (so that we can generate new edges which have not been used before). The relationship between variables and their current edges is one-to-one as two different variables cannot have the same edge.

An abstract property-oriented specification of a data type \( Env \) of environments is given below:

```scheme
scheme ENV1 =
  hide map, used-edges, is-one-to-one, inverse-of in class
  type Env,
  Relation =
    \{ (m : V.VarId \rightarrow E.Edge \cdot is-one-to-one(m)) \}
  value
    /* non-derived Env observers */
    map : Env \rightarrow Relation,
    used-edges : Env \rightarrow E.Edge-set,
    /* derived Env observers */
    varids-of : Env \rightarrow V.VarId-set,
    edge-of : V.VarId \times Env \rightarrow E.Edge,
    edges : Env \rightarrow E.Edge-set,
    varid-of : E.Edge \times Env \rightarrow V.VarId
  post
    let ed1 = edge-of(v, pl) in
    ed2 \notin used-edges(\rho1) \land
    used-edges(\rho2) = used-edges(\rho1) \cup \{ed2\} \land
    E.type-of(ed2) = E.type-of(ed1) \land
    map(\rho2) = map(\rho1) \uplus [v \mapsto ed2]
    pre v \in varids.of(\rho1)
end
```

The type \( Env \) of environments is chosen to be a sort, in order to defer any decision of what their representation should be. A number of observer functions have been defined. As there is no representation for environments, the constructors are defined implicitly by predicates or post conditions. The post condition for an \( Env \) constructor, \( op \), states for each non-derived \( Env \) observer, \( obs \), what happens when \( obs \) is applied to an environment returned by the operation \( op \). If \( op \) also returns values of other types (e.g. \( E.Edge \)) the post condition also comprises similar conditions for these values.

### 4.6. The GRAPH\_WITH\_OPS module

When defining the translation functions in the TRANS module, it will be convenient to define these in terms of semantic operations on graphs (i.e. functions which take graphs as arguments and combine these into new graphs). We therefore define an extension, GRAPH\_WITH\_OPS1, of the GRAPH1 module with such operations, and replace our original definition of the object \( G \) with the following:

```scheme
object G : GRAPH\_WITH\_OPS1
```

This change of \( G \) illustrates how the process of creating the initial specification of a system typically consists of iterations. The revised modular decomposition of the initial specification is shown in figure 6.

![Figure 6. Revised decomposition of initial specification](image)

**GRAPH\_WITH\_OPS1** is defined as follows:

```scheme
scheme GRAPH\_WITH\_OPS1 =
  extend GRAPH1 with extend ENV1 with class
  value
    /* semantic operations */
    sequence : Graph \times Env \times Graph \times Env \rightarrow Graph
    sequence(g1, \rho1, g2, \rho2) as g
    post
      let ns1 = nodes(g1), ns2 = nodes(g2),
      i2 = in.edges(g2),
      o1 = out.edges(g1), o2 = out.edges(g2),
      written_in_both =
        written_in(g1, \rho1) \cap written_in(g2, \rho2),
      vs = written_in_both \setminus read_in(g2, \rho1),
      xx = edges.of(written_in_both, \rho1)
      in
        nodes(g) =
          ns1 \cup ns2 \cup
```
WITH-OPS 1 module, it is now possible to give explicit definitions of some operator nodes which convert the result from an abstract syntax tree for SubC constructs into graphs. The interpretation is given below:

As there are no constructor functions for graphs, each of the semantic operations is defined implicitly by a post condition. The post condition for an operation, op, states for each non-derived graph observer, obs, what happens when obs is applied to a graph returned by the operation op. The operations sequence, assign and variable_ref will be explained in connection with their use in the TRANS1 module.

In graphs, but not in C programs, there is a distinction between Booleans and integers. Therefore, when translating C programs to graphs, it is sometimes necessary to add constructor functions which translate parse trees (abstract syntax trees) into graphs. The specification is given below:

4.7. The TRANS module

Using the semantic operations provided by the GRAPH.-WITH-OPS1 module, it is now possible to give explicit definitions of the tr functions which translate parse trees (abstract syntax trees for SubC constructs) into graphs. The specification is given below:

A program is translated by translating its body statement in an environment which is obtained by translating its global
declarations in the initial environment. The translation of a list of declarations in an environment updates the environment such that each declared variable get an associated edge, which has not been used before. Statements and expressions are translated in an environment into a possibly updated environment and a graph, $g$, representing the statement/expression. Expressions additionally translate into an edge, the result edge, which is that output edge of $g$ on which the value of $e$ is to be found. (When expressions do not have side effects, as it is the case in SubC, the result edge will be the only output edge of the graph.)

A value expression which is a variable reference, $mk_{-}VarRef(v)$, is translated to the graph consisting of no nodes and only one edge, $ed$, which is the edge of $v$ in the current environment. The result edge is $ed$.

An assignment statement, $mk_{-}Asg(v, e)$, is translated by first translating $e$ and making any necessary type conversions on the result edge. In this way a graph, $g'$, with result edge, $res_{-}ed'$, is obtained. Then a new edge, $v_{-}ed$, for $v$ is created and the environment is updated accordingly. Finally, the $G_{-}assign$ operation (specified in GRAPH-WITH-OPS1) is used to combine $g'$ with a nop node whose input edge is $res_{-}ed'$ and whose output edge is $v_{-}ed$. A nop node is like a prefix node, whose associated function is the identity function. The nop node was added to combine the result edge $res_{-}ed'$ with the new current edge of $v$. Later, a graph optimizer could remove the nop node by identifying the two edges. This has for instance been done in the graphs shown in figures 1 and 2 in section 3.

A non-empty sequence of sentences is translated by first translating the first sentence of the sequence obtaining a graph, $g_1$, and a new environment, $pl_1$, and then translating the remaining sentences obtaining a graph, $g_2$, and a new environment, $pl_2$. After that $G_{-}sequence$ (specified in GRAPH-WITH-OPS1) is used to make a sequentially composition of $g_1$ and $g_2$ to obtain the resulting graph, $g$. $g$ consists of the union of the nodes and edges of $g_1$ and $g_2$ and some additional void nodes. The output edges of $g$ consists of the output edges of $g_1$ and some of the output edges of $g_2$. The output edges of $g_1$ can be divided into three groups: (1) those which are also input edges of $g_2$ (i.e. belong to variables which are read in $g_2$) and hence connect the two graphs, (2) those which belong to variables, which are neither read nor written in $g_2$, and (3) those which belong to variables which are not read but written in $g_2$. The edges from group (2) are the additional output edges, and edges from group (3) are those which the additional void nodes are voiding. An example of a translation of a sequence is shown in figure 7.

Other kinds of expressions and statements are translated in a similar way.

![Diagram](attachment:diagram.png)

**Figure 7. Translation of $x = y ; x = z$**

### 4.8. Correctness of the initial specification

Having developed the initial specification, the question is whether it satisfies the informally stated requirement in section 3 that a program should be translated into a graph representing the computation of the program. If there had been a common formal semantics for C programs and data flow graphs, i.e. functions $sem_1$ and $sem_2$ mapping values of type $Progr$ and $Graph$, respectively, into some semantic domain $Sem$, then we could have formalized our proof obligation as

$$\forall p : Progr \cdot sem_2(tr(p)) = sem_1(p)$$

and formally verified that this was true. However, for a non-trivial language, like C, it would be an enormous task to define its semantics, and we decided just to argue informally for the correctness.

### 5. Development

In this section we aim at developing the initial RSL specification into a new RSL specification which is sufficiently concrete so that almost all of it can be automatically translated into C++ by the RAISE C++ code generator. If there are parts of the specification which need to be translated by hand, these must be localized in separate modules. (This is a requirement by the C++ code generator.)

In the initial specification the following non-translatable RSL constructs appear:

1. sorts and/or implicit value definitions (in EDGE1, ENV1, GRAPH1, GRAPH-WITH-OPS1)
2. set comprehensions (in GRAPH-WITH-OPS1)
3. class extensions (in GRAPH-WITH-OPS1)

We remove these constructs step by step in the given order.

#### 5.1. Removing sorts and implicit value definitions

##### 5.1.1. Development of the EDGE module

We develop the EDGE1 module by replacing the sort definition of $Edge$ and its observer in EDGE1 with a short record type obtaining a new module EDGE2:
scheme EDGE2 =
  class type Edge :: num_of : Nat type_of : TType end

The RAISE tools have been used to justify that EDGE2 implements EDGE1. We now replace EDGE1 with EDGE2 in the definition of E:

object E : EDGE2

As EDGE2 implements EDGE1 replacing EDGE1 with EDGE2 will not affect the modules referring to E.

5.1.2. Development of the ENV module

We now develop the ENV1 module into a new module, ENV2, by replacing the sort Env with a concrete type and replacing the implicit value definitions with explicit value definitions in such a way that ENV2 implements ENV1. The principles for doing this are the same as for the development of the GRAPH and GRAPH-WITH-OPS modules (described below) and we will not show the details here.

5.1.3. Development of the GRAPH module

We now develop the GRAPH1 module by replacing the definitions

type Graph = { g : Graph' * is.wff(g) }, Graph'
value
  nodes : Graph' → Node-set,
  edges : Graph' → E.Edge-set,
  in-edges : Graph' → E.Edge-set,
  out-edges : Graph' → E.Edge-set

in GRAPH1 with

type Graph = { g : Graph' * is.wff(g) }, Graph'
value
  nodes : Graph' * is.wff(g)
  Graph'' =
    { g : Graph'' * in-edges(g) = edges.with_no_source(g) },
  Graph'' ::
    nodes : Node-set edges : E.Edge-set
    in-edges : E.Edge-set out-edges : E.Edge-set

obtaining a new module GRAPH2.

As the input edges of a graph can be derived by inspecting the nodes and edges of the graph, it is not necessary to include the in-edges component in the representation. However, we decided to include it, as the unparser needs this component (since the Lycos textual representation of graphs has redundant information concerning the input edges), and it is more efficient to calculate it on the fly in the translation process, where the graph is built by combining sub-graphs, than calculating it afterwards. Note how the inclusion of the in-edges component in the representation implies the need for an additional subtype.

Instead of using sets to represent collections of nodes etc., one could have used lists:

Graph' ::
  nodes : Node* edges : E.Edge*
  in.edges : E.Edge* out.edges : E.Edge*

However, there is no reason for using lists rather than sets. The RAISE C++ code generator is able to translate lists as well as sets, and the produced code would not be more efficient for lists than for sets as they are both translated into linked lists. There is actually a good reason for using sets and not lists. Using lists would require more work than using sets, as one then would have had to define additional operations like union and intersection, which are built-in for sets.

The RAISE tools have been used to generate and justify conditions which ensure that GRAPH2 implements GRAPH1. For example the following condition was generated and immediately reduced to true by a simplifier:

\[ \forall g : Graph' \cdot in.edges(g) \equiv edges.with_no_source(g) \]

simplify :

[true]

qed

5.1.4. Development of the GRAPH_WITh.OPS module

If we in the GRAPH_WITh.OPS1 module replace GRAPH1 with GRAPH2 we can then also replace the implicit definitions of the semantic operations with explicit definitions. Furthermore, we integrate the development of the ENV module by replacing ENV1 with ENV2. In this way we obtain the following new module.

scheme GRAPH_WITh.OPS2 =
  extend ENV2 with extend GRAPH2 with class
value
  /* semantic operations */
  sequence : Graph × Env × Graph × Env → Graph
  sequence(g1, p1, g2, p2) =
    let
      ns1 = nodes(g1), ns2 = nodes(g2),
      i1 = in.edges(g1), i2 = in.edges(g2),
      o1 = out.edges(g1), o2 = out.edges(g2),
      connected = o1 \ i2,
      written_in_both =
        written_in(g1, p1) \ written_in(g2, p2),
      vs = written_in_both \ read_in(g2, p1),
      xx = edges_of(written_in_both, p1)
    in
      mk_Graph''( ns1 \ ns2 \ U
        { Voidnode(edge.of(v, p1)) | v : V.VarId * v \ vs },
        edges(g1) \ edges(g2),
        i1 \ (i2 \ connected),
        o2 \ (o1 \ xx))
  end
  pre ..., end

end
The RAISE tools have been used to justify that GRAPH\_WITH\_OPS2 implements GRAPH\_WITH\_OPS1. As GRAPH\_WITH\_OPS2 implements GRAPH\_WITH\_OPS1 we can now replace GRAPH\_WITH\_OPS1 with GRAPH\_WITH\_OPS2 in the definition of G:

\[
\text{object } G : \text{GRAPH\_WITH\_OPS2}
\]

5.2. Removing set comprehensions

Set comprehensions have been used extensively in the specification because they provide an elegant and short way of constructing sets based on properties of their elements, but they are not in the translatable subset of RSL. Therefore we must refine them into other RSL constructs that are translatable. No advice on how to do this is offered by the RAISE method book [21]. Here we propose a systematic approach to refining set comprehensions.

5.2.1. General method

A set comprehension of the form

\[
\{ f(x, y_1, \ldots, y_n) \mid x : X \times X \in xset \}
\]

where \( X \) is some type, \( xset \) is an expression of type \( X\text{-set} \), \( f \) has type \( X \times Y_1 \times \ldots \times Y_n \rightarrow Y \), and \( y_1, \ldots, y_n \) are free names, may be replaced by a function application

\[
\text{comprehend}(xset, y_1, \ldots, y_n)
\]

if the following definitions are added:

\[
\begin{align*}
\text{value} \\
\text{comprehend} : X\text{-set} \times Y_1 \times \ldots \times Y_n \rightarrow Y\text{-set} \\
\text{comprehend}(xset, y_1, \ldots, y_n) & \equiv \\
\text{if } xset = \{ \} \text{ then } \{ \} \\
\text{else let } x = \text{pick}(xset) \text{ in} \\
& \{ f(x, y_1, \ldots, y_n) \} \cup \\
& \text{comprehend}(xset \{ x \}, y_1, \ldots, y_n) \\
\end{align*}
\]

\[
\text{pick} : X\text{-set} \rightarrow X \text{-set}
\]

\[
\text{pick}(xset) \equiv \text{let } x : X \times x \in xset \text{ in } x \text{ end}
\]

\[
\text{pre } xset \neq \{ \}
\]

\( \text{pick} \) is a function which takes a set as argument and returns non-deterministically one of its elements.

In [10] it is proven that this development step is an implementation, when \( xset \) is convergent and of type \( X\text{-set} \).

The \text{comprehend} function is translatable by the RAISE code generators, but \( \text{pick} \) is not. However, \( \text{pick} \) can easily be translated by hand, since the C++ class which \( X\text{-set} \) is translated into, provides a function, which given a set returns one of its elements.

5.2.2. Development of the GRAPH\_WITH\_OPS module

An example of a set comprehension in module GRAPH\_WITH\_OPS2 is

\[
\{ \text{Voidnode}(\text{edge-of}(v, p)) \mid v : \text{V.VarId } v \in \text{vs} \}
\]

This can be replaced by \text{Voidnodes}(vs, p) if the following definition is added to the module

\[
\text{Voidnodes} : \text{V.VarId-set } \times \text{Env} \rightarrow \text{Node-set}
\]

\[
\text{Voidnodes}(vs, p) \equiv \\
\text{if } vs = \{ \} \text{ then } \{ \} \\
\text{else let } v = \text{PICK.pick}(vs) \text{ in} \\
& \{ \text{Voidnode}(\text{edge-of}(v, p)) \} \cup \\
& \text{Voidnodes}(vs \{ v \}, p) \\
\end{align*}
\]

and \( \text{PICK} \) is a module defined as follows

\[
\text{object } \text{PICK} : \\
\text{class} \\
\text{value} \\
\text{pick} : \text{V.VarId-set } \rightarrow \text{V.VarId} \\
pick(vs) \equiv \text{let } v : \text{V.VarId } v \in vs \text{ in } v \text{ end} \\
\text{pre } vs \neq \{ \}
\]

All other set comprehensions in the GRAPH\_WITH\_OPS2 module should be refined in a similar way. In this way we obtain a new module, GRAPH\_WITH\_OPS3, which is an implementation of GRAPH\_WITH\_OPS2. The RAISE tools can be used to justify that.

As GRAPH\_WITH\_OPS3 implements GRAPH\_WITH\_OPS2 we can now replace GRAPH\_WITH\_OPS2 with GRAPH\_WITH\_OPS3 in the definition of G:

\[
\text{object } G : \text{GRAPH\_WITH\_OPS3}
\]

The configuration of the specification at this point is shown in figure 8.

**Figure 8. Intermediate specification**
5.3. Removing class extensions

Current limitations of the C++ code generator means that class extensions must be removed before translation. Therefore we must do a step in which we expand the right-hand side of GRAPH.WITH.OPS3 to a basic class expression obtaining a new module GRAPH.WITH.OPS4 which obviously implements GRAPH.WITH.OPS3.

As GRAPH.WITH.OPS4 implements GRAPH.WITH.OPS3 we can now replace GRAPH.WITH.OPS3 with GRAPH.WITH.OPS4 in the definition of G:

\[ \text{object } G : \text{GRAPH.WITH.OPS4} \]

5.4. The final configuration

The configuration of the final specification is shown in figure 9.

![Figure 9. Final specification](image)

6. Generating C++ code

Having developed the specification as far as explained above, it can now be translated to C++. This is done by translating the IO and PICK modules by hand to C++ modules IO.h and PICK.h, while the remaining modules can be translated automatically by the RAISE C++ code generator.

In order to get a running C-to-data-flow-graph translator, one now only has to write a trivial C++ main function that calls the C++ function, SYSTEM.translate, which the RSL SYSTEM.translate function was translated to.

7. Summary

It has been illustrated how a translator from C programs to data flow graphs can be developed stepwise and separately from an abstract property-oriented specification into a concrete model-oriented one using the RAISE method. The RAISE tools have been used to syntax and type check the specifications, to generate and justify the conditions that the development steps are implementations, and to translate the final specification into a C++ program.

In addition to illustrating the general methodology of RAISE, a specific method for refining set comprehensions has been proposed.

My experiences from this development example was that the following features of RAISE were useful:

- The module concept which made it possible to decompose the specification into small manageable units which I could develop separately.
- The stepwise development principle together with good abstraction facilities, which made it possible to cope with details one at a time. For instance, in the initial specification I could use abstract data types (for edges, graphs and environments) and first in later development steps make a design decision on the data type representations.
- The formal basis, which made the meaning of specifications unambiguous and allowed formal verification of the development steps.
- The rigour, which allowed me to use informal arguments in the verification, whenever I found that sufficient. (To have formally verified everything would have been too time-consuming.)
- The tools support, which I used to
  - eliminate syntax and type errors in specifications
  - justify (verify) the development steps faster and with more confidence than possible by hand
  - generate C++ code

The C++ code generator would have been even more useful if it had been able to handle a larger subset of RSL. For instance, it should have been able to handle class extensions such that the last development step would not have been necessary.

These features are not only beneficial for the presented development example, but are general features that make RAISE useful for the development of high-assurance systems. In particular, the use of formal (and thereby unambiguous) specifications and the use of formal verification in the development process increase the reliability of the produced software.
7.1. Acknowledgements

The author would like to thank Chris George, Bo Stig Hansen, Jan Madsen, Jan Storbæk Pedersen, Jørgen Staunstrup and the anonymous referees for valuable comments to a draft of this paper.

The work described in this paper has been supported by the Danish Technical Research Council under the “Code-sign” programme.

References