CHILD Program Description

Version 2.0

Greg Tucker[1]

Department of Civil and Environmental Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139

Part IV-B of final technical report submitted to U.S. Army Corps of
Engineers Construction Engineering Research Laboratory (USACERL)
by Gregory E. Tucker, Nicole M. Gasparini, Rafael L. Bras, and Stephen T. Lancaster
in fulfillment of contract number DACA88-95-C-0017

April, 1999

1. To whom correspondence should be addressed: Dept. of Civil & Environmental Engineering, MIT Room 48-429, Cambridge, MA 02139, ph. (617) 252-1607, fax (617) 253-7475, email gtucker@mit.edu

CHILD Program Description

Version 2.0

Design Philosophy

The CHILD software has been designed to maximize modularity and flexibility. For that reason, the C++ code is divided among a group of files and classes, many of which are general utilities that are independent of the model itself and may be reused in other applications. The mesh-handling routines are designed to function independently of the process routines, so that in principal the mesh-handling routines can be used by other applications unrelated to CHILD (note that with the current release, some explicit references to CHILD's meandering and "layering" capabilities still exist within the mesh routines; further development is needed to fully realize the potential for completely independent mesh capabilities). Basic parameters and functionality related to each process are in most cases grouped into a single "process class" or group of related classes; for example, overbank deposition is managed using the tFloodplain class, which is wholly contained within the files tFloodplain.h and tFloodplain.cpp. The various functional capabilities of the model and the associated classes and files are summarized in Table 1.

The tMesh class manages the simulation mesh and it includes functions for creating, reading, and updating the mesh, as well as routines for managing dynamic point addition, deletion, and movement. Class tMesh contains linked lists of each of the three mesh elements, which are encapsulated in the classes tNode, tEdge, and tTriangle. Decoupling of these mesh-management capabilities from the various process routines is accomplished in part by the use of templates. tMesh is a templated class, with the template variable being the data type that represents a node on the mesh. In principal, this design makes it possible to create a mesh using any user-specified

node type as long as the user-defined node class is a descendent of the base class tNode (again, with the current release this decoupling is not complete; some “landscape-related” functionality is still required within the inherited node class). The base tNode class simply represents a node in a Delaunay triangulation, and does not include any data or methods related to landscape processes. Instead, landscape data for individual nodes (data such as discharge, flow direction, erodibility, etc.) are contained within the inherited class tLNode (see summary in Table 1). The base class tNode provides several virtual functions that are overridden by tLNode to handle cases such as node initialization following mesh insertion and the addition or removal of a “spoke.”

Output routines follow a similar design. The tOutput class manages output of triangulation data for a generic mesh. It provides a virtual function, WriteNodeData, that can be overridden by any descendent class to perform additional output specific to a given application. In CHILD, the inherited tLOutput class manages output of various hydrologic and geomorphic quantities.

Typically, a given process or set of processes requires both “global” parameters and functions —e.g., physical constants that are uniform in space and time and routines that compute process rates across the whole domain — as well as properties that vary in space (such as runoff) and must therefore be included at the node level. In most cases, this is handled by encapsulating global data and methods within a master process class, and adding node-specific variables as needed to the tLNode class (or the classes embedded within it; see Table 1). In this regard, CHILD’s erosion, transport, and deposition routines deserve special mention. The tErosion class contains basic parameters and functions related to hillslope and channel erosion. This class also includes a numerical solution scheme for stream detachment and transport. The solution scheme is not specifically tied to any given formula. Ideally, therefore, one should be able to modify or replace the

underlying detachment and/or transport calculations without needing to recreate the overall solution scheme. To make this possible, the tLNode class contains two embedded objects, “detach” and “sedTrans,” that perform the detachment and transport capacity calculations, respectively, at a specific point on the landscape. The classes from which these objects are derived may be changed at compile-time (in the present version, for example, one can alternately use the tSedTransPwrLaw or tSedTransWilcock classes to compute sediment transport capacity at a given node). While it would be convenient to have the transport-function option available at run time instead of compile time, the need for a branch operation at every sub-iteration for every node would impose a significant performance burden.

The CHILD code also includes several generic utility classes, including a class that handles input files (tInputFiles), a class that keeps track of simulation time (tRunTimer), and a collection of container classes (tArray, tMatrix, tList, tPtrList, and tMeshList). The linked-list classes in particular are extensively used throughout the code. The tMeshList class is used to store mesh nodes and directed edges. It is inherited from tList, and it divides a list into two parts: an “active” portion containing nodes and edges in the interior of the mesh, and a “boundary” portion containing boundary nodes or “no flow” edges (edges connected to at least one closed boundary node). Data on the linked lists are accessed using a separate “iterator” class that can move up or down along a list to retrieving data items. Because of the extensive use of iterators, the syntax of loops in the CHILD code may look unusual to those accustomed to more traditional programming style. A typical example of loop syntax is:

```
tMeshList<tLNode> nodeList = meshPtr->getNodeList();
tMeshListIter<tLNode> nodIter( nodeList );
tLNode *cn;
```

```
for(cn=nodIter.FirstP();nodIter.IsActive();cn=nodIter.NextP())
{ ... }
```

In this example, the loop sweeps through the “active” portion of the list of nodes from top to bottom. At each step, the position of the iterator on the list is updated through the call to nodIter.NextP(), which returns a pointer to the mesh node that is stored at the current position on the list.

Table 1: Summary of CHILD classes and files organized by functional category

FUNCTION / CATEGORY	CLASS(ES)	FILE(S)	NOTES
Initialization and main loop	-	main.cpp	
Mesh handling	tMesh	tMesh.h/.cpp	*-> master mesh class*
	tNode	meshElements.h/.cpp	
	tEdge	“	
	tTriangle	“	
	tListInputData	tListInputData.h/.cpp	*-> reads mesh input files*
	-	globalFns.h/.cpp	*-> geometry functions*
Landscape-specific node data and methods	tNode : tLNode	tLNode.h/.cpp	*-> represents one node*
	tLayer	“	*-> represents one layer*
	tChannel	“	*-> embedded in tLNode*
	tMeander	“	“
	tSurface	“	“
	tRegolith	“	“
	tBedrock	“	“
Output	tOutput	tOutput.h/.cpp	*-> generic mesh output*
	tOutput : tLOutput	“	*-> CHILD-specific output*
Storms and rainfall	tStorm	tStorm.h/.cpp	
Runoff, flow routing, and hydraulic geometry	tStreamNet	tStreamNet.h/.cpp	*(node-specific data stored*
	tInlet	“	*in tLNode and tChannel)*
Channel and hillslope erosion, transport, and deposition	tErosion	erosion.h/.cpp	*-> master erosion class*
	tBedErodePwrLaw	“	*-> bedrock erosion class*
	tSedTransPwrLaw	“	*-> sed transport class*
	tSedTransWilcock	“	*-> alternate transpt. class*
	tEquilibCheck	“	*-> not presently used*
Stream meandering	tStreamMeander	tStreamMeander.h/.cpp	*-> handles mesh interface*
	tReach	tReach.h/.cpp	*-> constructs 1D reaches*
	-	meander.f	*-> 1D model (FORTRAN)*

Table 1: Summary of CHILD classes and files organized by functional category

FUNCTION / CATEGORY	CLASS(ES)	FILE(S)	NOTES
Layer handling: intialization, updating, interpolation, age and exposure age tracking	tLayer tNode : tLNode tMesh	tLNode.h/.cpp " tMesh.h/.cpp	*-> one layer* *-> tLNode has layer fns* *-> layer input done in tMesh*
Floodplain overbank deposition	tFloodplain	tFloodplain.h/.cpp	
Eolian deposition	tEolian	tEolian.h/.cpp	
Tectonics and baselevel change	tUplift	tUplift.h/.cpp	
General simulation utilities	tInputFile tRunTimer	tInputFile.h/.cpp tRunTimer.h/.cpp	*-> manages main input file* *-> tracks/updates run time*
Container utilities	tArray tMatrix tList tListNode tListIter tPtrList tPtrListNode tPtrListIter tList : tMeshList tListIter:tMeshListIter	tArray.h/.cpp tMatrix.h/.cpp tList.h/.cpp " " tPtrList.h/.cpp " " tMeshList.h/.cpp "	*-> 1D arrays* *-> 2D arrays* *-> linked lists* *-> element on list* *-> list iterator* *-> list of pointers* *-> list of mesh elems*
Miscellaneous math functions	- - Predicates	mathUtil.h/.cpp globalFns.h/.cpp predicates.h/.cpp	*-> includes rand # gen* *-> public domain code by J.R.Shewchuk, Carnegie Mellon Univ.*
Error handling	-	errors.h/.cpp	
General header data: global inclusions, declarations, and definitions	-	Classes.h Definitions.h Inclusions.h	

Chain of Events in Response to Node Movement

CHILD's dynamic remeshing capabilities are rather complex. To illustrate how dynamic remeshing works, this section outlines the sequence of function calls in response to addition of a single node to the mesh.

The calling function is assumed to “pre-move” the nodes by setting their (newx, newy) values. It then calls MoveNodes, which initiates the sequence of function calls shown below. Here “g” denotes a global function, <> denotes a virtual function that may be overridden, and () denotes a conditional call (i.e., one which only happens if a certain condition is met).

```
CheckTriEdgeIntersect
   NewTriCCW (g)
   (Intersect (g))
   (InNewTri (g))
   (FlipEdge)
      DeleteEdge
         ExtricateEdge
            DeleteTriangle
               ExtricateTriangle
            <tSubNode>::WarnSpokeLeaving
      AddEdgeAndMakeTriangle
         AddEdge
            (<tSubNode>::AttachFirstSpoke)
            tEdge::WelcomeCCWNeighbor
         MakeTriangle
      MakeTriangle
   (LocateTriangle)
   (DeleteNode)
      ExtricateNode
         DeleteEdge
            ExtricateEdge
               DeleteTriangle
                  ExtricateTriangle
               <tSubNode>::WarnSpokeLeaving
      (RepairMesh)
         Next3Delaunay (g)
         AddEdgeAndMakeTriangle
            AddEdge
               <tSubNode>::AttachFirstSpoke
               tEdge::WelcomeCCWNeighbor
            MakeTriangle
         MakeTriangle
      (UpdateMesh)
         MakeCCWEdges
         setVoronoiVertices
         CalcVoronoiEdgeLengths
   tNode::UpdateCoords
```

```
   (AddNode)
CheckLocallyDelaunay
   CheckForFlip
      TriPasses
      (FlipEdge)
         DeleteEdge
            ExtricateEdge
               DeleteTriangle
                  ExtricateTriangle
               <tSubNode>::WarnSpokeLeaving
         AddEdgeAndMakeTriangle
            AddEdge
               (<tSubNode>::AttachFirstSpoke)
               tEdge::WelcomeCCWNeighbor
            MakeTriangle
         MakeTriangle
UpdateMesh
   MakeCCWEdges
   setVoronoiVertices
   CalcVoronoiEdgeLengths
```