

The Use of Autocoding in the Support of Application Specific Data Structures in GEDAE

M. J. T. Alpey
BAE SYSTEMS Avionics
Sensor Systems Division,
Crewe Toll House, Ferry Road,
Edinburgh EH5 2XS, U.K.
(Tel. +44 131 343 4273)

marcus.alpey@baesystems.com

ABSTRACT

When combining traditionally-developed software application code with implementations based around alternative design representations there is often a need to provide interface support for application specific data structures. Integration of such structures within GEDAE flowgraphs can be an expensive task due to the time required to develop the necessary library of boxes to provide this level of support. Furthermore, additional boxes will be required to control the flow of these custom-designed data structures around the model.

This paper addresses the problems associated with the creation of a library of support boxes. These problems are primarily the development time, the diverse range of boxes desirable and the possibility of coding errors within the developed library. Both the cost and the risk associated with these problems can be reduced by automatically generating the appropriate boxes directly from the data structure definitions.

Automatic code generation offers a fast, flexible, extendable and consistent means by which to create a library of interface support and dataflow control boxes. Boxes thus generated can be updated quickly, easily and in a controlled and repeatable manner should the structure definitions themselves change, or should the required coding change.

The libraries created can be tailored by offering optional or configurable generation of specified boxes in a hierarchical directory structure that is consistent with GEDAE's own supplied embeddable library. This facilitates the finding of boxes within the library, thereby improving its useability.

The automatic code generators can be configurable and portable to ensure that their use is not restricted to specific data structure representations or development platforms. Parsing modules for various source languages can be written and integration into the generators is simplified if a modular design approach is adopted. Implementation in PERL makes their domain of applicability and use the same as that of GEDAE.

An automatic code generator has been developed and used on the Captor Tranche 2 programme – a radar project implemented in GEDAE that makes use of legacy code from the original Captor programme. The data structures to be supported for Captor Tranche 2 are hierarchical compound datatype definitions from C header files, based on those from legacy code.

This paper describes the points that should be considered in the development of such an automatic code generator, the range of boxes that can be generated and the methods by which this is achieved. Experiences of the use of this automatic code generator on the Captor Tranche 2 project are presented, and examples of the types of boxes that can be produced are given, along with the definitions from which they were generated. Areas in which the automatic code generator may be further extended are identified.

Keywords: GEDAE, application code, data structures, support, library, automatic code generation, PERL

1. INTRODUCTION

Application-specific data structures are often used to simplify the transfer of particular sets of related values within a software system, by grouping them into a single, named entity. This entity can then be transferred as a unit instead, and the individual components accessed or extracted as required. Such techniques are often used for grouping control values into a message, rather than having to communicate each value individually.

Since the content of these data-structures is entirely dependent on the application, software-modelling tools cannot provide any form of tailored support for them. This leaves the task of producing any such required support to the developer, thereby adding to the effort of the overall software development. However, the likely forms of support that will be required can be identified in advance and are fairly generic in nature. This suggests that something could be done to reduce the burden of having to manually generate the desired functionality, by intelligently automating the process.

The feasibility of such automation obviously depends on the specific support required for the chosen software-development tool. In GEDAE, the required support would be via the provision of customised processing boxes, tailored for the individual application-specific data structures. The ASCII-based format of GEDAE's processing boxes makes them very amenable to automatic code generation – a concept at the very core of the tool.

1.1 GEDAE and Data Structures

GEDAE already provides an embeddable library containing a large number of boxes for processing and controlling the flow of data of the types that it directly supports – ints, floats and complex values. When using application-specific datatypes and data structures, a developer will normally require a similar library of boxes that offers the same range of functionality but tailored for each of the datatypes and data structures to be used. This set of boxes can be classified into a number of different categories: flow control (which are usually referred to as '*logic*' boxes within GEDAE), data structure manipulation and input/output boxes. Since all of these categories of boxes may be required for each of the types or structures being used, the number of support boxes to be generated can be quite large.

1.2 Automatic Code Generation

Since the code structure of the required boxes will be the same as that of any corresponding int, float or complex library components, many of the requisite boxes could be produced by modifying their existing counterparts from the supplied embeddable library. The only differences in the box code would be due to the details of the application-specific data structure in question.

Some boxes would be more complicated to modify than others would and may require considerable amounts of code to be written or re-written. Other boxes would have no corresponding embeddable library component and would need to be written from scratch. Even here, the layout of the code would normally still be consistent for a particular type of box

across all of the various data structures for which it would be required. The only differences would be due to the details of the data structure being supported.

In all cases, making such changes manually would be both time-consuming and fraught with the possibility of the developer introducing a coding error. A lot of work could be saved by automatically generating the code for the application-specific data structure support instead. Furthermore, such automatic code generation would greatly reduce the time taken to produce the library of boxes and also reduce the risk of typographical coding errors.

2. CONFIGURABLE AUTOMATIC CODE GENERATORS

When producing a utility to perform automatic code generation, some thought must be given to the design to ensure as wide a scope as possible for its use. This requires considering the platforms on which it must run, the data which it must process, the output it must be able to produce and any other information that it will require to permit it to carry out its function.

This utility is intended to be run alongside GEDAE and will be processing text files (the data structure definitions) to produce more text files (the library of data structure support boxes). Consequently, the use of PERL – a scripting language, well suited to text-based processing and available on a wide variety of platforms – seems a good choice for the programming language in which to write the automatic code generator. Furthermore, GEDAE itself uses PERL to carry out much of its autocoding and other text-based processing, so it can be guaranteed to be available on every system that supports GEDAE.

Although in principle there is no restriction on the source language of the original data structures, the easiest one to use is C, since any other language will have to be translated to produce a C-compatible version that can be used within the GEDAE primitives. Consequently, the primary parsing routines should be for C-based data structure definitions. These should be kept modular so that alternative parsing modules can be written and used instead if this becomes desirable at some point in the future. Note that C bit-field constructs are not supported by GEDAE, so there should be an option to have these converted into something useable, *e.g.* a whole integer, instead. This will result in a modified data structure definition file that will then be used as the source for the box generation routines.

The file containing the original data structure definitions should be supplied as an argument to the automatic code generator – this allows different sets of data structures to be processed easily, without the need to move or copy their definitions to specific locations. It should also be possible, however, to assume a default location if one is not supplied.

Since the expected level of use of the boxes cannot always be predicted at the time of their generation, it is often desirable to generate all of the possible boxes and then make use of only those required during the development of the project. However, for models with many application-specific datatypes, this can result in a very large number of boxes being

produced. If it can be determined in advance that some types of box will not be required, then it should not be necessary to generate them.

The automatic code generator should therefore accept a set of command line arguments specifying which boxes should be produced, or allowing particular variants of certain boxes to be generated. This allows the library generated to be tailored to include only the required boxes, thereby keeping the size of the library small. It can also be advantageous to have this option in case a change or an improvement is made to a code template or rule as it means that only those affected boxes need be regenerated.

The generator should also be able to process only a named data structure out of a source file, again to avoid unnecessary re-generation of boxes whose definitions have not changed.

Other information that may be required by the generator includes the location of the GEDAE supplied library. This must be specified to allow the scripts to access the standard (or customised) box template files – however, a default location can be assumed should the information not be explicitly provided.

The root of the directory tree into which the automatically generated code will be written should also be specifiable to allow the user to maintain separate libraries if so desired – again, a default location can be assumed if one is not explicitly provided. Below this root level, a directory will be created for each of the different data structures, and each of these directories will have their own sub-directories, matching the hierarchy found in the supplied GEDAE library with regard to logic, source and sink boxes.

Combinations of the arguments allow a very flexible degree of control to be exercised over which boxes are to be generated.

3. PARSING DATA STRUCTURE DEFINITIONS

Automatic code generation requires certain information about the data structure being processed to correctly customise the template code. The necessary information can be gathered by parsing the data structure definitions in the source file.

The information required by the automatic code generator is as follows :

- The type-name of the compound data structure
- The number of fields or elements that it contains
- The types, names and dimensions of all of those fields
- The names and values of any program constants used in defining the dimensions of any of the fields
- The details of, and dependencies for, any nested sub-structures

When processing a data structure that contains an (optionally multidimensional) array, the sizes of this array must be carefully processed to allow for a variety of ways in which these may have been defined. The sizes may be explicitly specified as numeric values, or using program constants, or as arithmetic expressions containing a mixture of the two. Any

program constants used must be defined elsewhere in the file containing the data structure definition or in one of the files that it includes.

In either case, the automatic code generator needs to know not only the names and values of the program constants used, but also the expressions in which they are used so that the correct substitutions can be performed in the appropriate places.

In order for a data structure to make use of other datatypes or data structures, their definitions must already be known. This is normally achieved by ensuring that the sub-structure definitions are read first by placing them earlier in the source file, or in a separate file that is included prior to any definitions in the current file. When including other files, it is better if their locations are given relative to the current one. This makes the code more portable by not requiring specific fixed locations for any of the files concerned.

Clearly, where datatypes and data structures from different projects are being combined in a single GEDAE model, great care must be taken that there are no duplicate program constant, datatype or data structure names as this could lead to compilation difficulties or to unexpected results.

4. AUTOMATIC CODE GENERATION

Automatic code generation, or autocoding, is the process of creating compilable code from a set of coding rules or template code and a database of information about the code elements or variables. The basic premise is to insert the types, names, sizes or values of the appropriate code elements at identified places in the coding rule or template code.

In a GEDAE context, the coding rules and template code will be used to construct the processing boxes. The primary difference between the types of boxes is whether they are flowgraphs or primitives, although there are also different varieties of primitives. Since both flowgraphs and primitives have an ASCII-based representation, similar techniques can be used for the automatic generation of either sort.

4.1 Flowgraphs

Flowgraphs are a graphical way of describing the required connectivity between a set of primitives. As such, they do not actually describe any processing, which is all carried out within primitive boxes. They may, however, contain data declarations but these are not directly required in the autocoded support of data-structures.

The primitives used in the autocoded data structure support flowgraphs are also automatically generated, so the names of these boxes and all of their inputs and outputs are known. The content of flowgraph boxes consists of a listing of the inputs, locals and outputs as well as the set of boxes used (be they primitives or nested flowgraphs), their connections and placement information. Due to the nature of the flowgraphs being generated, the connections are usually fairly simple – a one-to-one mapping between one box's outputs and another's inputs.

When writing a flowgraph file, the automatic code generator must work out the required positions of the inputs, outputs and boxes on the canvas making sure that they are appropriately spaced – non-overlapping and neatly aligned. It must then fill

in these co-ordinates at the appropriate points for the box placement, and for any corners used in routing their connections. Calculation of sensible co-ordinates is simplified by the fact that the units used correspond approximately to the size of an individual character in the default font used to display box and connection names. This means that box dimensions can be calculated by assessing the string lengths of the box title, the longest input and output names and the greatest number of inputs or outputs.

Finally, the flowgraph canvas and window sizes must be set. The canvas size should be as small as possible while being large enough to hold all of the boxes and the window size should show as much of the canvas as possible within the constraints of a reasonable screen size or resolution.

4.2 Primitives

There are different types of primitive box that may be required in providing the full range of functionality for a data structure support library. All of these different types of primitive consider the data structures in one of two ways. These two views, either using the data structure as an entity or as a collection of elements, give rise to differences in the code structure required to provide the necessary functionality. In turn, these differences lead to variations in the processing required to autocode these primitive boxes.

1) Template-Based Boxes

The code for a particular type of box in this class differs only in the type of the token being processed. This means that the boxes can be produced from a piece of template code that has a placeholder for the datatype or data structure name and for the name of the box. This template code contains all of the necessary Method sections, any required header information and appropriate commenting. Some boxes additionally require that the names of inputs and outputs be generated in a way that incorporates the datatype or structure name – again, placeholders can be used to accommodate this where necessary. The automatic code generator then substitutes these placeholders accordingly when processing the templates.

2) Script-Generated Boxes

The boxes in this class all make use of the various fields or elements of the data structures they are processing. In generating the code for these boxes, there are normally set pieces of template code that can be copied at the start and end of each Method section. In between is a variable part where the appropriate information about the data structure must be looked up from the database that was created when the original structure definition was parsed.

Each of the sections is processed in turn – different box types may require different Methods. The automatic code generator can be designed such that it has a separate subroutine for each type of box that determines which sections are required and how they must be modified for that particular type of box. Some will require Local and Reset Methods, and others may not require an Input or an Output Method.

The name of the Method and any fixed template code at the start is written first. When processing the variable part, the script iterates through each of the structure elements in turn, writing the necessary code and substituting the appropriate types, names, sizes and any other required information. Note that different pieces of information about the elements may need to be written into different sections – e.g. Input Methods will need to list any parameters required to size their output streams. Finally, any fixed template code and the end of Method will be written.

4.3 Array-sizing considerations

Care must be taken when generating primitives from data structures that contain array elements. In the data structure, these array elements will be sized using program constants from the source file. In GEDAE, the sizes of input streams are inherited from the upstream box that sources the stream, and output streams can only be sized by incoming dataflow controlling parameters or input parameters, both of which must be fixed prior to scheduling.

This means that boxes that receive such sized input streams can do nothing to modify the dimensions of that input, and must trust that it has been correctly set up by its source. Primitives that wish to output a sized stream must provide a parameter input for each of the named values to be used in sizing any of the outputs and then construct the output dimensions from the appropriate combinations of these names.

Passing this information as parameter inputs is not a convenient way to incorporate the required information into the model, particularly since the definition of the values is already available in the file providing the data structure definition. Furthermore, it is desirable to be able to explicitly define these required values within the primitive for a number of reasons. Firstly, this keeps the definition of the values visible to the code that makes use of them. Secondly, failure to do so means that any changes to these values must now be maintained in two locations – the data structure definition source file (or the appropriate included file), and the flowgraph canvas on which this primitive is used (or another box on that canvas). Finally, if the values are not assigned locally there can be no guarantee that the intended values will be used, which could lead to processing problems such as invalid memory accesses.

However, there is a way around this problem that is amenable to the techniques used to create the boxes. Dummy parameter inputs are generated for each of the non-numeric sizes (normally program constants) after these have been processed to form a unique list, with each required constant only appearing once. The names of these dummy inputs are deliberately distinguished from the names of their corresponding program constants by prefixing them with the word “DUMMY_”. (The actual prefix string is not significant, but for clarity something sensible should be used.) The dummy parameters then have their values set in the box’s Init Method section, by assigning to them the program constant that they represent. These values can be guaranteed to be available since the source file must be listed in the box’s Include section. In this way, the required sizes are all known

at scheduling time, and are defined only once in the data structure definition source file.

It should be noted that this method means that the dummy parameter inputs to the box need not be connected on the canvas of the parent flowgraph. This fact should be explained by clarifying comments on the canvas, since if other values are connected it might be unclear to someone reading the graph which values will be used.

4.4 Hierarchy considerations

When processing data structures that include nested (and possibly themselves hierarchical) sub-structures, it is necessary to check that the details of the necessary sub-structures (*e.g.* its number of fields and their types) are already known. This means that the sub-structure must have already been parsed in this run, or should be available to be parsed so that the required information can be gathered immediately.

Should any other files need to be included in a primitive, to provide definitions of types or program constant values, it is better if the file that contains those definitions is in a location that is easily accessible by GEDAE. This means that the path to the include file should either be relative to the current box, or one of GEDAE's default include paths, so that no additional flags are required to enable the compilation of the box code.

5. DATA STRUCTURE SUPPORT BOXES

As identified in Section 1.1, the set of boxes produced in support of the use of application-specific data structures can be split into three categories. These are described below, along with the list of types of boxes that can be generated in each category by the current version of the automatic code generator.

5.1 Flow Control Boxes

The boxes most frequently used in support of the integration of data structures into a GEDAE model are those concerned with controlling the flow of tokens of that type. These so-called 'logic' boxes are used to route tokens into and out of particular blocks of processing, or to discard tokens that are not required under certain conditions. This set of boxes also includes those concerned with the duplication and synchronisation of tokens within the application flowgraph.

All of these boxes operate on full tokens of the datatype – *i.e.* they require no information about the content or structure of the tokens that they are processing. This means that they can be generated using the template-based approach.

The set of boxes that can currently be generated includes the following :

- branch, branchf, branchf_c
- merge, mergef, mergef_c
- hold
- valve, i_valve (this is an inverse lookup valve, not an integer variant)
- copy
- sync_s, sync_si

The number of branches to be processed can be specified for the branch and merge boxes. (The branchf, branchf_c and their corresponding mergef and mergef_c boxes do not require this additional information.)

The sync boxes perform the same function as the gate primitive provided for ints, floats and complexes as part of the standard GEDAE library, but for different synchronising token datatypes. They do not actually carry out any processing, but the static dataflow requirements of the boxes require that a token be present on both inputs before the box will fire, thus synchronising the two streams.

5.2 Data Structure Manipulation Boxes

These boxes explicitly reference the fields or elements of application-specific data structures. These are generally used to create or destroy the data structures from or into their constituent components. This allows other processing to be carried out on the individual fields or elements using boxes from the standard GEDAE library.

These boxes must be generated using the script-based generation technique, since sections of code must be written for each of the structure elements.

The set of boxes that can currently be generated includes the following :

- explode, implode
- GEDAE typedef box
- C-to-GEDAE, GEDAE-to-C, GEDAE-to-GEDAE
- data generator boxes

The explode boxes take in a stream of the specified data structure compound type and convert it to a set of output streams, one for each field or element of the original structure. The types of the output streams are determined from the data structure definition and thus may be other, possibly hierarchical, data structures or multi-dimensional arrays. Implode boxes perform the inverse operation, converting a set of streams of the appropriate types into a single stream of tokens of the appropriate data structure.

The GEDAE typedef (constructor) box is a graphical convenience that allows multiple data flows to be represented by a single line on the canvas without actually combining their data content or affecting the data flow of the individual components at all. Its converse (the destructor box) can be automatically generated by GEDAE when required.

The C-to-GEDAE, GEDAE-to-C and GEDAE-to-GEDAE boxes are flowgraphs that carry out the specified conversions. The C-to-GEDAE box consists of an explode box connected to a GEDAE typedef constructor, that converts a stream of the C data structure compound types into a set of streams of tokens represented by a GEDAE typedef. The GEDAE-to-C box consists of a GEDAE typedef destructor box connected to an implode box that performs the reverse operation. The GEDAE-to-GEDAE box is built from a GEDAE typedef destructor connected to a GEDAE typedef constructor and is provided to save the developer from having to manually make all of the other connections in situations where a small number of the fields must be explicitly modified – the automatically generated box can be used as a template, the required

processing added and the box renamed and saved. Boxes that carry out application-specific processing on one or more fields or elements are not normally generic enough to warrant automatic generation.

Boxes that explode or implode hierarchical data structures must additionally explode or implode each of the sub-structures accordingly. This requires the sub-structure to have already been processed so that their explode and implode boxes exist. It is also possible that the parent structure has a member that is an array of the sub-structures. In this case, the explode or implode boxes must be promoted to the required dimensionality – *e.g.* to make vector or matrix variants, or higher dimensions if necessary. This requires that a new primitive be automatically generated with an additional degree of dimensionality added to that existing, for each level of promotion.

5.3 Input / Output Boxes

Different boxes from the input / output category consider the tokens either in their entirety (*i.e.* as the complete structure of the compound datatype) or taking into account the different fields or elements. This means that both the template-based and script-based techniques are used to generate various boxes in this class. Binary read and write boxes are created using the template-based approach, referencing the data as a block, whereas ASCII input and output boxes such as `scanf` and `echohdr` boxes that explicitly refer to the individual components of the structure are created using the script-based generation method

The set of boxes that can currently be generated includes the following :

- `read`, `write`
- `scanf`, `echohdr`, `printf`

Note that `fscanf` and `fechohdr` or `fprintf` boxes could also be generated in this way. Processing that minimises the amount of file system interaction is generated for all of the boxes in this category.

To help test the input boxes, data generators can also be produced. These are intended to be only templates and provide simple data without any constraints on acceptable values within the fields of the structures. It is the application flowgraph developer's responsibility to modify the template code to provide meaningful data. The binary data generators are produced as C code which is then automatically compiled and run by the automatic code generator script. It should be noted that there are byte-ordering issues associated with the use of binary data, and that consequently the data generators must be recompiled for each platform on which the model is to be run.

6. TESTING

With such a large library of boxes potentially being produced, testing all of these boxes for correctness is clearly an issue. Instead, structured inspection of the automatic code generator scripts and all template box code is carried out to find any typographical or coding errors. This should identify any mistakes in the generator itself, thereby ensuring that any boxes produced will contain the expected code.

Following this, the behaviour of the generated boxes is tested. A minimum of two different data structures are processed using all of the generation options, to produce a full set of boxes – two different structures are required to ensure that there are no explicit references to a particular structure or element. It is preferable that at least one of the structures processed be complicated enough to allow all of the different sub-routines of the automatic code generation scripts to be tested. Certainly all of the sub-routines should at least be covered by the combination of the structures tested. The boxes thus generated are used to produce a flowgraph test harness, and this is then run and the output examined to confirm that the boxes are behaving as expected. A combination of configuration management and version control further allows individual boxes to be checked to ensure that any modifications to the generator scripts do not result in unexpected changes to the code produced.

If all of these tests are passed, no other generated boxes should need to be tested since they will all have been produced in the same way according to the same rules.

7. USE ON CAPTOR TRANCHE 2

The CAPTOR programme is a joint European project to develop the multi-mode nose radar for the Eurofighter Typhoon aircraft. The project is split into a number of phases :

- Tranche 1 (T1) developed a baseline radar which is now in production and being flown on a number of Eurofighter aircraft.
- Tranche 2 (T2) is a re-development of the system, primarily to support a COTS processor.

The T1 Signal Processor software is being re-developed in GEDAE at Tranche 2 to facilitate its porting to COTS hardware.

Since the signal processing itself will not change, it was decided to re-use some parts of the legacy T1 design, where possible. Keeping the existing legacy application-specific data structures simplifies the dataflow around the model and reduces the complexity of the re-development work and associated documentation.

The availability of the automatic code generator has greatly helped development of the project. It provides a simple way for developers to produce the required support boxes when new data structures are defined, or to update the current support library when existing structures are modified. This has allowed updates to the data structures to be propagated rapidly and improvements to the boxes to be incorporated with minimal impact.

Use of the automatic code generator has thus saved a considerable amount of work. There are 32 data structures being used on the Tranche 2 programme, and with all of the generation options switched on, 1173 boxes are produced by the script to support their use. The time taken to manually create these boxes would have been quite significant, even assuming no typographical mistakes were made and that templates had existed for all of the boxes. It takes only five minutes to regenerate the entire set using the automatic code generator.

This is a worthwhile saving in itself, but additional savings have been made on maintenance time as well. Some of the data structures were modified to improve on some of the original T1 legacy design decisions. These changes require the updating of all of the boxes relating to those particular structures – about thirty boxes per structure. It would have taken several hours to perform this task manually. Using the automatic code generator, it was achieved in a matter of seconds. Similar savings were made when propagating improvements to the template box code.

Had the boxes been generated or modified manually, they would have required testing or structured inspection. This would have greatly increased the cost of the task.

Examples of the types of box that can be generated are given in Figures 1–4, along with the data structure definitions from which they were generated in Figure 5.

8. FUTURE WORK

Development of the automatic code generator is on-going. As the Tranche 2 programme proceeds, new ideas arise – and will continue to do so – on how the tool can be improved or extended.

It is clear that the library of boxes produced can be easily augmented by making available template code for the desired boxes and extending the generator to process these when so directed. Should any additional information about the data structures be required in order to generate the code, it should be relatively straight-forward to extend the parsing module to extract the necessary data.

Other potential extensions to the generator itself that have been identified include :

- Automatic identification and extraction of library search keywords from the structures' source code

- Automatic generation of test harnesses for automatically generated data structure support boxes
- Parsing modules to support other source code languages
- A graphical user interface (GUI) front-end
- Direct incorporation of the generator into GEDAE

The development of these improvements will be driven by demand on the project and other desirable extensions will undoubtedly come to light as the generator is exercised still further.

9. CONCLUSIONS

Automatic code generation has been successfully used to support the integration of application-specific data structures within GEDAE models on the Captor Tranche 2 programme. It has been shown that it is possible to automatically create a library of support boxes from a set of data structure definitions. Various categories of boxes have been identified that require different techniques to be employed in their generation. Issues relating to this generation and to the testing of boxes so produced have been considered. A solution that offers a powerful and flexible means to automate the writing of the boxes of the support library has been developed.

The use of the automatic code generator on the Captor Tranche 2 programme has shown that considerable savings of time and effort can be made in the development and maintenance of such a library. Further savings may be possible, and development work on the tool is continuing.

10. ACKNOWLEDGEMENTS

The author acknowledges the support of Andrea Cook at SSD Crewe Toll who contributed to the development of this work.

```

Name: explode_trivial_struct_t
Type: static
Comment: "
    Automatically generated by gen_typedefs.pl from
    marcus/data_structures.h
"
Keyword: {"meta trivial_struct", trivial_struct_t}

Input: {
    stream trivial_struct_t c_trivial_struct;
    int DUMMY_ARRAY_SIZE;
}
Output: {
    stream int id;
    stream float field_1;
    stream float field_2;
    stream complex data_values[DUMMY_ARRAY_SIZE];
}

Include: {
    /*
     * Header file for all of the types and constants used in the structures
     */
    #include "marcus/data_structures.h"
    #include <complex.h>
}

Init: {
    /*
     * Initialise Array dimensions
     */
    DUMMY_ARRAY_SIZE = ARRAY_SIZE;
}

Apply: {
    /*
     * Copy the data from the stream typedef structure to the individual
     * stream elements, according to the box granularity and size
     *
     * Should make use of e_*fill functions where possible, but most of the
     * types present here will not have a corresponding function
     */
    int i, G = granularity;

    for (i = 0; i < G; i++) {
        memcpy(&(id[i]),
            &((c_trivial_struct[i]).id),
            sizeof(int));
        memcpy(&(field_1[i]),
            &((c_trivial_struct[i]).field_1),
            sizeof(float));
        memcpy(&(field_2[i]),
            &((c_trivial_struct[i]).field_2),
            sizeof(float));
        memcpy(&(data_values[i * (DUMMY_ARRAY_SIZE)]),
            &((c_trivial_struct[i]).data_values),
            (DUMMY_ARRAY_SIZE) * sizeof(complex));
    }
}

```

Figure 1 - Example primitive generated using the script-based generation technique

```

Name: write_hierarchical_struct_t
Type: static
Comment: "
  Automatically generated by gen_typedefs.pl from
  marcus/data_structures.h
"
Keyword: {"meta hierarchical_struct", hierarchical_struct_t}

Input: {
  char Name[N]; /* File name of the binary file */
  stream hierarchical_struct_t c_hierarchical_struct;
}

Local: {
  /* File descriptor goes here, so that it persists from run to run */
  int fd;
  hierarchical_struct_t l_hierarchical_struct;
}

Include: {
  /*
  Header file for all of the types and constants used in the structures
  */

  #include "marcus/data_structures.h"
}

Reset: {
  /*
  Make sure that the file descriptor is reset to the start of the file,
  and that the file is open
  */
  if (fd) embFClose(fd);
  fd = embFOpen(Name,"wb");
  if (!fd) OStaticFailed("Could not open file");
}

Apply: {
  /*
  Writes the data from the input stream straight to file
  */

  int i, G = granularity;

  for (i = 0; i < G; i++) {
    if (fd) { /* May no longer be valid... */
      embFWrite(fd, 1, sizeof(hierarchical_struct_t), &(c_hierarchical_struct[i]));
    } else {
      OStaticFailed("Invalid file descriptor");
    }
  }
}

```

Figure 2 – Example primitive generated using the template-based generation technique

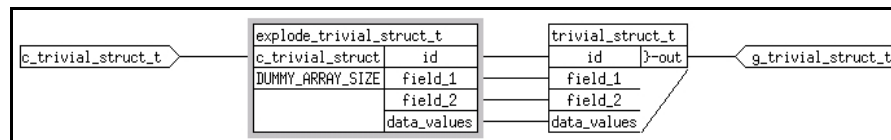


Figure 3 – Example flowgraph generated using the script-based generation technique

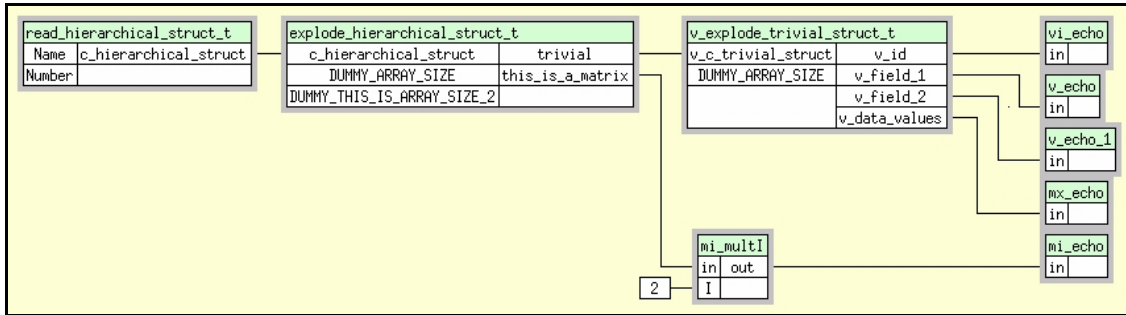


Figure 4 – Example flowgraph using automatically generated boxes and standard GEDAE primitives

```

/*****\
*   This is a header block comment   *
\*****/

#ifndef _DATA_STRUCTURES_H
#define _DATA_STRUCTURES_H

/*
   Include any other type definitions required
   by types declared in this file
*/
#include "complex.h"

/* Set up the program constants required */
#define ARRAY_SIZE 5
#define THIS_IS_ARRAY_SIZE_2 7

/*
   This is a trivial enum typedef that will
   not be processed as it is not a structure
*/
typedef enum {
    OFF_VALUE,
    ON_VALUE
} trivial_enum_t;

/* This is a trivial struct typedef */
typedef struct {
    int id;
    float field_1;
    float field_2;
    complex data_values[ARRAY_SIZE];
} trivial_struct_t;

/* This is a hierarchical struct typedef */
typedef struct {
    trivial_struct_t trivial[THIS_IS_ARRAY_SIZE_2];
    int this_is_a_matrix[ARRAY_SIZE][(THIS_IS_ARRAY_SIZE_2 - ARRAY_SIZE) * 2];
} hierarchical_struct_t;

/* This is another hierarchical struct typedef */
typedef struct {
    hierarchical_struct_t hierarchical;
} another_hierarchical_struct_t;

#endif /* _DATA_STRUCTURES_H */

```

Figure 5 – Source data structure definition file