DagTool: Tutorial

1 Introduction

In order to reduce the complexity of program development, a coordination language called Dagger has been developed on top of Charm system. In Charm, an entry point is executed when there is a message directed to it. If the computation in that entry point is dependent on the computation in another entry point within the same chare, then the programmer must handle this unexpected message by buffering it, and fetching it whenever the entry point becomes eligible for execution. Dagger hides these details from the programmer by providing ready/expect and when-block constructs. The partial orders specified in Dagger lend themselves naturally to a pictorial depiction. The intuitiveness of the pictures we drew while explaining Dagger programs led us to develop a visual programming language based on Dagger. Programmers can draw graphs, and fill in the text of when-blocks, and message specifications, and finally compile the picture into a Charmprogram. This visual programming tool is called DagTool.

Figure 1: **This figure shows the top-level menu.**

A DagTool module consists of many *components*, where a component can be either a *chare picture*, *boc picture*, *chare*, *functions*, or *declarations*. A chare or a boc picture is the graphical picture of a dag-chare (for more details read the Charm manual or Dagger tutorial) that one can draw using DagTool. For completeness, DagTool allows one to write code for chares, branch office chares, functions, and declarations inside DagTool. Each individual component is created, edited, and saved separately. The user can create a Charm (for more details, read the Charm manual or tutorial) file by using the *Translate* button in the File menu at the top-level 1. They can edit existing components, or create new ones by using the *Edit* menu at the top-level 1.

2 An illustrative example

A sample program drawn in DagTool appears in Figure 2. In a DagTool program, node symbols have the following meanings:

Figure 2: **This figure shows an example dag.**

1. Rectangles represent computations that execute without interprocessor communication.

2. Circles indicate points at which messages are received.

3. Double (concentric) circles represent ANY and ALL collections of messages. If an outgoing arc begins at the inner circle it is specifying the ANY condition–*any* one of the multiple messages expected at that node satifies the node and execution will continue on the edge leaving the node. If the outgoing arc begins at the outer circle, that indicates the ALL condition–*all* of the multiple messages must be received in order for execution to continue from this node.

4. Diamonds show branch tests. Execution following such a node depends on the outcome of the test.

5. Triangles represent control flow junctions at which control can be reached from more than one source.

Consequently, arcs between nodes have particular meanings depending on the type of nodes joined by an arc:

1. An arc between two rectangles indicates the satisfaction of a condition variable in the source, triggering the computation in the sink.

2. An arc from a rectangle to a circle indicates the expectation of a message at the circle node.

3. An arc from a circle to a rectangle shows the arrival of a message triggering a computation.

4. An arc leaving an ANY or ALL circle triggers the following computation according to the semantics of ANY and ALL as discussed above.

5. An arc emanating from a diamond is a branch depending on the truth or falsity of the branch condition.

In the program in Figure 2 therefore, there is an initial computation in the leftmost rectangle labelled *init.* Control flows from that rectangle to the next rectangle to the right, *comp1* which can also be reached from a previous iteration of the loop which is tested in the diamond at the far right in the program (*test*).

After *comp1* executes, continuation depends on the arrival of two different kinds of messages. The first, represented by the concentric circles in node *e1*, shows that multiple messages will arrive at this point. Node *comp3* will be triggered upon the arrival of any message at *e1*. Node *comp4* is not triggered until all of the messages expected at *e1* have arrived.

The second message type, at node *e2*, triggers the computation in rectangle *comp2* which must also complete before rectangle *comp4* begins. Note that only one message at *e2* is expected. After rectangle *comp4* executes, one more message is expected at node *e3* after which a test is made, which if successful, causes control to loop back to the junction node at the left (*loop*). If the branch test fails, the rightmost rectangle, *end*, executes to close the program.

3 Description of toolbar

Figure 3: **This figure shows the tool bar for DagTool.**

This section describes the functions of the buttons in the DagTool toolbar shown in Figure 3. The buttons are listed in column major order :

- Circle : this button is for drawing circles, which represent entry points at which messages are received. After selecting this button, click on the picture at the point where the centre of the circle is to be located. A query box will now pop up, where the name of the entry point and the type of message have to be entered.
- Double Circle : this button is for drawing ANY or ALL conditions. The inner circle is for an Any condition, and the outer is for the All condition. Again, click on the picture at the point where the centre of the concentric circles is to be located. Then fill in the label of the condition, the message type and the number of messages to be expected.
- Diamonds : these buttons are for drawing if-then-else type of branch conditions. The “T” represents the vertex where the flow of control branches when the condition is true.
- OR gate : this button is for drawing an OR condition.
- rectangle : this button is for drawing rectangles or code blocks.
- ‘T’ : this button is for inserting code in an or-block, condition-block or a code-block. Click inside the object for one of the above. This pops up a vi or emacs window, using which code can be entered into the block.
- Arrow : this button is for drawing arrows, which expresses a dependence. Click on the starting and ending objects, and any points in between to guide the arrow (if it needs to bend).
- True Arrow : this button is for drawing arrows which start from the True end of a diamond (branch) object.
- False Arrow : this button is for drawing arrows which start from the False end of a diamond (branch) object.
- All Arrow : this button is for drawing arrows which start from the All (outer) circle of a any-all object.
- Any Arrow : this button is for drawing arrows which start from the Any (inner) circle of a any-all object.
- Delete : this button is for deleting objects
- Edit : this button is for editing attributes of objects.