Automatic Visualization of Software Requirements:
Reactive Systems

R. Castelló
University of Texas at
Dallas
Castello@utdallas.edu

R. Mili*
University of Texas at
Dallas
rmili@utdallas.edu

I. G. Tollis
University of Texas at
Dallas
tollis@utdallas.edu

V. Winter
Sandia National Laboratories
tollis@sandia.gov

Abstract

In this paper we present an approach that facilitates the validation of high consequence system requirements. This approach consists of automatically generating a graphical representation from an informal document. Our choice of a graphical notation is statecharts. We proceed in two steps: we first extract a hierarchical decomposition tree from a textual description, then we draw a graph that models the statechart in a hierarchical fashion. The resulting drawing is an effective requirements assessment tool that allows the end user to easily pinpoint inconsistencies and incompleteness.

1. Introduction

Rigorous formal software construction is of paramount importance in several application domains. The Abstract, Synthesis and Transformation (AST) methodology [Winter97] developed by the High Integrity Software program at Sandia National Laboratories is an attempt towards this goal. AST aims at exploring correctness in the limit in the sense that unlike traditional approaches, correctness is not proved a posteriori, after the program is developed; rather, it is proved a priori, during each step of the transformation process. Even though AST provides a notational framework that facilitates the generation of formal specifications from informal descriptions, no research has been conducted on the validation of the resulting formalization. In this paper, we present an approach that will positively impact the validation of high consequence application domains. This approach is based on the automatic generation of a graphical representation of an informal (e.g., textual) document describing a reactive system's behavior. This graphical representation is an effective requirements assessment tool since it allows the end user to pinpoint inconsistencies and incompleteness' in the ambiguous informal document. It also allows the user to assess the conformity between the complex formal document produced by the specifier and the original informal requirements.

2. The AST Formal Notation

The AST formal model provides a notation that can be used to specify a class of reactive systems, called single-agent reactive systems. A single-agent reactive system is characterized by the fact that the controller is the only agent (i.e., the environment cannot initiate state change in the system), and all transitions are deterministic. A system \( s \) is modeled by means of a vector of monitored variables \( \bar{m} = (m_1, m_2, ..., m_n) \) and a vector of controlled variables \( \bar{c} = (c_1, c_2, ..., c_k) \). Controlled variables are independent, in the sense that the assignment of a value to a specific controlled variable does not restrict the value assigned to another controlled variable. Let \( M \) and \( C \) respectively denote the sets of all possible configurations of monitored and controlled variables. In order to specify correctly a system behavior, it is necessary to complement \( \bar{m} \) with a historical trace of the system. This trace is represented by a set of virtual monitored variables \( (v_1, v_2, ..., v_j) \). Hence, the observable state space of a

* Correspondence author.

This research is sponsored by Sandia National Laboratories.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
single-agent reactive system is defined as $S = \{ m, c | m \in M \land c \in C \}$. Given a state space $S$, the specification of system $s$ corresponds to a set of sequences of elements of $S$.

3. Statecharts

Because AST deals exclusively with the formalization of single-agent reactive systems, our first choice of a graphical representation for AST specifications is statecharts [Harel87]. Statecharts are extended finite state machines used to describe control aspects of reactive systems. They provide mechanisms to describe synchronization and concurrency, and manage exponential explosion of states by using state decomposition mechanisms.

In the statechart notation, a state is denoted by a box labeled in the upper left corner. Directed arcs are used to denote transitions between states and are labeled with a description of external stimuli and optionally with parenthesized conditions. A superstate can be repeatedly decomposed into substates in two ways, through the OR decomposition or through the AND decomposition. The OR decomposition reflects the hierarchical structure of a state machine and is represented by encapsulation. The AND decomposition reflects concurrency of independent state machines and is represented by splitting a box with dashed lines.

4. Automatic Generation of Statecharts

In our approach, a statechart is treated as a graph. Nodes\(^1\) in the graph correspond to states, and arcs correspond to transitions between states. A description of an object data structure is given in Figure 1. The decomposition of a superstate into substates is captured by a structure called a decomposition tree. The root of a decomposition tree corresponds to the system superstate; leaves correspond to atomic states. Each object in the tree can be decomposed through the AND or OR decomposition. Figure 2 shows an example of a simple statechart, and figure 3 shows its decomposition tree. Our algorithm proceeds as follows: first, the decomposition tree is traversed in order to determine the dimensions (and origin point) of every object in a recursive manner. If a node is a leaf then a drawing algorithm is called. This algorithm produces a drawing of the graph represented by the leaf and returns the dimensions and point of origin of the rectangle enclosing the drawing. If $v$ is an AND node then a recursive

\begin{verbatim}
class Object{
  ObjectName: String;
  width, height : Integer; // drawing's dimensions
  origin_x, origin_y: Integer; // drawing's origin point
  belongs: Object; // object's parent
  children: objectList; // decomposition objects
  decompositionType: AND / OR / LEAF
  incomingRelations: RelationList;
  outgoingRelations: RelationList;
  attributes(object's features): StringList;
}
\end{verbatim}

Figure 1. Object Data Structure.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{statechart.png}
\caption{Statecharts Diagram.}
\end{figure}

\(^1\)In the remaining of this paper we will use the words \textit{node} and \textit{object} interchangeably
algorithm computes the drawings of each child of \( v \) and places the drawings next to each other. If \( v \) is an OR node then a recursive algorithm computes the drawings and the dimensions of each child of \( v \), then computes basic drawing of \( v \). Figure 4 describes the recursive procedure \( \text{objectDrawing} \) in terms of the three basic drawing components (i.e., LEAF, AND, and OR).

\[
\text{leafDrawing}(\text{Object } o) \\
\text{Begin} \\
o.\text{origin}_x = 0; \\
o.\text{origin}_y = 0; \\
o.\text{width} = \text{WIDTH}; \\
o.\text{height} = \text{HEIGHT}; \\
\text{End}
\]

Figure 5. Procedure that determines the dimension of a trivial statechart object, where \( \text{HEIGHT} \) and \( \text{WIDTH} \) are predefined constants.

Let us now describe how to draw an AND node. An AND node represents an AND decomposition. Each of the components of the decomposition is drawn inside its own rectangle. The drawings of the various components are placed next to each other and are separated by dashed lines. The height of the rectangle of an AND node is equal to the maximum of the heights of the children’s rectangles; its width corresponds to the sum of the widths of the children’s rectangles. The algorithm that computes the drawing of an AND node is provided in Figure 6.

Finally, we describe how to draw an OR node. An OR node represents a hierarchical subgraph composed by the substates (children) of the parent state (node). Each of these substates is drawn inside its own rectangle. In Figure 3, objects 11, 12, and 13 form an OR decomposition of object 1. The drawing (and hence the dimensions of the enclosing rectangle) of an OR node is obtained by recursively performing a hierarchical drawing [Sugiyama81] [Di Battista99] on the node and each of its substates.

The algorithm that constructs the drawing of an OR node has two crucial steps: (i) first the drawing of each substate is obtained recursively; (ii) we perform a variant of the hierarchical drawing algorithm [Sugiyama81] [Di Battista99] since the drawings of the substates may have different dimensions. Although this algorithm can be clearly implemented in a single step, for ease of explanation, we choose to describe it as a two step process: (a) each substate is initially treated as a single node; (b) after a hierarchical drawing of each substate is obtained applying the
hierarchicalDrawing procedure, the proper dimensions of each substate are determined by recursively executing the objectDrawing procedure.

```
ANDDrawing(Object o)
Begin
  o.origin_x = 0;
  o.origin_y = 0;
  o.width = 0;
  o.height = 0;
  for i = 1 to #(o.children) do
    objectDrawing(child[i]);
  o.height = max{child[i].width};
  child.origin_x = o.width;
  o.width = ∑ child[i].width;
  end_do;
End
```

Figure 6. Procedure that determines the dimension of an AND decomposition object.

Consequently, once the real dimensions of each substate are determined, we modify the original hierarchical drawing by considering the dimensions of each node in the drawing. Once the node dimensions have been inserted into the hierarchy, then the total height and width of the parent object/state are determined. The hierarchy obtained in step (a) above is used to determine the coordinates of the origin of the rectangle of each object. The algorithm to determine the dimensions of an OR decomposition is shown in Figure 7. Figure 8 gives a brief description of an algorithm, called hierarchyDrawing, that computes a hierarchical (layered) drawing of any OR node. In order to simplify our description of the algorithm, we only consider hierarchical drawings drawn from left to right, where the first layer is the leftmost layer, as shown in Figures 9 and 11.

A similar approach can be employed if a user wishes to draw the drawing from top to bottom. First, we temporarily eliminate all cycles by inverting the direction of all edges that form the cycle. Second, the initial state of the graph is assigned as the node of the first layer. Subsequently, every other node $v$ is assigned to a particular layer that is determined by the length of a longest path from the start node to $v$.

```
ORDrawing(Object o)
Begin
  o.origin_x = 0;
  o.origin_y = 0;
  for i = 1 to #(o.children) do
    objectDrawing(child[i]);
  realDimensionHierarchyDrawing(o.children);
  o.width = hierarchy.width;
  o.height = hierarchy.height;
End
```

Figure 7. Procedure that determines the dimension of an OR decomposition object.

Therefore, after this step, every node will have an $x$-coordinate assignment. Finally, we apply a node ordering step that minimizes edge crossings within each layer. Using this ordering we can obtain the $y$-coordinate of each node. An example of a hierarchical drawing that depicts the layer assignment and node ordering is given in Figure 9.

After this preliminary hierarchical drawing (treating of the substates as points) is completed, we can compute the final drawing by incorporating the dimensions of each node. This can be achieved by adding a horizontal offset to each layer, where the offset will be the largest width dimension among all objects in a layer. The total width of the drawing will be the sum of the offsets of all layers.

```
hierarchyDrawing(ObjectList children)
Begin
  1. Eliminate cycles by inverting the direction of the cycle edges.
  2. Assign the child node that represents the initial state to the first layer of the graph.
  3. Further children nodes are assigned using the longest path-layering algorithm.
  4. Use dummy nodes to edges with span greater than 1.
  5. Use an ordering method to reduce the number of crossings between any two layers.
  6. Set the correct direction of the inverted edges.
End
```

Figure 8. Procedure that determines the hierarchy drawing if the substates.
Similarly, the height of the drawing is computed by adding a vertical offset to each node. The vertical offset is equal to the height of the drawing corresponding that node. The height of the complete drawing is the largest height of any layer. The origin coordinates of each node must be modified appropriately. A complete description of this algorithm is provided in Figure 10. Figure 12 shows the resulting drawing after we apply procedure `realDimensionHierarchyDrawing` to the drawing of Figure 10.

![Figure 9. Example of a left to right hierarchy.]

**Figure 9. Example of a left to right hierarchy.**

```
realDimensionHierarchyDrawing(ObjectList children)
Begin
    hierarchyDrawing(children);
    hierarchy.height = 0;
    hierarchy.width = 0;
    for i = 1 to depth(hierarchyDrawing of o.children) do
        begin
            1. layer[i].largestWidth = largest width among the objects in layer[i];
            2. if (layer[i+1] ≤ depth(hierarchyDrawing of o.children)) then
                Add layer[i].largestWidth as an offset to the origin_x of every object in layer[i+1];
            3. layer[i].height = summation of each object height at layer[i];
            4. if (hierarchy.height < layer[i].height) then
                hierarchy.height = layer[i].height;
            5. hierarchy.width = hierarchy.width + layer[i].largestWidth;
            6. Increase the origin_y of each object in layer[i] in order to deal with height of each object and avoid overlapping.
        end_do;
    End
```

**Figure 10. Procedure that modifies the hierarchy drawing to incorporate the real dimension of the objects.**

5. **Illustrative example**

In order to illustrate our approach, we use a simplified version of the early warning system described in [Harel90]. The informal description of the system reads as follows:

The early warning system (EWS) is a reactive control system that reads the value of an input sensor and compares it to a specified stored value. The system can receive three commands `setup`, `execute` and `reset`. Whenever the system is in the `setup` mode, a value is entered into the system, and is stored for further use. If the system is in the `execute` mode and if the sensor is connected, then the system reads the input value generated by the sensor. If this value is different from the stored value, then an alarm goes off and the system will wait to receive the `reset` command. At any time, the receipt of a `reset` command will move the system to the idle state where it will have to wait for the next command.

From this textual description we extract the decomposition tree depicted in Figure 13. The algorithm starts by calling procedure `ObjectDrawing` with argument `EWS`. `EWS` represents the root of the decomposition tree and is decomposed into objects `EWS_off` and `EWS_on`. Because this decomposition is of type OR, `ObjectDrawing` calls procedure `ORDrawing` which assigns the value (0,0) to `EWS`’ origin point, and proceeds by determining the dimensions of each of `EWS`’ children. The following is a complete run of our algorithm on the EWS example.

```
ObjectDrawing(EWS)
ANDDrawing(EWS)
    EWS.origin_x = 0;
    EWS.origin_y = 0;
    EWS.width = 0;
    EWS.height = 0;
ObjectDrawing(Interface)
ORDrawing(Interface)
    Interface.origin_x = 0
```

Where `Comparing` is at L1, and `Enable_Alarm` is at L2.

`Comparing.width`, `Enable_Alarm.origin_y` = 0;

`Comparing.origin.x = 0;
Comparing.height = constant_value;
Comparing.width = constant_value;

`Enable_Alarm.origin.x = 0;
Enable_Alarm.origin.y = 0;
Enable_Alarm.width = constant_value;
Enable_Alarm.height = constant_value;

`RealDimensionHierarchyDrawing(Comparing, Enable_Alarm)`

`HierarchyDrawing(Comparing, Enable_Alarm)`

`Setting_up.width = Get_Limit_Values.width + Store_Limits.width;
Setting_up.height = Get_Limit_Values.height;`
RealDimensionHierarchyDrawing(Idle, Executing, Setting_up)
HierarchyDrawing(Idle, Executing, Setting_up)

Where Idle is at L1, and Executing and Setting_up are at L2, represented by V1 and V2 respectively.

Idle.origin_x = Interface.L1;
Idle.origin_y = 0;
Executing.origin_x = Interface.L2 + Idle.width;
Executing.origin_y = 0 + Setting_up.height;
Comparing.origin_x = Comparing.origin_x + Executing.origin_x;
Comparing.origin_y = Comparing.origin_y + Executing.origin_y;
Enable_Alarm.origin_x = Enable_Alarm.origin_x + Executing.origin_x;
Enable_Alarm.origin_y = Enable_Alarm.origin_y + Executing.origin_y;
Setting_up.origin_x = Interface.L2 + Idle.width;
Setting_up.origin_y = 0;
Get_Limit.Values.origin_x = Get_Limit.Values.origin_x + Setting_up.origin_x;
Get_Limit.Values.origin_y = Get_Limit.Values.origin_y + Setting_up.origin_y;
Store_Limits.origin_x = Store_Limits.origin_x + Setting_up.origin_x;
Store_Limits.origin_y = Store_Limits.origin_y + Setting_up.origin_y;
Hierarchy.height = Setting_up.height + Executing.height;
Hierarchy.width = Idle.width + Executing.width;

Interface.height = Setting_up.height + Executing.height;
Interface.width = Idle.width + Executing.width;

EWS.height = Interface.height;
Interface.origin_x = EWS.width;
EWS.width = EWS.width + Interface.width;
ObjectDrawing(Sensor)

ORDrawing(Sensor)

Sensor.origin_x = 0;
Sensor.origin_y = 0;

ObjectDrawing(Sensor_not_Connected)

leafDrawing(Sensor_not_Connected)

Sensor_not_Connected.origin_x = 0;
Sensor_not_Connected.origin_y = 0;
Sensor_not_Connected.width = constant_value;
Sensor_not_Connected.height = constant_value;

ObjectDrawing(Connected)

ORDrawing(Connected)

Connected.origin_x = 0;
Connected.origin_y = 0;

ObjectDrawing(Waiting)

leafDrawing(Waiting)

Waiting.origin_x = 0;
Waiting.origin_y = 0;
Waiting.width = constant_value;
Waiting.height = constant_value;

ObjectDrawing(Get_Signal)

leafDrawing(Get_Signal)

Get_Signal.origin_x = 0;
Get_Signal.origin_y = 0;
Get_Signal.width = constant_value;
Get_Signal.height = constant_value;

RealDimensionHierarchyDrawing(Waiting, Get_Signal)
HierarchyDrawing((Waiting, Get_Signal)

Where Waiting is at L1, and Get_Signal is at L2.

Waiting.origin_x = Connected.L1;
Waiting.origin_y = 0;
Get_Signal.origin_x = Connected.L2 + Waiting.width;
Get_Signal.origin_y = 0;
Hierarchy.height = Waiting.height;
Hierarchy.width = Waiting.width + Get_Signal.width;

Where Sensor_not_Connected is at L1, and Connected is at L2.

Sensor_not_Connected.origin_x = Sensor.L1;
Sensor_not_Connected.origin_y = 0;
Connected.origin_x = Sensor.L2 +
Sensor_not_Connected.width;
Connected.origin_y = 0;
Waiting.origin_x = Waiting.origin_x +
Sensor.origin_x;
Waiting.origin_y = Connected.origin_y +
Connected.origin_y;
Get_Signal.origin_x =
Get_Signal.origin_x + Connected.origin_x;
Get_Signal.origin_y =
Get_Signal.origin_y + Connected.origin_y;
Hierarchy.height = Connected.height;
Hierarchy.width = Sensor_not_Connected.width +
Connected.width;

Sensor.height = Connected.height;
Sensor.width = Sensor_not_Connected.width +
Connected.width;
The algorithm has determined the coordinates as well as the height and width of each component; therefore, the final diagram (see Figure YY) can be drawn.

6. Conclusion and Future Work

Summary and Assessment
In this paper we presented a methodology for automatically generating statecharts from an informal description of software requirements. Our approach consists of two main steps: first, we extract a hierarchical decomposition tree from an informal description; this tree represents the recursive decomposition of superstates into substates. Next we draw the graph that models the statechart in a hierarchical fashion taking into account the structure of the decomposition tree. Our drawings enjoy several properties: 1) they emphasize the natural recursive hierarchical decomposition of superstates into substates; 2) nodes are placed on layers according to their distance from the local initial state; 3) the number of arc crossings is low; 4) the area of the final drawing is small. Therefore, in addition to capturing the intent of the words accurately, our graphical representations are also legible.

Related Work
Several tools have been developed to visualize software requirements. Statemate [Harel90] is a graphical tool used to represent reactive system requirements through three different views: structural, functional and behavioral. The structural view provides a hierarchical decomposition of the system into its components; the functional view describes the functions and processes of the system; the behavioral view specify the control activities and uses the statechart notation. ObjectTime [Objectime99] is a graphical tool used to visualize requirements written in the Room formal specification language. It uses graphics to describe the structure and behavior of reactive systems. Telelogic [O'Donnell99] is a tool suite used to visualize, develop, implement and test distributed realtime system software. It is based on formal languages such as SDL, and graphical notations such as Statecharts and UML. Artisan Real-Time Studio [Artisan99] uses a UML based notation to model a real-time systems functionality and architecture. The graphical representations offered by the tools discussed above are human generated.

Future work
Our prospects for future work include the development and implementation of methods that will allow the automatic extraction of decomposition trees from informal descriptions, and the automatic generation of AST specifications from statecharts.

References


Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

*Figure 11*. Hierarchy with real dimension of objects.
Figure 12. EWS Statecharts.

Figure 13. Decomposition tree for EWS.