PRECOMPILED SUBMODELS: A GENERAL SORTING PROCEDURE

Guasch A., Huber R.
Institut de Cibernètica (UPC/CSIC)
Barcelona, Spain

ABSTRACT

Simulation languages supporting hierarchical submodel structures, face sorting problems not present in monolithic CSSL7 models. These problems, result from algebraic relationships between submodels leading to information loops (1) in the model code.

This paper proposes a global solution, based on the segmentation of the submodels through graph analysis, and its promising application to sorting problems.

INTRODUCTION

The separate compilation of declarative submodels is still not widely accepted to be included in a desirable new Continuous System Simulation Language (CSSL) specification (TC3-IHMS Newsletter No. 12).

The main point against its inclusion is the problem of sorting the code relatively to other submodels. Using classical methods, information loops (those that disappear if the called lower level submodels are handled like macros) appear in the calling submodel.

The authors contribution is an attempt to solve the sorting problem by means of a segmentation procedure. The report, first introduces elementary graph concepts: second, the submodel digraph is defined; third, a general segmentation procedure is described; fourth, a classical sorting procedure is introduced to sort the sentences within each segment; and finally, a few related points are analyzed: side effects, submodel object code, and an example.

ELEMENTARY GRAPH CONCEPTS

The CSSL sorting problem can be solved through directed graph (digraph) sorting procedures. A digraph is a collection of vertices \( V = \{v_1, v_2, \ldots \} \), a collection of edges \( E = \{e_1, e_2, \ldots \} \) and a mapping \( \psi \) that maps every edge onto some ordered pair of vertices \( \{v_i, v_j\} \). Vertices are simple objects and an edge is a connection between \( v_i \) and \( v_j \) with an arrow directed from \( v_i \) to \( v_j \).

A path from vertex \( v_i \) to \( v_j \) in a graph is a list of vertices in which successive vertices are connected by edges in the graph. A path is directed -directed path- if the edges have the same orientation, otherwise, it is a semipath. A graph is connected if there is a path from every vertex to every other vertex in the graph. A graph which is not connected is made up of connected components. A connected graph is strongly connected if there is at least one directed path from every vertex to other vertex, otherwise, it is weakly connected. A simple directed path is a directed path in which no vertex is repeated and a directed cycle is a directed path which is simple except that the first and last vertex are the same.

A vertex \( v_j \) is reachable from a vertex \( v_i \) if there is a directed path from \( v_i \) to \( v_j \).

Graphs with all edges present are called complete graphs; graphs with relatively few edges are called sparse; graphs with relatively few of the possible edges missing are called dense.

In order to be segmented and sorted, graphs have to be directed and acyclic. These graphs are called directed acyclic graphs (dags).

In our CSSL problem the sentences are the vertices and the edges are the relationships between sentences due to its input and output variables - explicit edges. The implicit relationships between segments constitute the implicit edges or hidden edges which will be defined later.

The following code (*), where \( k_1, k_2 \) and \( k_3 \) are constants, \( y \) is a state variable (**), and \( a, x, y, z, r, q \) and \( z \) are real variables,

\[
\text{Sentence i} \quad (V_i) \quad (a := \text{Fun}(x,y));
(V_i+1) \quad y' := a^k_3
(V_i+2) \quad x := \ldots
(V_i+3) \quad z := r^k_1
(V_i+4) \quad t, q := \text{Sub}(r, k_2)
(V_i+5) \quad v := z^k t^k q
(V_i+6) \quad r := \text{Time}
\]

will be represented by the following digraph:

\[
\begin{align*}
V_i+2 & \rightarrow V_i+6 \\
V_i & \rightarrow V_i+3 \\
V_i+1 & \rightarrow V_i+5 \\
V_i + & \leftarrow t \\
V_i & \rightarrow x \\
V_i & \leftarrow a \\
\end{align*}
\]

*) The syntax used for coding examples is informal.

**) Integration output variables.
The state variables of the system do not appear in this graph because their values are known in advance.

The graphs associated with CSSL models (CSSL-graphs) are:

- Sparse: Graph with relatively few edges (< \(V \log V\)).
- Directed: The edges are one-way.
- Acyclic: Algebraic implicit loops leading to directed cycles may appear in incorrect models. They have to be eliminated beforehand.

- In general, they are not connected.

**SUBMODEL DIGRAPH**

Let be a submodel coded in a CSSL language:

```plaintext
* v1  v2 */
SUBMODEL example(in:REAL f; out:REAL cy) IS
CONSTANT
  REAL a,b,c;
END CONSTANT;
STATE v,v; REAL av,by;
DYNAMIC
  /*v3*/ s := f*a+v+by;
  /*v4*/ av := a-s*v;
  /*v5*/ by := b*y;
  /*v6*/ y' := s;
  /*v7*/ cy := c*y;
END DYNAMIC;
END example;
```

Its associated digraph \(G_m=(V_m,E_m)\) will be:

```
\[\begin{array}{c}
v_1-
\end{array}\]
```

where:

- \(V_m = (I_m,O_m,I_m)\) is the set of vertices, \(V_m = \{v_1,v_2, v_3, v_4, v_5, v_6, v_7, v_8\}\)

being:

- \(I_m\): set of input vertices. Each formal parameter in the input list becomes an output vertex.
  \(I_m = \{v_1\}\)

- \(O_m\): set of output vertices. Each formal parameter in the output list becomes an output vertex.
  \(O_m = \{v_2\}\)

- \(E_m\): set of executable vertices
  \(E_m = \{v_3,v_4,v_5,v_6,v_7,v_8\}\)

- \(E_m\) is the set of directed edges.

- \(E_m = \{(v_1,v_3), (v_4,v_3), (v_5,v_3), \ldots\}\)

Given a vertex \(v_i\) belonging to \(V_m\), \(R(v_i)\) is the set of vertices reachable from the vertex \(v_i\), i.e.

```
R(v_1) = \{v_1,v_3,v_6\}
```

the reaching set \(Q(v_i)\) is the set of vertices from which vertex \(v_i\) can be reached
i.e.

```
Q(v_6) = \{v_6,v_1,v_4,v_5,v_3\}
```

\(p(v_i)\) is the set of predecessors of \(v_i\)

i.e.

```
p(v_6) = \{v_3\}
```

and \(s(v_i)\) is the set of successors from \(v_i\)

i.e.

```
s(v_3) = \{v_6\}
```

**SEGMENTATION**

Some Definitions

- The output weight of a vertex \(v_i\) of the submodel digraph is the number of output vertices reachable from vertex \(v_i\).
  \(Ow(v_i) = |R(v_i) \cap O_m|\)

- The input weight of a vertex \(v_i\) of the submodel digraph is the number of input vertices which can reach \(v_i\).
  \(Iw(v_i) = |Q(v_i) \cap I_m|\)

- A pair of vertices \(v_i\) and \(v_j\) are said to be fused, if both are replaced by a single new vertex such that every vertex that is incident on \(v_i\) or \(v_j\) or both is now incident on the new vertex \(2\).

Once the executable vertices have been clustered (fused) using the rules explained later in order to get a reduced digraph \((G_r)\), a segment \(S=(V,E)\) will be a subdigraph of the reduced digraph.

According to the nature of its vertices, three types of segments are defined:

- State segment \((S=(V_6,E_6))\): Its executable vertices only depend on state variables or constants.
Fusion Rules

The following fusion steps are proposed to get the reduced digraph from the submodel digraph.

- Step 1:
  Fuse the weakly connected executable vertices that have the same output weight.

- Step 2:
  After completing the previous step, fuse the weakly connected executable vertices that have the same input weight.

- Step 3:
  Fuse the executable vertices having zero input weight.
  Fuse the executable vertices having zero output weight.

The executable vertices having input and output weights equal to zero are fused with any executable vertex.

Segments

Each executable vertex \( t_i \) of the reduced digraph has an associated segment, its vertices being,

\[
T_i = t_i
\]

\[
I_i = p(t_i) \land I_m
\]

\[
O_i = s(t_i) \land O_m
\]

If \( t_s \) is an executable vertex of the reduced digraph with input weight equal to zero, it has an associated state segment \( S_s = \{V_s, E_s\} \), its vertices being,

\[
T_s = t_s
\]

\[
I_s = p(t_s) \land I_m \quad \text{(empty)}
\]

\[
O_s = s(t_s) \land O_m
\]

If \( t_d \) is an executable vertex of the reduced digraph with output weight equal to zero, it has an associated derivative segment \( S_d = \{V_d, E_d\} \), its vertices being,

\[
T_d = t_d
\]

\[
I_d = p(t_d) \land I_m
\]

\[
O_d = s(t_d) \land O_m \quad \text{(empty)}
\]

All the other executable vertices \( t_{ai} \) have associated algebraic segments because they belong, at least, to a directed path from an input vertex to an output vertex.

Example

Let be the submodel digraph shown in the figure:

![Diagram](image)

Each vertex \( t_{ai} \) has an associated algebraic segment \( S_{ai} = \{V_{ai}, E_{ai}\} \), its vertices being,

\[
T_{ai} = t_{ai}
\]

\[
I_{ai} = p(t_{ai}) \land I_m
\]

\[
O_{ai} = s(t_{ai}) \land O_m
\]

where

\[
I_m = \{1, 2, 3, 4, 5\}
\]

\[
O_m = \{6, 7, 23, 28, 26\}
\]

\[
I_m = \{8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 27, 29, 30\}
\]

Fusion: The fusion steps are applied to cluster executable vertices with equivalent sorting properties:

- Step 1: Fuse weakly connected executable vertices that have the same output weight.
Step 2: Fuse weakly connected executable vertices that have the same input weight.

\[ 1-1 \quad 5-1 \quad 2-1 \quad 3-1 \quad 4-1 \]
\[ \{16,17,18,19,12,21,22\} \]
\[ (8,10,15,20) \quad 23-o \quad 6-o \]
\[ (13,14,11) \]

Step 3: Fuse the executable vertices having zero input weight and fuse the executable vertices with zero output weight, even if they are not weakly connected.

The reduced digraph is:

\[ 1-1 \quad 5-1 \quad 2-1 \quad 3-1 \quad 4-1 \]
\[ \{16,17,18,19,12,21,22\} \]
\[ (8,10,15,20) \quad 23-o \quad 6-o \]
\[ (13,14,11) \]

Resulting Segments: The segments will be:

- Vertices of the state segment
  \( V_s = \{1a, 0s, 1s\} \)
  \( T_0 = \{24, 25, 27, 29, 30\} \)
  \( I_0 = p(T_0) \) Im = Empty
  \( O_0 = s(T_0) \) Om = \( \{28, 26\} \)

- Vertices of the derivative segment
  \( V_d = \{1d, 0d, Td\} \)
  \( Td = \{9, 13, 14, 11\} \)
  \( Id = p(Td) \) Im = \( \{1, 4\} \)
  \( Od = s(Td) \) Om = Empty

- Vertices of the algebraic segment
  \( V_a = \{1a_l, 0a_l, T_a\} \)
  \( T_a = \{8, 10, 15, 20\} \)
  \( I_a = p(T_a) \) Im = \{3\}
  \( O_a = s(T_a) \) Om = \{23, 6\}

- Vertices of the algebraic segment
  \( V_a = \{1a_2, 0a_2, T_a_2\} \)
  \( T_a_2 = \{16, 17, 18, 19, 12, 21, 22\} \)
  \( I_a_2 = p(T_a_2) \) Im = \( \{1, 5, 2\} \)
  \( O_a_2 = s(T_a_2) \) Om = \{7\}

A segment digraph is defined to show the sorting relationship between segments.

\[ S_5 \quad S_1 \quad S_2 \quad S_d \]

The call to the previous submodel from a higher level submodel will be split into four calls, one for each segment.

The edges of the segment digraph are called hidden edges because they can not be inferred from the higher level submodel through the input-output variables in the segment calls.

Furthermore, to properly sort both submodels (high level submodel and low level submodel) as a whole, the knowledge of the segment digraph is required. This knowledge about submodels interfacing can be stored in a Submodel Information Library (SIL).

Submodel Blocks: A Critique From the Segmentation Point of View

Although combined system simulation languages can be used to describe systems composed by continuous and discrete subsystems, at present we will restrict ourselves to piecewise continuous system simulation languages.

A first criticism of the blocks used in CSSL67 languages from the segmentation point of view is shown in the next points. The authors feeling is that a deeper analysis is necessary before attempting the definition of the blocks in a new CSSL specification if the sorting of precompiled submodels is wanted.

Initial Region

"The Initial region — subregion— is procedural code that performs initialization calculations, input..." (3).

If we accept the above CSSL67 definition the initial region has to be managed as a single vertex — initial vertex—, nevertheless, information loops may appear if the INPUT vertices to the initial vertex are not constants.

If the initial region is to be preserved, some modifications or restrictions have to be introduced in its definition in order to avoid information loops.

At first glance, two approaches are possible:

1. To preserve the initial region definition but restrict the INPUT vertices to it to constants.
2. To define the initial region as non-procedural code. The sentences of the
initial region will be placed in different segments according to the stated segmentation steps.

The proposed segmentation method allows the initialization of models in a single pass.

Terminal Region

Computations done within the terminal region are always part of the experiment.

Besides, the existence of a terminal region in a submodel increases the difficulty of segmentation.

The terminal region is likely to disappear in a model or submodel structure definition.

Procedural Pseudoblock

The procedural pseudoblock has to be viewed as a single vertex in the submodel digraph. This vertex will be connected to the other vertices through edges defined by the formal parameters of the input/output procedural list.

MACRO Pseudoblock

The MACRO facility is not needed anymore since the proposed segmentation method avoids the necessity of spreading the submodel sentences throughout the calling submodel in order to sort it (4) (5).

NOSORT Pseudoblock

The NOSORT pseudoblock is a potential source of errors for non-expert users.

This type of procedural block is not compatible with the proposed segmentation strategy.

Sample Block

The authors conceive the SAMPLE block as a submodel to be called within the dynamic region of a higher level submodel (6) rather than a block that shares the dynamic region with the derivative segments (7) (8).

In that case, calling a SAMPLE submodel will be viewed as a single vertex.

SUBMODEL SORTING

Detection of directed cycles (algebraic implicit loops in the model) and sorting is done before the segmentation of a submodel digraph.

The following steps are taken to sort a submodel:

1. Replace the calls to lower-level submodels by calls to its segments.
2. Create the submodel digraph of the submodel being sorted.

3. Apply the modified depth-first search algorithm (9) to detect directed cycles and sort the digraph.

At the end of the segmentation which will follow, the sentences within each segment have to maintain the precedence in the sorted submodel digraph.

SIDE EFFECTS

If a submodel is to be called more than once within a model the generated code has to be reentrant. In this case, the consideration below has to be taken into account.

The code in the segments of a submodel holds regular sentences and segment calls to lower level submodels.

Therefore, in general, the hidden edges in the segment digraph associated to the higher level submodel can be divided into two groups:

- Those coming from the regular sentences.
- Those coming from hidden edges in the segment digraph of the lower level submodels.

Dynamic memory is needed only for the variables associated to the edges coming from the regular sentences in order to get the submodel reentrancy.

SUBMODEL OBJECT CODE

FORTRAN 77 has been selected as the preprocessor target language looking for the compatibility with the FORTRAN 77 CSS167 (3) simulation language developed at the Institut de Cibernètica.

Two ways of coding a submodel are possible:

- To use a subprogram with different entry points to the segments.
- To use a subprogram for each segment with a local COMMON in order to share the variables with global scope within the submodel.

The first solution seems to be neater and easier to implement.

EXAMPLE

The example proposed by Prof. Rinvall (10) is very suitable to support the outlined segmentation procedure.

It consists of two coupled masses moving on a frictionless wall (fig. 1). The left mass is connected with a spring and a damping to a wall and the second mass is externally forced.
the equations for the system are:

\[ m_1 x_1'' + k_1 x_1 + c_1 x_1' - k_2 (x_2 - x_1) - c_2 (x_2' - x_1') = 0 \]

\[ m_2 x_2'' + k_2 (x_2 - x_1) + c_2 (x_2' - x_1') = \sin(\omega t) \]

and the code in a CSSL language may be:

\[
\begin{align*}
v_1' &= (f_1 + f_{rl}) / M_1; \\
x_1' &= v_1; \\
f_{rl} &= -K_1 * x_1 - C_1 * v_1; \\
f_1 &= f_{rl}; \\
v_2' &= (f_{rl} + f_{r2}) / M_2; \\
x_2' &= v_2; \\
f_{r2} &= -K_2 * x_2 - C_2 (v_2' - v_1'); \\
f_{r2} &= \sin(\Omega * M_1, \text{PHASE});
\end{align*}
\]

The above mass system submodel can be extended to increase its generality (fig. 2).

![Fig. 2: Generalized mass subsystem](image)

The code associated to the generalized mass subsystem is:

```plaintext
/* ------------------------ mass_system */
/ * v0, v1, v2, v3 */
SUBMODEL mass_system(in:REAL x0, v0, f12, f1; 
/ * v4, v5 */
/ out:STATE x1, v1; REAL f13;)IS
REAL fr1;
CONSTANT
REAL M1, K1, C1;
END CONSTANT;
DYNAMIC
/ * v0 */
/ v1' := (f1 + f_{rl}) / M_1;
/ v8 */
/ f_{rl} := f_{rl};
/ * v5 */
/ f_{rl} := -K_1 * (x_1 - x_0) - C_1 * (v_1 - v_0);
/ * v10 */
/ x_1' := v_1;
END DYNAMIC;
END mass_system;
```

and the coupled masses system model will become:

```plaintext
/* -------------------- coupled_mass_system */
SUBMODEL coupled_mass_system () IS
CONSTANT
REAL x0 := 0.0, v0 := 0.0, f1 := 0.0, 
FL3 := 0.0, OMEGA := 1.0,
PHASE := 0.0;
END CONSTANT;
SUBMODEL
mass_system.left WITH CONSTANT
REAL M1 := 10.0, C1 := 10.0, 
K1 := 25.0;
END WITH;
mass_system.right WITH CONSTANT
REAL M1 := 8.0, C1 := 6.0, 
K1 := 35.0;
END WITH;
END SUBMODEL;
DYNAMIC
f2 := \sin(\Omega * \text{PHASE});
x_1, v1, f_{rl} := 
mass_system.left(x0, v0, f_{rl}, f1);
x_2, v2, f_{r2} := 
mass_system.right(x1, v1, FL3, f2);
END DYNAMIC;
END coupled_mass_system;
```

The call to the generalized mass subsystem submodel has a qualifier field that is used to initialize the constants. The constants of a submodel can be initialized within the submodel or from higher level submodels. The highest level initialization of a given constant overrides lower level initializations of the same constant.

The digraph associated to the generalized mass subsystem submodel is:

![Digraph](image)

and the segments are:

1. State segment Ss=(Is, Os, Ts)
   - Is = Empty
   - Os = (v4, v5)
   - Ts = Empty
2. Derivative segment Sd=(Id, Od, Id)
   - Id = (v2, v3)
   - Od = Empty
   - Id = (v8, v7, v10)
3. Algebraic segment $Sa = (Ia, Oa, Ta)$

\[ Ia = \{ v0, v1 \} \]
\[ Oa = \{ v6 \} \]
\[ Ta = \{ v9 \} \]

Writing the calls to the preceding segments using the CSSL syntax, the shape of the dynamic region becomes:

```
DYNAMIC
b1 f2 := sine(OMEGA, PHASE);
```
```
b2a x1, v1 := mass_system_state_left();
b2d mass_system_derivative_left(f12, f3);
b2a f11 := mass_system_algebraic_left(x0, v0);
b3s x2, v2 := mass_system_state_right();
b3d mass_system_derivative_right(f13, f2);
b3a f12 := mass_system_algebraic_right(x1, v1);
END DYNAMIC;
```

The associated digraph is:
```
  b1 ---- b3s ---- b2s
    \       /       /
     \     /       /
      \   /       /
         \ /       /
          b3d ---- b3a ---- b2a
               \
                b2d
```

Edges represented as continuous arrows are inferred from the input/output variables of each sentence (explicit edges) and dotted edges arise from the segment digraph (hidden edges).

Since the coupled mass system submodel does not have input/output variables, all the sentences within the submodel will be placed in the state segment.

**CONCLUSIONS**

The outlined segmentation procedure has been successfully applied to sort a complex electrical network model. The model has 17 submodels and the hierarchical structure has four levels.

Nowadays, a CSSL language with segmentation capability is under development.

**REFERENCES**

2. Deo N., Graph Theory With Applications to Engineering and Computer Science, Prentice-Hall, 1974.